Ana Bosque Arbiol

Filtering directory lookups in CMPS

Departamento

Informática e Ingeniería de Sistemas

Director/es

Llabería Griñó, José María
Viñals Yúfera, Víctor
Ibáñez Marín, Pablo Enrique

ISSN 2254-7606

Tesis Doctoral

FILTERING DIRECTORY LOOKUPS IN CMPS

Autor

Ana Bosque Arbiol

Director/es

Llabería Griñó, José María
Viñals Yúfera, Víctor
Ibáñez Marín, Pablo Enrique

UNIVERSIDAD DE ZARAGOZA
Informática e Ingeniería de Sistemas

2011

Universidad Zaragoza 1542

Departamento de Informática e Ingeniería de Sistemas

MEMORIA DE TESIS DOCTORAL

FILTERING DIRECTORY LOOKUPS IN CMPS

Autor: **Ana Bosque Arbiol**
Directores: **Pablo Ibáñez**
José M. Llabería
Víctor Viñals

Zaragoza

Grupo de Arquitectura de Computadores
de la Universidad de Zaragoza

Abstract

Nowadays, most computer manufacturers offer chip multiprocessors (CMPs) due to the always increasing chip density. These CMPs have a broad range of characteristics, but all of them support the shared memory programming model. As a result, every CMP implements a coherence protocol to keep local caches coherent.

Coherence protocols consume an important fraction of power to determine which coherence action to perform. Specifically, on CMPs with write-through local caches, a shared cache and a directory-based coherence protocol implemented as a duplicate of local caches tags, we have observed that energy is wasted in the directory due to two main reasons.

Firstly, an important fraction of directory lookups are useless, because the target block is not located in any local cache. The power consumed by the directory could be reduce by filtering out useless directory lookups.

Secondly, useful directory lookups (there are local copies of the target block) are performed over target blocks that are shared by a small number of processors. The directory power consumption could be reduced by limiting the directory lookups to only the directory entries that have a copy of the block.

Along this thesis we propose two filtering mechanisms. Each of these mechanisms is focused on one of the problems described above: while our first proposal focuses on reducing number of directory lookups performed, our second proposal aims at reducing the associativity of directory lookups. Several implementations of both filtering approaches have been proposed and evaluated, having all of them a very limited hardware complexity. Our results show that the power consumed by the directory can be reduced as much as 30%.

Contents

Chapter 1

Introduction

This initial chapter of this thesis summarizes the state of the art of coherence protocols, paying special attention to directory-based protocols and mechanisms that reduce the energy consumed by the coherence protocols. In the last section, we briefly motivate and describe the mechanisms proposed along this thesis as well as the organization of this document in the different chapters.

During the last decades, ever higher operating frequencies have been the main factor driving the performance growth of single-core processors. However, further increases in operating frequencies are increasingly hard to obtain with newer generations of technology. One of the main reasons is the impact of wire delays as feature sizes continue to shrink. To compensate, single-core processors have become increasingly more complex even leading to designs with inefficiencies in power/performance.

However, the density in the chips continues increasing, which have encouraged the development of chip multiprocessors (CMPs). Nowadays, most computer manufacturers offer CMPs such as the IBM Power7 with 8 cores with four threads each [30], the SUN Rainbow Falls with 16 cores with 8 threads each [47], the Intel Nehalem-EX with 8 cores with 2 threads each [32], the Fujitsu SPARC64 VIIIfx with 8 cores [37], the AMD Phenom II with 6 cores, and the SUN Niagara2 with 8 cores [29]. Figure 1.1 shows Niagara2 chip overview.

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, in all of them there is at least a local cache level per node and all of them support the shared memory programming model. Thus, every CMP implements a coherence protocol to keep local caches coherent.

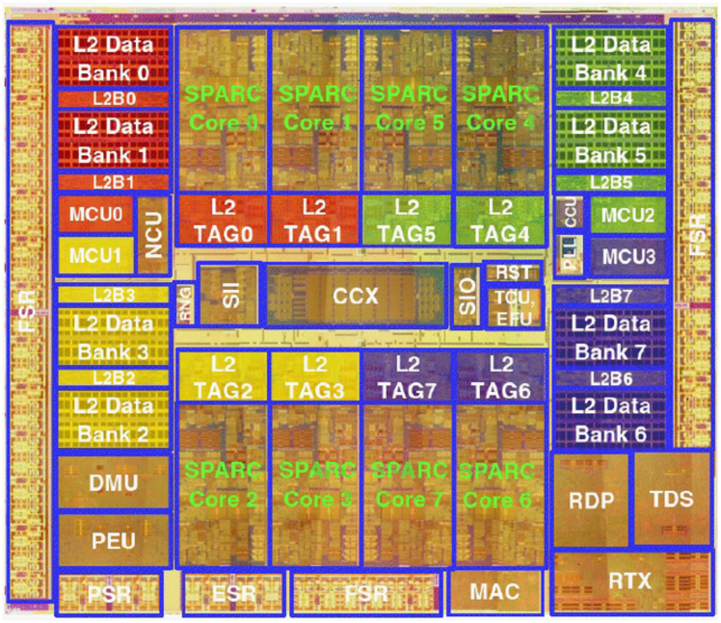


Figure 1.1: Niagara2 chip overview. In this chip we can easily distinguish 8 cores (*SPARC Core*), 8 shared cache banks (*L2 TAG and L2 Data Bank*), and the interconnection network on-chip (*CCX*).

1.1 Coherence protocols

A coherence problem arises in a multiprocessor because every processor access the shared memory through its local cache. Without a coherence mechanism, one processor can write a new value to a memory address and another processor can still access the old value cached in its local cache. Figure 1.2 shows an example of coherence problem. Processors P1 and P2 have local caches and are connected by a bus to main memory. X is a memory address that is read and written by the processors in the order specified by the numbers in the circles. First, P1 reads X. It needs to access main memory and it allocates X in its local cache. P2 also reads X from main memory and keeps X in its local cache. Then, P1 writes a new value in X, but it only updates the value of X in its local cache. Later, when P2 reads X, it reads the value of X from its local copy, so it reads an stale value.

The behavior of a memory system is correct if any read to a memory address returns the most recently written value to that memory address, that is, if the memory system is coherent and consistent. Coherence ensures that all the copies of a memory address in the system are coherent: a) writes to any memory address are propagated to all the processors in the system (write propagation property), and b) writes to a specific memory address are seen in the same order by all the processors (write serialization property).

Consistency defines the memory model of the system, that is, it defines the constraints on the order in which reads and writes to any memory location can appear to execute with respect to one another. Depending on the specific constraints applied, different consistency models can be distinguished. The memory consistency model is defined in the ISA of the architecture and programmers are responsible for writing correct programs under the specific constraints of the consistency model.

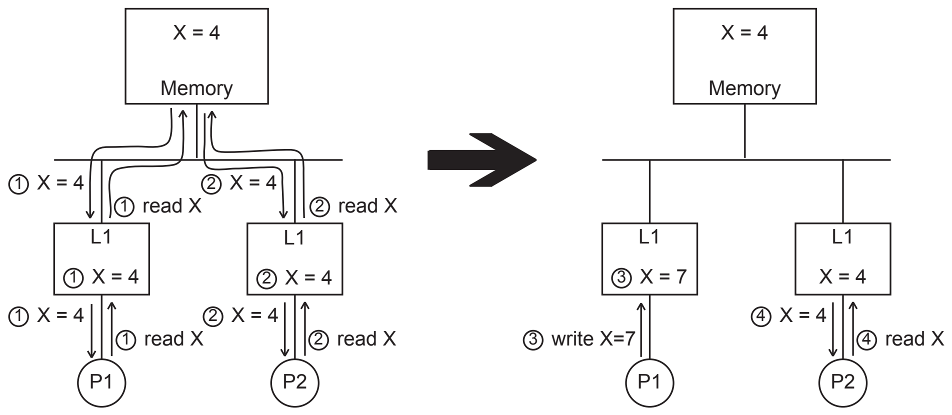


Figure 1.2: Example of cache coherence problem. Processors P1 and P2 are connected to main memory and both of them have a local cache. They read and write the value of X which is a memory address in the order specified by the numbers in the circles. First, both P1 and P2 read the value of X from memory and they allocate it in their local caches. Then, P1 writes X, but it only updates its local cache. Later, P2 reads X again, but it only access to the copy in its local cache so it gets a stale value.

The most straightforward consistency model is called sequential consistency (SC) [33]. It requires that the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program, like in a sharing-time system. This model is very simple to understand by programmers, but it restricts many of the performance optimizations that modern uniprocessor compilers and microprocessors employ. When using SC most of the memory latency is directly seen by processors as stall time. Thus, other consistency models more relaxed have been defined. For example, Total Store Ordering (TSO) [54] and Processor Consistency (PC) [23] allow reads to be performed before earlier writes in program order; Partial Store Ordering (PSO) [54] allows writes to be performed before earlier writes as long as they access different locations; and Weak Consistency [14], Release Consistency [20], and Relaxed Memory Ordering (RMO) [59] do not guarantee program order besides data and control dependences within a process. In these consistency models, barriers are used to guarantee SC in synchronization points.

The mechanisms to keep a multiprocessor memory system coherent are called cache coherence protocols. These protocols keep track of the sharing of each memory block. 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 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 memory accesses should be sent to all the local caches in the system.

Coherence protocols can also be classified as invalidation or update protocols. If the write propagation invalidates the local copies of the location written, the protocol is an invalidation protocol. On the other hand, if local copies are updated, it is an update protocol. In any case, next time other processor reads the location written, it gets the last value written either because the local copy has been updated or because the local copy is not valid anymore in its local cache so it is necessary to access either the main memory or the local cache that keeps the last written value.

The write policy of local caches is very important to define a coherence protocol. If the local caches are write-through, the last value written to any location will always be in main memory. Due to that, the only information that the protocol needs to keep for each block is which local caches keep a copy of the block. On the other hand, if the local caches are write-back, a block written by a store can be only located in the local cache of the processor performing the store. Thus, the coherence protocol keeps not only which local caches keep a copy of a block but the block state in each local cache since it is necessary to know if the block is updated in main memory or not.

Any coherence protocol is defined by the different states in which a block can be and the transitions among the states. A transition consists of the event that causes the state change and the actions that need to be performed in order to maintain coherence. The events causing state changes are both the requests performed by the processors (reads and writes) and the actions performed to maintain coherence (invalidations or updates depending on the coherence protocol). Figure 1.3 shows an example of a basic coherence protocol based on invalidation with write-back local caches called MSI. The name of this protocol stands for the name of the three states in which a memory block can be: *Modified* (M), *Shared* (S), and *Invalid* (I). A block is *Invalid* for a processor when its local cache does not keep a copy of the block. The state *Shared* identifies a block that is located in the local cache of a processor and that its value has not been modified so the value in main memory is up-to-date and there can be copies in other local caches. A block is classified as *Modified* for a processor when its local cache keeps a copy and it is the only valid copy in the system.

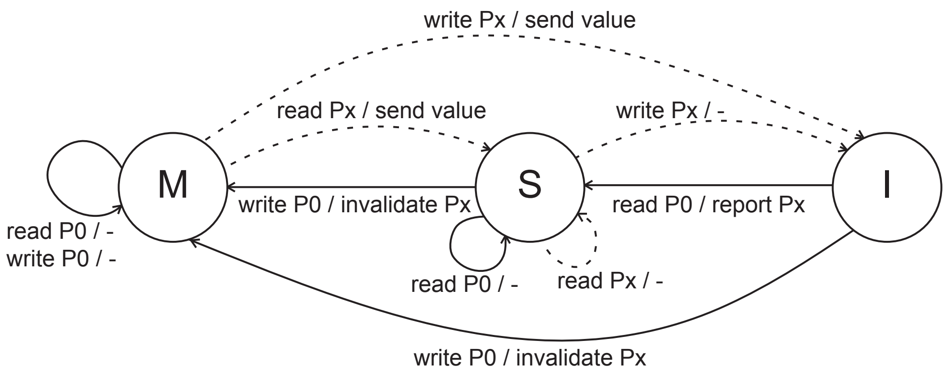


Figure 1.3: State transition diagram for an invalidation coherence protocol called MSI. This protocol has three states: *Modified* (M), *Shared* (S), and *Invalid* (I). The solid lines indicate state changes caused by reads or writes performed by the processor to which this state diagram belongs (read P0 and write P0). The dotted lines indicate state changes due to reads or writes performed by other processors (read Px and write Px). Every line is labeled (A/B) with the event causing the transition (A) and the action performed by the processor (B).

Different coherence protocols have been defined along the last decades. These protocols differ from each other in the write policy of local caches (write-back or write-through), the interconnection network they use, or the responsible for supplying a block (local caches or main memory). Some examples of protocols are: a) write-once [24], Synapse N+1 [19], Berkeley [31], and Illinois (or MESI) [46] which are based on invalidations, and b) Firefly and Dragon [38] which are based on updates.

1.1.1 Snoop-based coherence protocols

In snoop-based coherence protocols, the state of each block located in the local caches is maintained by the local caches, that is, the information about the state of any cached block is distributed. To know the state of any block (local caches sharing and state of the copy in main memory), it is necessary to gather in the information about that block in each local cache.

Local caches should be informed of any memory access performed by any processor that it is not a local hit to maintain the information about their local blocks updated. For example, any local cache ($L1_0$) has to the informed of a store performed in another local cache ($L1_1$) that does not keep a local copy of that block in exclusive state in order to invalidate/update $L1_0$'s local copy (if it exists), or any local cache ($L1_2$) needs to know when a load miss is performed in other local cache ($L1_3$) since it is possible that $L1_2$ keeps the only copy of that block in the system.

Figure 1.4 shows a multiprocessor with four processors with local caches that are connected by a bus. This multiprocessor keeps coherence by a snoop-based coherence protocol. In this example, P1 performs a write over a memory address X that is not located in its local cache. It is necessary to inform the rest of the local caches about this memory operation, so the write miss is broadcasted on the bus. Depending on the specific protocol and the block state it would be also necessary to send the memory access to main memory.

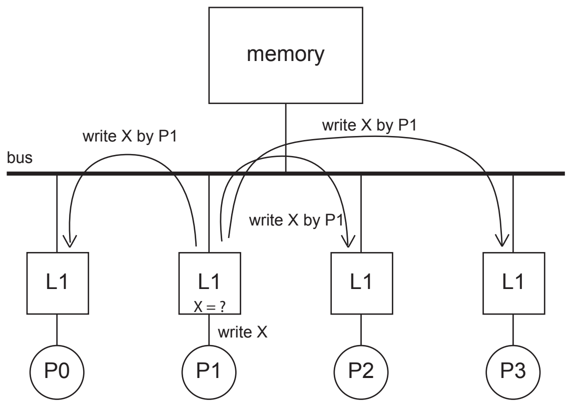


Figure 1.4: Snoop-based coherence protocol.

Snoop-based protocols became important in multiprocessors that connect the different processors with local caches using a bus since very few changes were necessary to implement the coherence protocol. In the local caches it was only necessary to add a mechanism to snoop the bus and maintain the state of local cached blocks. The reads and writes performed by the processors and sent through the bus keep the caches coherent and the serialization provided by the bus maintains coherency and consistency. However, a bus-based multiprocessor does not scale well.

The process of snooping can be implemented using the local cache tags. However, processor performance might be reduced since processor requests can be delayed by snoop requests. This situation gets worse with multithreading processors as they put more pressure on the local cache. To avoid this performance loss, the local cache tags are duplicated. The snoop requests access the copy of the local cache tags and only when there is a hit, the state of the corresponding block is accessed to be modified.

Any local cache miss and most writes generate a bus transaction either to request the block or to invalidate or update the local copies of the block. All these transactions perform lookups in all the local caches in the system. However, only a small fraction of these lookups hit in the local caches because the fraction of effective shared cache blocks is small and at any specific point of time these shared blocks are just shared by a few number of processors [41]. Snoop-based protocols consume a significant amount of energy to broadcast requests and perform snoop lookups. This energy is in general wasted because the snoop lookups typically miss in the local caches. In Section 1.2 several mechanism to reduce energy consumption in snoop-based protocols are described.

1.1.2 Directory-based coherence protocols

Directory-based coherence protocols maintain the state of every block of main memory in a structure called directory. Any processor request must access the directory and get the state of the target block. Once the state of the block is read, the appropriate actions to maintain coherence must be performed by the directory and the processors involved in the request. Figure 1.5 shows the different actions carried out by a directory-based coherence protocol in a multiprocessor with write-back local caches when a processor performs a read request over a dirty block. Processor P0 reads location X. A dirty copy of X is located in the local cache of processor P3 and this information is kept in the directory. P0 sends a read request to the directory. The directory responses to P0 indicating that P3 keeps the current value of X. Then processor P0 sends a request to P3 and P3 sends the data to P0 and a revision message to the directory in order it updates its state. The final state of X in the directory and the local caches is identified in the figure with the number four inside of a circle.

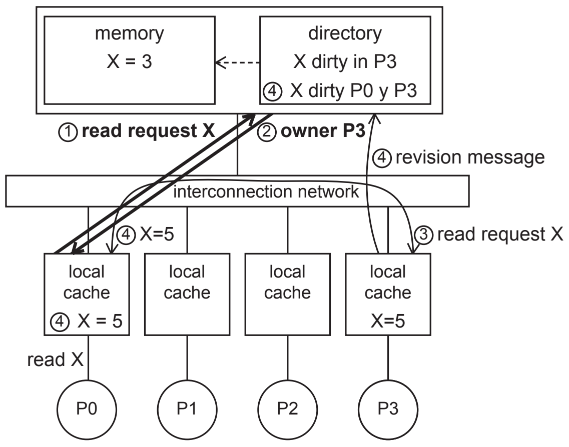
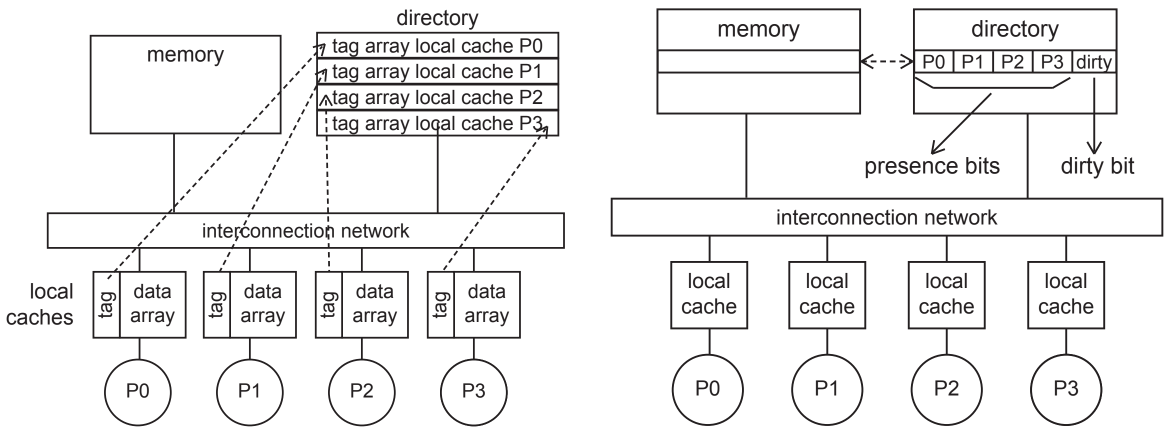


Figure 1.5: Directory-based coherence protocol in a multiprocessor with write-back local caches.

There are two basic schemes to implement a directory: duplicate tags and full-map [56, 9]. Figure 1.6(a) shows a duplicate tag directory scheme. The directory is a separate structure that keeps a copy of all tags and state bits of each line in the local caches [6]. Figure 1.6(b) shows a full-map directory scheme. Each memory block keeps a presence bit vector and a dirty bit. Each bit of this vector corresponds to one processor or to one local cache. An additional bit is used to identify if the local copies of the block are dirty, that is, the block is up-to-date in memory. The two directory schemes are logically equivalent.



(a) duplicate tag directory (b) full-map directory

Figure 1.6: Basic directory scheme implementations.

Both directory schemes can be distributed among interleaved memory modules. In a full-map directory, the interleaving for a directory and memory modules is the same. In a duplicate tag directory, the size of the directory depends on the interleaving. If it exists an injective mapping of the cache sets to memory modules then the total directory size is the local cache tag size, e.g., when memory modules are interleaved by the lower address bits of the cache index [55, 57]. In other cases the directory size is bigger than the size of the local cache tags [44].

Several directory schemes have been proposed to reduce the high storage requirements of full-map directories. Some proposals reduce the number of entries in the directory based on the memory hierarchy organization, while other schemes reduce the number of bits per directory entry according to the program behavior.

Sparse directory schemes have less entries than full-map directories [25, 45]. Based on the fact that the number of blocks stored in local caches is much smaller than the number of memory blocks or blocks in a shared inclusive cache, sparse directories are organized as caches. Their entries are dynamically allocated to keep coherence information about the blocks kept in the local caches. The number of blocks that can be cached simultaneously is restricted by cache organization parameters (number of sets and associativity). The management of conflicts increase the miss rate.

Limited pointer directory schemes reduce the number of bits of each directory entry [60, 1, 10]. Since only a few local caches have a copy of a specific block, it is possible to only maintain, instead of a presence bit vector, a fixed number of pointers, each pointing to a processor that currently caches the block. The number of concurrent copies of a block is limited by the number of pointers. When the number of pointers is exhausted and a new processor requests a new copy of the block, overflow strategies are used [4, 10, 42, 1, 25, 34]

Recently, Ferdman et at. design a new directory scheme based on sparse directories, but preventing the set conflicts that diminish performance and reducing the area cost [18].

This directory scheme is intended for many cores since it keeps the power consumption and area utilization nearly constant regardless of core count.

Differences between duplicate tag directory and other directory schemes mentioned above arise in size, lookup method, and retrieved information in a lookup operation. In a duplicate tag directory the lookup is associative, while in a full map directory a lookup is performed using the address to index the structure. The lookup associativity in the duplicate tag directory is the aggregate associativity of the caches it tracks, that is, all possible locations are checked like in a snooping protocol. 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. Concerning size, the duplicate tag directory uses the smallest explicit representation of all blocks contained in local caches.

Any coherence request in a snoop-based protocol looks up every local cache. However, most of these lookups are not necessary since the target block is not located in any local cache or it is allocated only in a small number of local caches. These pointless lookups waste energy. Using a full-map directory this problem can be solved since only the local caches that effectively have a local copy of the target block are looked up. Despite accessing only the local caches involved in the coherence request, it requires the local cache tags to be duplicated in order to not delay processor requests causing a performance loss. The problem when using a full-map directory is the directory structure itself. This structure is accessed by any coherence request (dynamic energy consumption) and some schemes have high storage requirements (leakage power). Moreover, any coherence action requires several messages through the interconnection network. Although these messages are point-to-point and they do not have a high energy consumption, they might introduce an important delay in every coherence request.

A duplicate tag directory has smaller storage requirements. However, the directory lookups consume a significant amount of power since they are associative. In fact, all possible locations are checked like in a snooping protocol, though now only the directory structure is accessed. The advantage of a duplicate tag directory is that a directory lookup identifies not only the local caches that have a copy of the target block, but the way in the local caches in which the block is located. Thus, invalidation messages only consist of the index and the way of the local cache that has to be invalidated. Moreover, as invalidation messages does not need to access the local cache tag array, the processor requests are never delayed. So it is not necessary to duplicate the local cache tag array.

1.1.3 Other proposals

Recently, Zebchuk et. al. [61] propose a directory structure that use an implicit and conservative representation of the blocks located in the local caches instead of the explicit representation used in conventional directory schemes. The structure organization is like a duplicate tag directory but the blocks located in each set of the local caches are represented using a bloom filter. In this proposal, the storage overhead of the directory structure is smaller than in conventional directories organizations, but it requires to introduce several extensions to a base coherence protocol.

This proposal is in between snoop-based and directory-based protocols. Unlike directory-based protocols, the tagless directory is not able to limit local cache lookups performed by coherence requests to the ones performed over blocks present in the local caches. However, not all the local caches are looked up for every coherence request as it is necessary in a snoop-based protocol. The tagless directory requires a small structure that scales nicely with the number of cores and that does not have a high energy consumption. However, to maintain the directory up-to-date, it is necessary to add new coherence messages or at least to add more information to the coherence messages performed by a directory-based protocol.

1.2 Mechanisms to reduce power consumption in coherence protocols

During the last decade several techniques to filter out coherence actions 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 filter structure is necessary. This structure is either placed together with the local caches or distributed in the on-chip network.

First, we will go over proposals that use a filter which is placed together with the local caches. The filters are small structures that are accessed before any snoop-induced local cache tag lookup is performed and try to avoid useless local cache lookups. They determine whether the target block might be located or definitely it is not present in the local cache. The former case requires a local cache lookup while, in the latter case, the cache lookup is known to ”miss”, that is, it is useless and it can be avoided.

Jetty [41] proposes for a SMP system three different filters: a) a filter that keeps blocks not present in the local cache (got from last snoop-induced tag lookups that missed), b) a filter that keeps a superset of the blocks in the local cache (updated with any allocation or replacement), and c) a mix of the two first filters. Ekman et al. [15] analyze the power performance of Jetty in a CMP system. They conclude that, as the local cache sizes are smaller than in a SMP system, the filter and the local caches energy consumption are similar. Thus, Jetty is not an interesting mechanism for CMP systems.

Salapura et al. [48, 49] proposes a mechanism similar to Jetty, but their main interest is to filter useless snoop-induced lookups not to reduce power consumption but to improve performance. Snoop-induced lookups can delay processor requests causing a performance loss, so if the number of snoop-induced lookups is reduced, less processor requests will be delayed and processor performance will improve. The filter used is a Stream Register which keeps a superset of the blocks in the local cache. A Stream Register consists of a set of base addresses with a corresponding bit mask. The bit mask identifies which base address bits are significant. An address is consider to be present in a Stream Register if it matches all base address significant bits from any entry.

When the goal of the filter is to avoid broadcasts, the filters are accessed before any coherence request is sent by any processor. Depending on the specific mechanism, the filters determine either whether the target block is shared by any other processor or not, or the subset of processors that shared the target block. If the target block is not shared at all, broadcast is not necessary. If the subset of processors sharing the target block is identified, the broadcast can be replaced with as many point-to-point messages as necessary.

PST (Page Sharing Table) [16] is a unit integrated with the TLB that keeps track for each page of how many blocks the local cache has and which processors have blocks from that page. Using this structure coherence requests are snooped only by those local caches that have blocks located in the same page as the target block.

RegionScout [40] identifies which regions are not shared. A region is defined as a continuous, aligned memory area. RegionScout uses two structures: NSRT (Not Shared Region Table) and CRH (Cached Region Hash). The first one keeps track of all regions that are identified as not shared by any other processor. A not shared region is identified after a coherence request over a block included in that region. Processors indicate for any coherence requests that reach them if any block of the same region of the target block is located in its caches or not. If every processor classifies the region of a block as not shared, then that region is included in the NSRT. The CRH is a Bloom filter that keeps a superset of all regions locally cached. It is used to identify not shared regions by a processor without accessing its local cache. An access to a memory address that belongs to any region in the NSRT does not require a broadcast since that memory address is not shared by any other processor.

Cantin et al. [8] propose a filter that, like RegionScout, keeps sharing region information for each processor. This proposal requires a structure called RCA (Region Coherence Array). This structure keeps the state of any region of which there is at least a locally cached block. As the information kept for any region is precise and it indicates not only the presence of a block belonging to that region but the state of it, the proposed structure is bigger than in RegionScout. However, they are able to remove more coherence requests.

Strauss et al. [53] analyze how to implement adaptive forwarding algorithms in network rings using filters that indicate the probability of a snoop-induced lookup to hit in the local cache. These filters are similar to the ones in the previous proposals [41, 40, 8].

All previous proposals use both processors requests (reads and writes) and coherence requests performed by other processors to keep the filter structures up-to-date. There are also some proposals that use filters that classify blocks as shared or not shared based on program semantics. Dash et al.[12] proposes to add information regarding shared memory regions during software development and compilation. If a processor request access a block that belongs to a region identified as non-shared, none coherence request will be necessary. Ballapuram [5] proposes a similar idea. It describes two different techniques. The first one, called SSP (Selective Snoop Probe), labels blocks as stack and non-stack. Stack blocks are never shared so no coherence actions are performed for them. For the non-stack blocks, a superset of the locally cache blocks is kept using a bloom filter to filter snoop-induced local cache tag lookups. The second technique, called ESP (Essential Snoop Probe) labels at compilation time variables (stack, global, and heap) as blocks that do not need coherence actions since they are not shared.

There are also 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 located in the local caches of the processors reached from a specific port, so useless broadcast messages are not sent. Jerger [28], 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.

1.3 Our proposal

We propose filtering mechanisms to reduce the power consumed by a duplicate tag directory in a CMP which has multithreaded processors with local instruction and write-through data caches, and an inclusive shared cache. (A detailed description of the CMP is in Section 2.1.)

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 [52, 11]. 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 [29].

Every directory lookup requires an expensive associative lookup in the duplicate tag structure. However, we analyze the behavior of several programs and we conclude that: a) an important fraction of directory lookups are useless since there are no local copies of the target block, and b) the directory lookups that are useful are performed over blocks that are shared by a small number of processors. Based on these observations, we propose two filtering mechanisms. The first filtering mechanism tries to identify in advance directory lookups that are useless and filters them out. This filter reduces significantly the number of directory lookups performed (Chapter 3). The second filtering mechanism tries to identify which processors have local copies of the target block of a directory lookup, that is, the sharers of the target block. Once the sharers are determined, the directory lookup only checks the directory entries corresponding to those processors, and therefore, the directory lookup associativity is reduced (Chapter 4).

The rest of this dissertation is organized in four chapters. Chapter 2 describes the methodology used in the evaluation of the proposed filtering mechanisms: CMP model chosen, simulation infrastructure developed, and benchmarks used. Chapter 3 describes the first filtering mechanism which reduces the number of directory lookups performed. Chapter 4 introduces the second filtering mechanism proposed which reduces the directory lookup associativity. Finally, we conclude this thesis in Chapter 5 with a summary of the contributions and publications derived from this research.

Chapter 2

Methodology

Along this chapter we describe the evaluation methodology. First, we introduce the CMP model that we assume in all the experiments carried out to evaluate the different mechanisms proposed in this thesis. Next, we describe the most important characteristics of the simulation infrastructure. Finally, we present the benchmarks used and we introduce their main characteristics.

2.1 CMP model

Figure 2.1 shows the CMP configuration we assume along this thesis. 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.

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 mapped to it. Table 2.1 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 [6]), 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

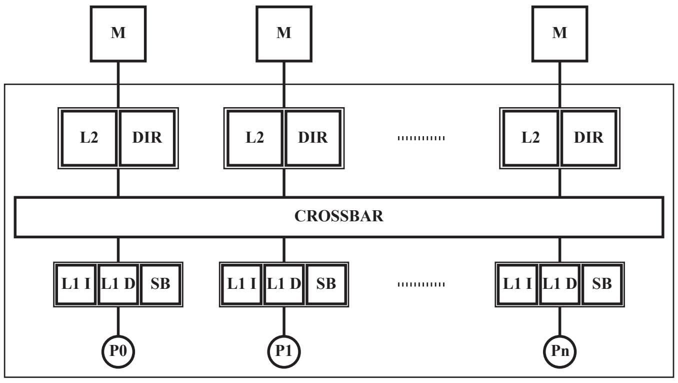


Figure 2.1: CMP model with a first-level local cache per core (instruction cache, data cache and store buffer) and a second-level shared cache divided in several banks.

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

Table 2.1: Memory hierarchy parameters

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]

Like in Niagara 2 [55], 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).

2.1.1 Coherence Directory

We assume a directory similar to that of Niagara 2 [55], which consists of a duplicate 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

[1]The bandwidth requirements to implement local write-through caches is so high in many-core systems (which is not the target of this thesis), 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 thesis.

structure whose area requirements is `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.

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.

Directory Organization

Duplicate instruction and data directories have a similar structure, but they are accessed in a different way depending on the kind of memory access to the shared cache that performs the access to the directory.

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. Figure 2.2 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`. Figure 2.3 shows the same system as Figure 2.2, but now the local cache block size is half the shared cache 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

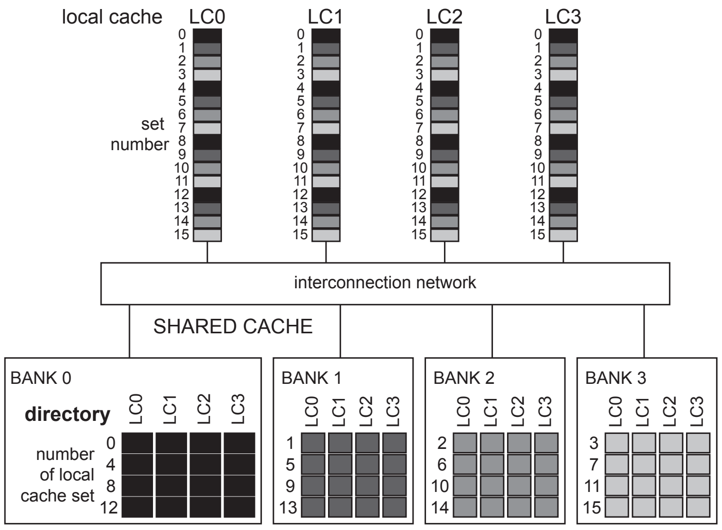


Figure 2.2: Mapping of local cache blocks to shared cache banks and directories when the shared cache and the local caches have the same block size.

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 2.1). 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 require to look up all these blocks. Figure 2.4 shows how the directory is organized in 16 different panels (using SUN's terminology [55]) 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 $\times$ 8 sets $\times$ 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.

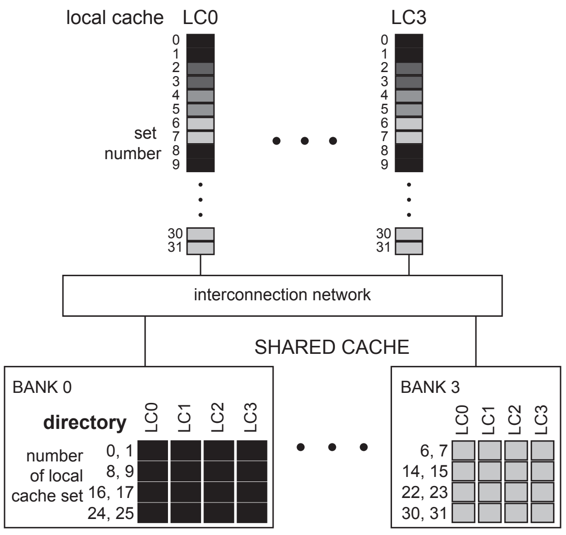


Figure 2.3: Mapping of local cache blocks to shared cache banks and directories when the shared cache block size is twice the local cache block size.

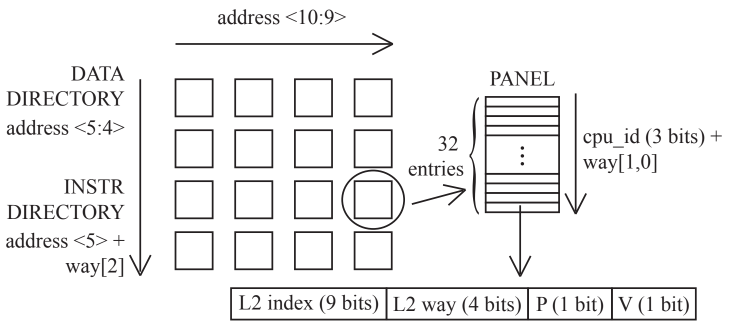


Figure 2.4: Data or instruction directory structure of Niagara 2 and how they are accessed.

Directory Operation

Any access to and any eviction from the shared cache performs a lookup in the directory. Along this thesis, we call "memory operations" to all these accesses and evictions from the shared cache, namely, loads and instruction fetches that miss in the local caches, stores and evictions from the shared cache. 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

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 2.2: 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.

all the local instruction caches. The address of this 16B block determines a specific set in the local instruction caches. 64 comparisons are performed since there are 8 local instruction 8-way associative caches.

- *Ifetch-miss*: It is an instruction fetch miss in the local instruction cache. The behavior is the same as in a load-miss 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. 64 comparisons are performed since there are 8 local data 4-way associative caches.

- *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*: It is an eviction from the shared cache. As inclusion is enforced in the system, the shared cache victim block must be 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 are invalidated. So, 128 comparisons are performed in each directory.

Table 2.2 summarizes the actions performed for every memory operation and the number of comparisons performed by 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.

2.2 Simulation infrastructure

This section presents the simulator used to model the CMP described in the previous section and the cache power model chosen to estimate the dynamic energy and the leakage power of the directory and the proposed filters.

2.2.1 Simics

We use a Simics-based simulator. Simics is a platform for full-system multiprocessor simulation [36]. This thesis focuses on CMPs which are intended for multiprocessor workloads such as databases or web servers. These applications highly depend on the operating system memory management and scheduling, so they cannot be simulated independently from the operating system. As a result, we chose Simics as simulation platform since it is capable of running unmodified commercial OSs and applications.

Simics simulates processors at the instruction-set level, including the full supervisor state. It supports a wide variety of processors such as UltraSparc, Alpha, x86, PowerPC, MIPS, or ARM. We configured Simics to model a SPARC V9 target system 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.

Simics is only a functional simulator by default, but it can be extended with detailed processor and memory system timing models. We decide to use in-order, blocking processors that perform one instruction per cycle. As all our contributions are intended to improve the power consumption of the coherence directory, which is part of the memory hierarchy, we require to model the timing of each memory system access.

We connect a timing model to the memory space of Simics accessed by all the instruction fetches and data accesses. We completely developed this timing model using the Simics API. This timing model accurately models the memory hierarchy of our CMP model. We model the different components of the hierarchy, their timing, and the contention in the different shared components such as the interconnection network, the shared cache banks, or the local caches shared by several threads. The main characteristics of this memory hierarchy are detailed in Section 2.1 and its parameters are described in Table 2.1. The consistency memory model of this memory hierarchy is Total Store Order (TSO).

2.2.2 CACTI

We use CACTI 6.5 [43] to estimate the dynamic energy and the leakage power for the shared cache, the coherence directory and the proposed filters. CACTI is an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. We modified CACTI to model CAM structures because the coherence directory is implemented as a CAM structure. These modifications were based in the CAM structures implemented in McPAT. McPAT (Multicore Power, Area, and Timing) is an integrated power, area, and timing modeling framework for multithreaded, multicore, and manycore architectures [35].

The shared cache, the coherence directory and the proposed filters were modeled using a 65nm 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.

2.3 Bechmarks

We use SPLASH2 and Specweb2005 to evaluate the different filtering mechanisms proposed in this thesis. Both SPLASH2 and Specweb2005 are benchmark suites that consists of several workloads. Along this thesis, we show the different metrics for all the workloads and also the average for all the workloads in each benchmark suite, that is, we compute the average for SPLASH2 and for Specweb2005. Depending on the metric, the average is the arithmetic or the geometric mean.

This section introduces the main characteristics of SPLASH2 and Specweb2005 and how we configure them. We also present a characterization analysis of the Apache web server using Specweb2005 as URL request generator.

2.3.1 SPLASH2

We use the applications of the SPLASH2 benchmark suite [50]. In order to adapt the SPLASH2 workloads to our simulated scenario, we scaled the input dataset up as proposed by Monchiero et al. [39]. 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 2.3 shows the applications used, the corresponding datasets, the billions of cycles and executed instructions, the local data and instruction cache miss rate (L1 data miss rate and L1 instr miss rate) per instruction, and the shared cache miss rate (L2 miss rate) per instruction.

benchmark	dataset	instr (10^9)	cycles (10^9)	L1 data miss rate	L1 instr miss rate	L2 miss rate
barnes	64K particles	4.97	0.62	2.48	0.004	0.045
fmm	64K particles	9.57	1.20	1.51	0.016	0.052
ocean	1026x1026	5.99	0.91	6.21	0.008	1.746
radiosity	-largeroom, -ae 5000 -en 0.050 -bf 0.1	7.45	0.94	3.13	0.177	0.003
raytrace	balls4	5.77	0.79	9.44	0.008	0.001
volrend	head	0.63	0.08	1.67	0.030	0.010
water-nsquared	2192 particles	13.79	1.72	0.80	0.001	0.004
water-spatial	4096 particles	4.02	0.50	1.32	0.001	0.002

Table 2.3: SPLASH2 benchmarks, the corresponding datasets, billions of cycles and instructions executed, local data and instruction miss rate per instruction, and shared cache miss rate per instruction.

2.3.2 Specweb2005

Specweb2005 [27] is a URL request generator. We use Specweb2005 to analyze and evaluate our proposals for the Apache web server. In this subsection we first describe Specweb2005 and the Apache web server configuration. Then, we explain how we run our simulations to analyze only the behavior of the Apache web server. Finally, we show a characterization of the Apache web server using Specweb2005.

Specweb2005 description

Specweb2005 is a software benchmark product developed by SPEC. It is designed to measure a system's ability to act as a web server servicing static and dynamic page requests. It is composed for three different workloads: `banking`, `ecommerce`, and `support`, in an attempt to cover the different scenarios in which a web server is commonly used.

- **Banking** simulates a bank web site in which the clients, once logged in, are able to do operations like getting the list of all the transactions done in the last month, checking the state of a payment or transferring money. All requests in this workload are SSL based (Secure Sockets Layer).
- **Ecommerce** is designed to simulate a Web server that sells computer systems. Each client could pass by three different phases: browse, customization and purchase. Only the last phase is SSL based.
- **Support** is to test the ability to download large files. It simulates a vendor's *support* web site. In this workload SSL is never used. The user will browse the catalog or search in it until he/she finds the file of interest and then downloads it.

In Specweb2005 the sequence of operations made by a client is defined by a Markov chain. Every operation is done in a different web page, that is, the web page in which the login is done is different from the one in which the clients check the last transactions in their bank accounts. The Markov chain states are the different web pages or operations, and the transactions among them are the different options that the clients could take.

Another important parameter is the thinking time which is the time between two consecutive requests made by the same client, that is, the time that the client needs to read the web page that has just received before requesting the next one. In all the workloads of Specweb2005 we set the thinking time to zero in order to be able to see a bigger number of transactions in the few seconds of the benchmark that we are simulating.

Specweb2005 has four major logical components: the clients that send HTTP requests to the server and receives the responses, the prime client that initializes and controls the execution, and collects statistics, the web server that handles the requests of the clients, and the back-end simulator (BeSim) that emulates a back-end application server.

Apache web server configuration

We use Apache 2.0.63 [26] configured with the process model *worker*, which implements a hybrid multi-process multi-threaded server in which a single control process (the parent) is responsible for launching child processes. Each child process creates a fixed number of server threads as well as a listener thread which listens for connections and passes them to a server thread for processing when they arrive. In our configuration the number of child processes is fixed along the whole simulation to 4 with 150 server threads each. We also configured Apache for using *posix threads* for all the synchronization tasks.

Simulation environment

Simics is capable of running unmodified applications, so we only need to set up the same environment of Specweb2005 as in a real machine. We require a machine for the Apache web server, another one for the clients, and one more for the BeSim. Simics allows us to simulate these machines and connect them by a simulated network. Table 2.4 describes the different machines that we are simulating and also the software applications that are necessary in each machine. We are only interested in the behavior of the Apache web server. Thus, we only take statistics from the machine running the Apache web server.

	Description	**Hardware**	**Software**
Clients	The clients run in this machine	4 processors	JRE HotSpot 1.5.0
Besim	Back-end simulator that emulates a back-end application server	4 processors	Apache 2.0.55 + library FastCGI 2.4.0
Web server	System Under Test in which the Apache web server is running	8 processors	Apache 2.0.55 + libraries php 5.1.2 & OpenSSL 0.9.8a

Table 2.4: Simulated machines and software applications necessary for Specweb2005

Table 2.5 indicates how many simultaneous sessions are running for each workload. Web servers present high time and space variability [3] and, on top of that, we cannot 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. All these simulations are run after fast forwarding the period of ramp up of Specweb2005 [51]. 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 2.5 shows the number of web transactions performed, the average billions of instructions executed for each workload, and the number of simulation runs. We use the mean of the simulations in all the graphics in this thesis in which Specweb2005 appears.

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

Table 2.5: Specweb2005 workloads, the corresponding simultaneous sessions, the number of web transactions, billions of instructions executed and number of simulation runs.

Apache web server characterization with Specweb2005

A commercial workload like the Apache web server has a behavior different from more traditional scientific applications. We decide to analyze the memory behavior of Apache using Specweb2005 as URL request generator. We measure the shared cache miss rate per data access, the breakdown of execution time, and the number of web pages requested.

As we do not require a detailed memory hierarchy, we decide to perform this characterization using Simics and VASA [58]. VASA is a configurable high-performance multiprocessor simulation package for Simics that can model all the components of the memory hierarchy, but it does not model contention. The memory hierarchy is slightly different to the memory hierarchy used in the rest of the thesis. Table 2.6 presents the most important parameters of this memory hierarchy. All workloads were executed with 200 simultaneous clients and only one simulation run per workload of 10 billion cycles was performed.

Processor speed	1.2 GHz
store buffer size	4 entries
store buffer line size	32 bytes
store buffer read latency	1 cycle
L1 instruction cache	perfect
L1 data cache size	32 KB
L1 data cache associativity	4-way
L1 data cache line size	32 bytes
L1 data cache latency	1 cycle
L1 data cache write policy	write-through
L2 data cache size	1 MB
L2 data cache associativity	4-way
L2 data cache line size	32 bytes
L2 data cache latency	12 cycles
L2 data cache write policy	write-back
memory access latency	150 cycles
network bandwidth	8 bytes/cycle
network latency	12 cycles

Table 2.6: System on-chip parameters

Figure 2.5 shows the shared cache miss rate per instruction fetch and classifies the misses into three basic categories based on the classification presented by Dubois et al. [13]: cold, capacity (replacement), and sharing. A miss is classified as a cold miss if the block has never been in this processor's cache before. A miss is a sharing miss if the last time the block was in the cache it was invalidated due to a store performed by another processor. All other misses are capacity misses.

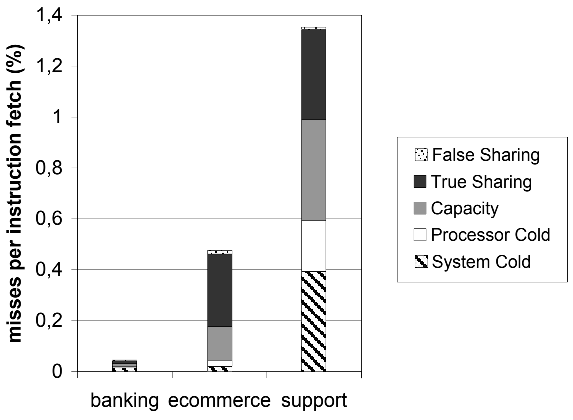


Figure 2.5: Shared cache data miss rate for the three workloads of Specweb2005.

The classification between capacity and sharing depends on the size of the cache. As we are interested in knowing the true communication in the application independent of the cache size, we change a little the classification assuming an infinite cache for coherence. In this classification a miss is a sharing miss if the block has been modified since the last time it was in the processor's cache. All other misses are capacity misses.

We decided to split the cold misses in two different categories: system cold misses and processor cold misses. In the first ones, the block has never been written by any processor.

We also split the sharing misses in two categories: true sharing and false sharing misses. A sharing miss is classified as true sharing if the bytes accessed by the processor during the lifetime of the block in the cache have been modified by another processor since the last time the block was in the cache. It is classified as false sharing miss in any other case.

Figure 2.6 shows the breakdown of execution time into six components: cpu time, shared cache hit access, memory access, cache to cache service, store buffer full stall cycles, and emptying the store buffer due to the execution of atomic instructions. The upper part of the graph shows the breakdown for the cycles executed in user level and the bottom part presents the same data, but for privilege level. The number on top of the privilege level cycles is the percentage of idle time. We get this number from the statistics of the *mpstat* command. We use this command along the whole simulation since it report statistics about the usage of the cpus: user, system, waiting, and idle time.

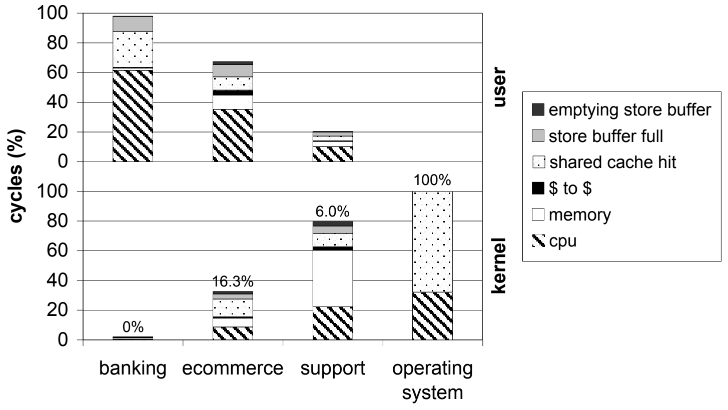


Figure 2.6: Breakdown of cycles for the three workloads of Specweb2005.

Figure 2.7 shows the number of web pages requested by the clients and the number of extra files included inside them. When a file is sent from the server to the client we always call that a transaction without taking into account if it is the main page or just a file included in it.

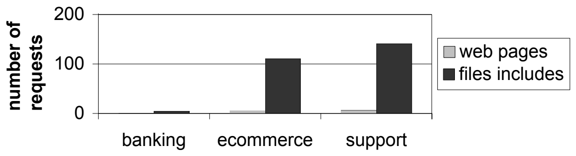


Figure 2.7: Number of web pages and files included requested per 100 billion cycles for the three workloads of Specweb2005.

Chapter 3

Reducing Directory Lookups

This chapter describes a filter which reduces the number of directory lookups performed. This filter takes into account that, in general, the sets of memory addresses of instructions and data are disjoint and it classifies blocks as data or instruction blocks.

We propose two different filter implementations:the Instruction-Data filter (ID filter) and the Decoupled filter (DPL filter). The ID filter keeps explicit filtering information for every block in the shared cache identifying it as a data or an instruction block. The DPL filter keeps a superset of the blocks that belong to each stream (data or instruction). We require a D-DPL filter for the data directory and a I-DPL filter for the instruction directory. The information kept in these filters is decoupled from the shared cache organization.

Our results show that, for SPLASH2, the proposed filters reduce the number of directory lookups performed by 60% while directory 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 ID filters (the I-DPL filter).

This chapter is organized as follows. Section 3.1 motivates why the proposed filtering mechanism can work. Section 3.2 describes the filtering mechanism and details the proposed implementations. Section 3.3 evaluates the most important characteristics of the proposed filters. Finally, Section 3.4 concludes this chapter.

3.1 Introduction

The sets of memory addresses of instructions and data (values of program counters of the executed instructions and addresses of referenced data) are disjoint. The computer memory is organized into segments: code segment, stack segment, and heap segment which are mapped to different memory regions. The code of a program is allocated in the code segment, while the static and dynamic variables used in the functions of this program are allocated in the stack segment and the heap segment, respectively. Thus, data and instructions addresses are expected to be different. However, though it is the common situation, it is not always true. There are special cases in which data and instructions are allocated to the same region so a cache block belongs to the data and the instruction stream simultaneously. Two examples of this situation are self-modifying code and constants located in the code segment:

- *Self-modifying code* Self-modifying code is code that alters its own instructions while it is executing. This code performs stores over blocks that later will be accessed to fetch the instructions to execute, that is, first it accesses a block as data and then as an instruction.

- *Constants located in the code segment* Sometimes compilers locate program constants along with the instructions that use them. Thus, a cached block could contain instructions and constants (data).

If any of the previous situations takes place, a block can be located simultaneously in both the data and the instruction local caches. As a result, a directory lookup always requires to look up both the data and the instruction directory.

The CMP model we are using along this thesis has a local instruction and data cache, and an inclusive shared cache. In this CMP, coherence is maintained by a coherence directory implemented as a duplicate of the local cache tags. Therefore, we can distinguish a data and a instruction directory (see Section 2.1).

Stores and evictions from the shared cache should invalidate all the copies of the target block in the local caches. In order to do that, the data and the instruction directory should be looked up since the system cannot assure that the target block has only been accessed as instruction or as data.

Instruction fetches (loads) that miss in the local caches should invalidate all the copies of the target block in the local data (instruction) caches to maintain instruction/data exclusivity (see Section 2.1). To carry out the corresponding invalidations, the data (instruction) directory should be looked up.

Table 3.1 gathers together the actions performed in the directory by any memory operation: loads and instruction fetches that miss in the local caches, stores, and evictions from the shared cache.

memory operation	**data directory**	**instruction directory**
load-miss	update	lookup
ifetch-miss	lookup	update
store	lookup	lookup
eviction from the shared cache	lookup	lookup

Table 3.1: Actions performed in the data and instruction directories for any shared cache access or eviction from the shared cache.

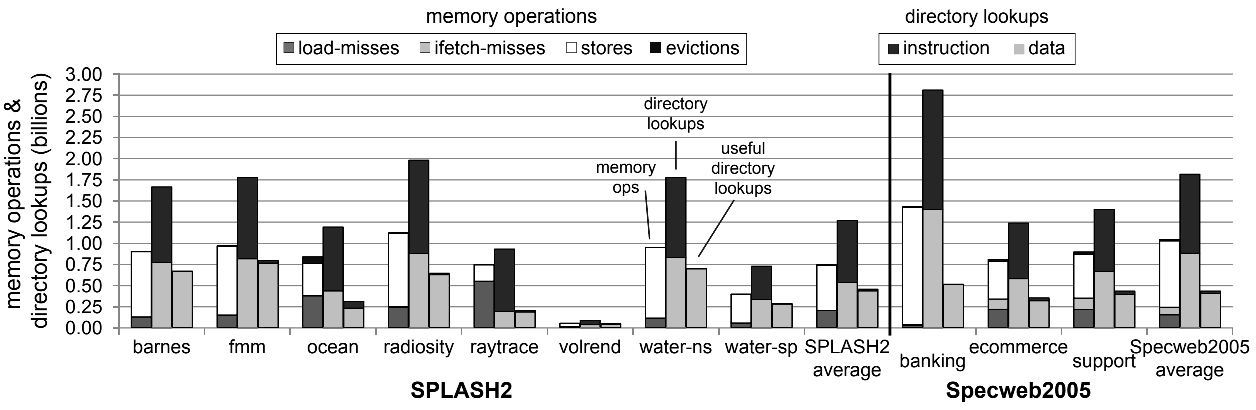


Figure 3.1: The first column for each benchmark represents the billions of memory operations that access the shared cache categorized as load-misses, ifetch-misses, stores and evictions. The second column collects the billions of directory lookups in each directory. The third column represents the billions of "useful" directory lookups in each directory

Any block cached in the shared cache has been usually accessed either as data or as instructions since, as we explain before, in general, the sets of memory addresses of instructions and data are disjoint. As a result, the local copies of a block cached in the shared cache are located either in the local data caches or in the local instruction caches. Therefore, a directory lookup performed in both the data and the instruction directory can only "hit" in one of them, that is, only one of the directories looked up reports that there are copies of the target block in the local caches that it represents. For example, an eviction from the shared cache performs a directory lookup in the data and the instruction directory. In general, if the directory lookup hits in the data directory, it misses in the instruction directory, and the other way around. In the same way, most of the times, instruction fetches (loads) that only require to lookup in the data (instruction) directory do not find a copy of the target block in that directory since a block accessed by an instruction fetch (load) generally has been previously accessed only by instruction fetches (loads or stores), and so, there are copies of that block only in the local instruction (data) caches.

We call "useless lookups" to the directory lookups that miss in a directory since there are no copies of the target block in any local cache represented by that directory. We call "useful lookups" lo the directory lookups that hit, that is, at least one local cache represented by the directory keeps a copy of the target block.

If we identify useless lookups in advance, we could prevent them to be performed. If their number is significantly important, the energy consumed by the directory could be reduced. Now, we analyze the number of useless lookups in the different benchmarks we use (see Section 2.3).

Figure 3.1 shows the distribution of memory operations that access the shared cache and the total and useful directory lookups in the modeled CMP. Figure 3.1 has three bars for each benchmark. The first bar shows the memory operations performed categorized as load-misses, ifetch-misses, stores and evictions (bottom-up). The second bar corresponds to the data and instruction directory lookups generated by the memory operations in the first bar. The last bar 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, on average, 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, and 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 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.[1]

Results from Figure 3.1 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.

3.2 Filtering mechanism

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, directory lookups are only performed in one of the directories (data or instruction), thus reducing the energy consumed. Figure 3.2 shows an scheme of the proposed filtering mechanism. We can see that the proposed filter is accessed for any directory lookup before accessing the directory itself. The filter determines if the target block is a data or an instruction block. In the former case, the directory lookup is performed only in the data directory. In the later case, the directory lookup is performed only in the instruction directory. For load-misses and ifetch-misses which only require to perform a data or an instruction directory lookup, respectively (see Section 2.1), the directory lookup can be completely avoided. For example, if the filter determines that the target block of a load-miss (ifetch-miss) is a data (instruction) block, no directory lookup is required.

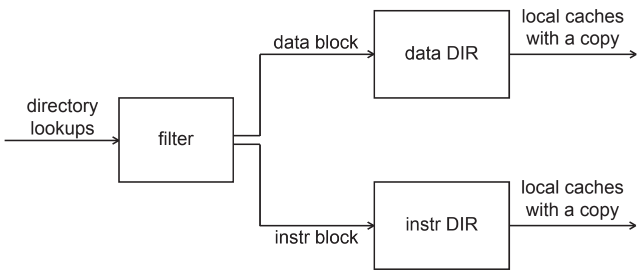


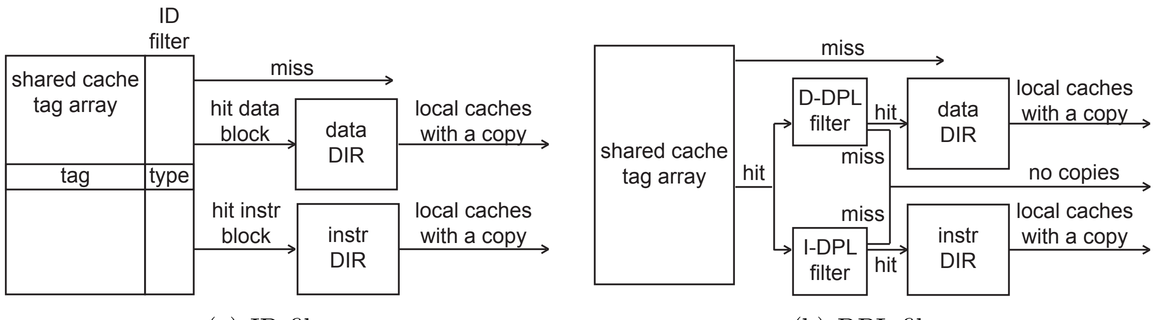
Figure 3.2: Filtering mechanism overview.

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

[1]Some directory lookups are performed to maintain instruction/data exclusivity. However, they represent a small fraction of the total number of lookups: 15% and 12%, on average, for SPLASH2 and Specweb2005, respectively.

block in the shared cache, similar to the state bits of the block. We call this kind of filters "instruction-data filter" (ID filter) (Figure 3.3(a)).

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 [7] are space-efficient probabilistic data structures used to test whether an element is a member of a set. We call this kind of filter "Decoupled filter" (DPL filter). We can distinguish a Data-DPL filter (D-DPL filter) for the data directory and a Instruction-DPL filter (I-DPL filter) for the instruction directory (Figure 3.3(b)).



(a) ID filter (b) DPL filter

Figure 3.3: Filter implementations.

Both proposed filters guarantee by their implementation that they do not produce false negatives, that is, they never indicate that a block is not located 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 covered with more detail in Sections 3.2.1 and 3.2.2. 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 located in the directory, the lookup is performed.

Both the ID-filter and the DPL-filter are read before the directory lookup is performed. Then, if necessary, the directory is accessed in parallel with the shared cache data array.

3.2.1 ID Filter implementation

In this section we introduce three different implementations of the ID filter: two-bit ID filter, one-bit ID filter, and one-bit improved ID filter. The difference among them is the information that is maintained for each shared cache block.

memory operation	block type	data directory	instruction directory
load-miss	data	update	—
	instr	update	lookup
	mixed	update	lookup
	no copies	update	—
ifetch-miss	data	lookup	update
	instr	—	update
	mixed	lookup	update
	no copies	—	update
store	data	lookup	—
	instr	—	lookup
	mixed	lookup	lookup
	no copies	—	—
eviction from the shared cache	data	lookup	—
	instr	—	lookup
	mixed	lookup	lookup
	no copies	—	—

Table 3.2: Actions performed in the data and instruction directories when using the two-bit ID filter for each memory operation.

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 classified as *data* blocks; blocks that have been accessed only by instruction fetches will be classified as *instruction* blocks; and blocks accessed by instruction fetches and stores or loads will be classified as *mixed* blocks. Therefore, the filter contains two bits per shared cache block.

Two-bits class identifier allows to classify blocks in four different categories. The basic performance only requires 3 types: *data*, *instruction*, and *mixed*. We add a new category called *no copies*. Local data caches are no write allocate (see Section 2.1), so blocks that have been accessed only by stores are not located in any local cache. These blocks will be classified as *no copies*.

The value of the filter 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 block classified as *no copies* changes its type to *data* when it is accessed by a load-miss and to *instruction* when it is accessed by an ifetch-miss. A *data* (*instruction*) block changes its type when it is accessed by an ifetch-miss (load-miss). The type of the block changes to mixed in both cases.

Table 3.2 collects the actions performed in the directory depending on the memory operation and the type of the target block. Comparing Table 3.1 and Table 3.2, we observe the following differences: a) for load-misses and ifetch-misses, the lookup actions can be eliminated for data and instruction blocks, respectively; and b) 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 is looked up. For a *no copies* block, directory lookups are not required.

This filter implementation has two drawbacks: it requires two bits per shared cache

memory operation	block type	data directory	instruction directory
load-miss	data	update	—
	instr	—	lookup
ifetch-miss	data	lookup	update
	instr	—	update
store	data	lookup	—
	instr	—	lookup
eviction from the shared cache	data	lookup	—
	instr	—	lookup

Table 3.3: Actions performed in the data and instruction directories when using the one-bit ID filter.

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, cannot 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*. In the modeled CMP, 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.

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 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 at a granularity of the shared cache block size, instead of the exclusivity at a granularity of the local data cache block size that has the modeled CMP. 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 shared cache block either in the local data caches or in the local instruction caches, but never in both of them.

Table 3.3 shows the actions performed in the directory for each memory operation when using the one-bit ID filter. Comparing Table 3.2 and Table 3.3, we see that stores and evictions from the shared cache either access the data directory or the instruction directory, but they never look up both as it happens if a block is classified as *mixed* when using the two-bit ID filter.

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.

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 (instruction to data) needs to invalidate all its copies in the local data (instruction) caches. This happens in three different situations: a) a load-miss accesses an instruction block, b) an ifetch-miss accesses a data block, and c) a store accesses an instruction block. In these situations, a directory lookup is needed to carry out the invalidations. In the system without filter, a directory lookup was also required when those memory operations were performed, but the directory lookup associativity was smaller. In the system without filter, the target block size of the directory lookup is the local data cache block size while, when using the one-bit ID filter, the target block size is the shared cache block size which, in our CMP model (see Section 2.1), is twice and four times the local instruction and data cache block size, respectively.

We analyze several workloads behavior and type block changes are rare so we expect the number of comparisons performed in the directory every time a block changes its type to be much smaller than the number of comparisons avoided when using the one-bit ID filter. However, we will see in Section 3.3.1, that, for some benchmarks such as `radiosity` or `banking`, the number of directory lookups performed in the data directory is bigger when using the one-bit ID filter than in the system without filtering. This increase is due to the existence of a few amount of highly accessed blocks which continuously change their type. These blocks come from the compiler location of program constants in the code region. It can happen, for example, that a shared cache block of 64B keeps instructions in its first 32B and data in its last 32B. In the system without filter, there are copies of the first 32B in the local instruction caches while there are copies of the last 32B in the local data caches. Therefore, instruction fetches (loads) that access the first (last) 32B of the block hit in the local instruction (data) caches and do not need to access the shared cache. However, in the system with the one-bit ID filter, any ifetch-miss (load-miss) that accesses the first (last) 32B sets the block type to instruction and invalidates all the copies of the last (first) 32B in the local data (instruction) caches. As a result, some loads and instruction fetches do not find a copy of the block in the local caches, increasing the local cache miss rate. That means that more memory operations access the shared cache and so, there is an increase in the number of directory lookups (recall Table 3.1). Moreover, these extra directory lookups require more comparisons than in the system without filter (since the directory target block size is the shared cache block size) and the filter cannot filter them out because the type of the block indicates that they should be performed (load-misses over instruction blocks and ifetch-misses over data blocks). Below we propose a different ID filter implementation that minimizes the block type changes.

Reducing invalidations: the one-bit improved ID filter

We propose the one-bit improved ID filter in order to avoid blocks that simultaneously keep instructions and data to continuously change its type. This filter assures instruction/data exclusivity by preventing a block classified as *instruction* block to change its type. Thus, when a load-miss access 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

memory operation	block type	data directory	instruction directory
load-miss	data	update	—
	instr	—	—
ifetch-miss	data	lookup	update
	instr	—	update
store	data	lookup	—
	instr	—	lookup
eviction from the shared cache	data	lookup	—
	instr	—	lookup

Table 3.4: Actions performed in the data and instruction directories when using the one-bit improved ID filter for each memory operation.

load-misses 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 filtering.

Table 3.4 shows the actions performed in the directory for each memory operation when using the one-bit improved ID filter. Comparing Table 3.3 and Table 3.4, we see that, the only difference is that now a load-miss never requires a directory lookup. When the target block is classified as *data*, a directory lookup is not necessary like in the system without filter. When the target block is classified as *instruction*, a directory lookup is not necessary since the block filter state does not change, so it is not necessary to invalidate the copies in the local instruction caches.

The filtering information is updated like in the one-bit ID filter. The only difference is that a block only changes its type from *data* to *instruction* when an ifetch-miss access a *data* block. The filter state of any block remains the same in the rest of the cases.

3.2.2 DPL Filter implementation

In this section, we introduce the DPL filter implementation in which filtering information is kept in structures decoupled from the shared cache organization (Figure 3.3(b)). There is one DPL filter per each stream (data or instruction). Each of these filters keep information about all blocks belonging to the corresponding stream. In our CMP model, this means that we use a DPL filter for the data directory (D-DPL filter) and another one for the instruction directory (I-DPL filter). Each of these filters keeps track of the shared cache block addresses located in each directory. We use the lower bits of the shared cache index of a shared cache block to do membership tests or updates in the filters.

We implement each DPL filter with a bloom filter. A bloom filter [7] is a space-efficient probabilistic data structure that is used to test whether an element is a member of a group or not. A bloom filter consists of a bit array that is accessed by several hash functions. To add a block to the set represented by a bloom filter, all the positions determined by performing the hash functions over the address bits of the block are set to 1. A target block is not in the set when any of the positions determined by the hash functions (performed over the address bits of the target block) is 0. Otherwise, there is a certain probability that the target block is in the set. False positives are possible, but false negatives are not.

We use Counting Bloom filters [17] in order to be able to remove elements from the group. A Counting Bloom filter maintains for each location in the bit array a count of the number of times that the bit is set to 1. The Counting Bloom filter we use is a Segmented Bloom filter [22]. The counter array is decoupled from the bit vector and the hash functions are duplicated. When an element has to be inserted/deleted it is sent to the counter array while a membership test is performed in the bit vector.

Before performing a directory lookup the corresponding DPL 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. Thus, the directory lookup is performed.

Each DPL filter is updated every time the corresponding directory is updated: a) a new block is allocated in a local cache, b) a block is evicted from a local cache, and c) a block is invalidated in a local cache from the shared cache due to a store, an eviction from the shared cache, or a load-miss or ifetch-miss (this last case is to maintain instruction/data exclusivity). Below we explain how the filter is updated:

- A new block is allocated in a local cache when a load-miss or an ifetch-miss access the shared cache. This new block is inserted in the corresponding DPL using the address of the block performing the load-miss or the ifetch-miss.
- A block is evicted from a local cache when a local cache miss occurs and the victim block is a valid block. This block should be removed from the corresponding DPL filter. The shared cache is informed about local cache evictions by load-misses and ifetch-misses since the victim local cache line is sent together with the new block address that has to be allocated in order to update the directory. The block cannot be removed from the DPL filter knowing only the local cache line evicted. It is necessary to access the directory to get the shared cache index of that block.
- A directory lookup identifies which blocks in the local caches should be invalidated. Therefore, after performing a directory lookup, the corresponding DPL filter should be updated removing the block that should be invalidated in the local caches.

We design DPL filters with bloom filters of different sizes and using different number of hash functions in order to analyze their effectiveness and to determine the size and number of hash functions of the bloom filters that we will use to implement the DPL filters. We choose two types of hash functions: based on bit shifts and based on XOR (exclusive or) operation [61]. 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*).

Figure 3.4 shows the percentage of directory lookups identified as useful for SPLASH2 by the Data-DPL (D-DPL in Figure 3.4(a)) and the Instruction-DPL (I-DPL in Figure 3.4(b))

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

Table 3.5: Different combination of hash functions used to analyze the effectiveness of the DPL filters (both the D-DPL and the I-DPL filters).

implemented using different bloom filters. We also include a perfect filter which really shows how many useful directory lookups are both for the data and the instruction directory. In Figure 3.4, the number of entries of the bloom filters used vary from 64 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 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 4.8 shows which hash functions are used when we indicate 1, 2, 3, or 4 in Figure 3.4.

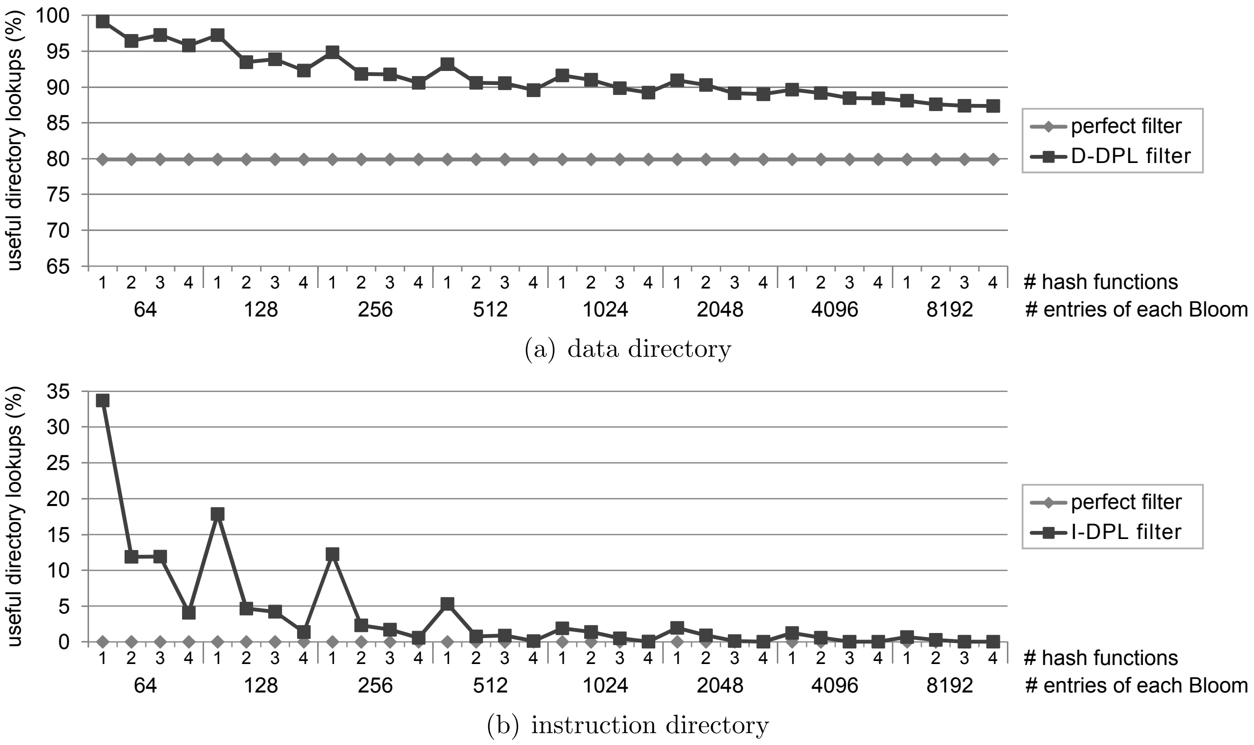


Figure 3.4: Percentage of useful directory lookups in the data and instruction directory in the system without filtering respect to the total number of directory lookups performed and percentage of directory lookups identified as useful by the DPL filters.

Figure 3.4(a) shows that the D-DPL filter is able to identify very few useless directory lookups, even using bloom filters with a big number of entries. Figure 3.4(b) shows that the effectiveness of the I-DPL filter is good using bloom filters with a quite small number of entries and enough number of hash functions.

Finally, we decide to use a bloom filter with 128 entries and 2 hash functions to implement the DPL 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 DPL filter has a size of 320B.

3.3 Evaluation

This section shows the benefits of the proposed filters. Section 3.3.1 shows the number of directory lookups performed when using the proposed filters, that is, the filter coverage. Section 3.3.2 shows the number of comparisons avoided in the directory due to the different filters. Section 3.3.3 analyzes how many operations are performed in the filter to reduce the number of directory lookups performed and to keep the filtering information up-to-date. Section 3.3.4 and 3.3.5 analyzes how many invalidations messages are sent and the performance loss, respectively, when using filters that change the coherence protocol (one-bit and one-bit improved ID filters). Section 3.3.6 shows the energy saving when using each of the proposed filters. Finally, Section 3.3.7 shows the same results as Section 3.3.6 but for different local and shared cache sizes and different technologies.

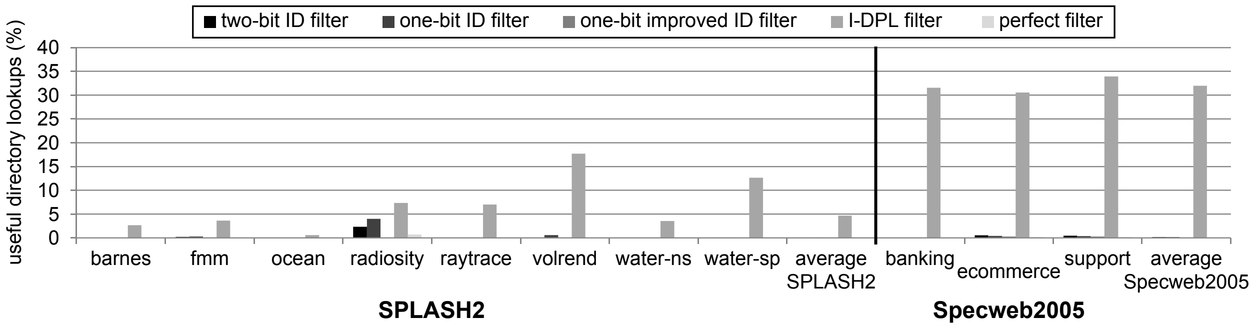
3.3.1 Coverage

In Figure 3.1, we showed that a big amount of directory lookups are useless. Now, we analyze whether the proposed filters are able to identify the useless lookups in advance or not.

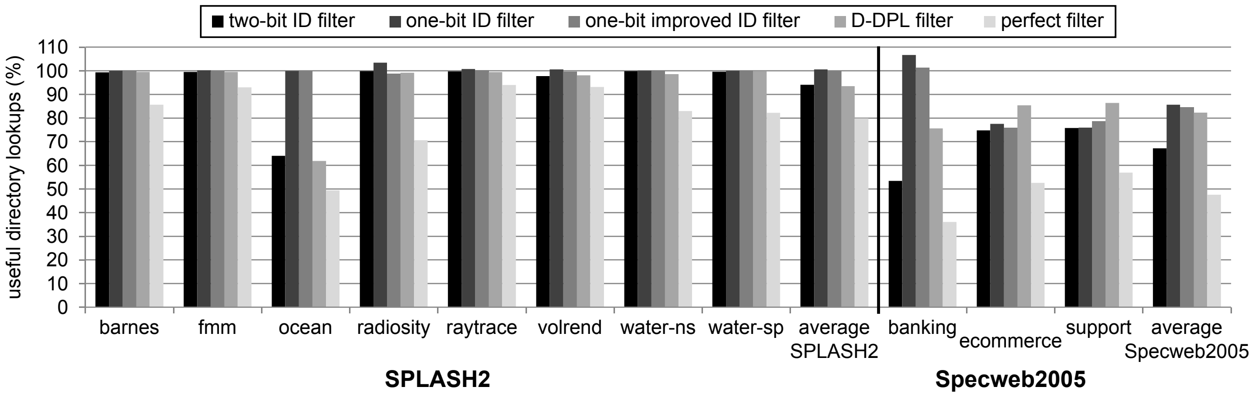
Figure 3.5 compares the number of directory lookups identified as useful by each proposed filter with the number of directory lookups identified as useful by a perfect filter. A perfect filter is a filter that is able to exactly identify useful directory lookups, that is, it exactly knows how many directory lookups performed in the system without filtering are useful. Figure 3.5(a) shows the percentage of useful instruction directory lookups and Figure 3.5(b) shows the percentage of useful data directory lookups. Both in Figure 3.5(a) and 3.5(b), for each benchmark, there are five bars. From left to right, the first three correspond to the ID filters: two-bit ID filter, one-bit ID filter, and one-bit improved ID filter. The fourth bar corresponds to the DPL filter: D-DPL for the data directory and I-DPL for the instruction directory. Finally, the last bar corresponds to the perfect filter.

For both SPLASH2 and Specweb2005, Figure 3.5(a) shows that, on average, the number of useful instruction directory lookups is below 1% (perfect filter). Any of the ID filters identifies almost all these cases. Thus, when using any of these filters, the instruction directory lookups performed are reduced, on average, to less than 1% of the instruction directory lookups performed in the system without filters. The I-DPL filter does not achieve such good results. For SPLASH2, the I-DPL filter reduces the instruction directory lookups to 4%, on average, but for Specweb2005, it only reduces the instruction directory lookups to 31%, on average.

The I-DPL filter has a bad performance for Specweb2005 because the bloom filter used to implement the filter is tuned for SPLASH2. We do not dedicate any effort to tune the I-DPL filter for Specweb2005 because later we show that, from a power consumption point of view, the performance of the I-DPL filter for SPLASH2 is not better than the performance



(a) instruction directory



(b) data directory

Figure 3.5: Percentage of directory lookups identified as useful by each proposed filter and a perfect filter. (a) corresponds to the instruction directory lookups and (b) corresponds to the data directory. In both graphs, for every benchmark, each bar correspond to the system using a different filter: two-bit ID filter, one-bit improved ID filter, DPL (D-DPL or I-DPL depending on the directory), and the perfect filter (from left to right).

of any of the ID filters. Moreover, to tune the I-DPL filter for Specweb2005 requires to use more memory address bits in the hash functions. To get these bits the tag array should be read and this will increase the energy consumption.

Comparing Figures 3.5(a) and 3.5(b), we see 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. We can see this in Table 3.6 which gathers together, for each benchmark, the number of memory operations performed in the shared cache and the percentage that corresponds to each memory operation type (load-misses, ifetch-misses, stores, and evictions from the shared cache). Table 3.6 shows that, on average, 67% and 68% of the memory operations performed are stores for SPLASH2 and Specweb2005, respectively. Most of the directory lookups performed in both directories are performed by stores. As local data caches are write-through (see Section 2.1), an important fraction of the stores 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. Data directory lookups performed by stores, on the other hand, are useful and should not be filtered out since directory lookups are performed also to identify which way in the local data cache has to be updated by a store.

benchmark	memory operations	load-miss	ifetch-miss	store	eviction from the shared cache
barnes	894.61	13.67	0.02	86.12	0.17
fmm	959.79	14.85	0.16	84.60	0.37
ocean	754.82	41.79	0.06	47.58	10.56
radiosity	1114.08	20.93	1.18	77.87	0.004
raytrace	739.42	73.72	0.06	26.20	0.0003
volrend	50.74	21.04	0.44	78.50	0.001
water-ns	943.61	11.70	0.006	88.24	0.04
water-sp	391.14	13.56	0.007	86.42	0.001
banking	1422.16	1.57	0.79	97.46	0.17
ecommerce	780.38	25.52	15.36	55.88	3.21
support	865.79	22.61	15.44	58.79	3.14

Table 3.6: Millions of memory operations performed in the shared cache and its distribution in ifetch-misses, stores, evictions from the shared cache, and load-misses.

Figure 3.5(b) 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 identified using any of the proposed filters is less than 1% of the data directory lookups in the system without filtering. The behavior of `ocean` is different: 50% of the data directory lookups are useless and both the two-bit ID filter and the D-DPL 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 evictions from the shared cache 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, Figure 3.5(b) 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 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 DPL filter requires a specific filter for each directory, that is, the D-DPL filter to filter out data directory lookups and the I-DPL filter to filter out instruction directory lookups. We propose to not use the D-DPL filter due to its poor performance and to only filter out the instruction directory lookups using the I-DPL filter. From now on, we will analyze the behavior of the whole DPL filter that uses both the I-DPL and the D-DPL and the behavior only for the I-DPL that will only filter out the instruction directory lookups while the data directory lookups will exactly remain the same as in the system without filtering.

3.3.2 Comparisons avoided in the directory

A directory lookup in the modeled CMP requires a different number of comparisons depending on the memory operation that performs it (see Section 2.1.1). This is due to the differences in the block size of the local caches and the shared cache. For example, a store requires 32 comparisons in the data directory (8 local data caches x 4-way local data cache) and 64 comparisons in the instruction directory (8 local instruction caches x 8-way local instruction cache). However, an eviction requires 128 comparisons in each directory (8 local data caches x 4 local data cache sets to invalidate x 4-way local data cache; 8 local instruction caches x 2 local instruction cache sets to invalidate x 8-way local instruction cache).

Figure 3.6 shows the number of comparisons performed by directory lookups in a system without filter and when using the different proposed filters. For each benchmark there are six bars. From left to right, the first bar represents the number of comparisons performed in the directory in a system without filtering. The next three bars show the comparisons performed when using the ID filters: two-bit ID filter, one-bit ID filter, and one-bit improved ID filter. The last two bars correspond to the DPL filter and the I-DPL filter.

We can see, in Figure 3.6, that the two-bit and the one-bit improved ID filter outperform the rest of the filters since they are the ones in which less comparisons are performed in the directories. The number of comparisons performed for both the two-bit and the one-bit improved ID filters is reduced, on average, to 25% of the comparisons performed in the system without filtering. `Banking` is the only benchmark in which there is a clear difference between the number of comparisons performed by the two-bit and the one-bit improved ID filters. In Banking, the two-bit ID filter reduces more the number of comparisons than the one-bit improved ID filter. This benchmark is the only one that though all memory operations are stores, the two-bit ID filter is able to filter data directory lookups. These stores are performed over blocks classified as *no copies*.

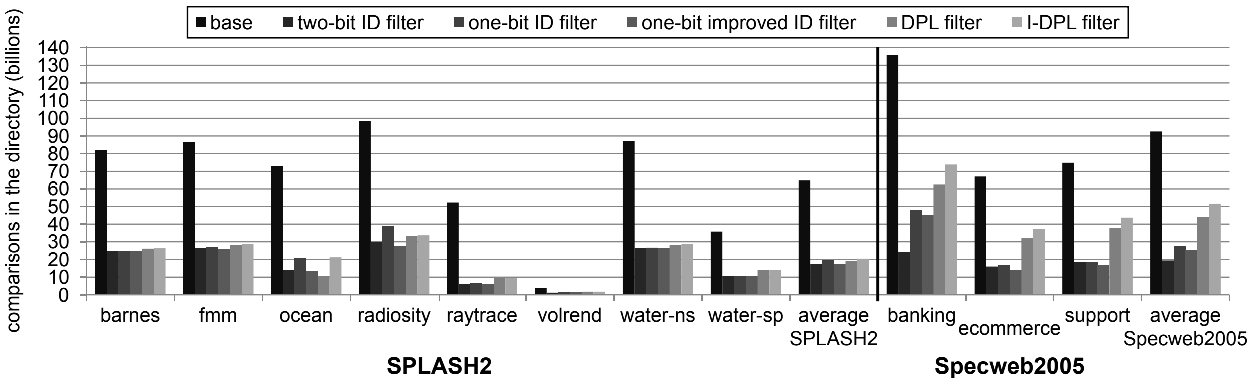


Figure 3.6: Billions of comparisons performed by directory lookups in both directories. For each benchmark, we show the number of comparisons in the system without filtering (base) and in the different proposed filters: two-bit ID filter, one-bit ID filter, one-bit improved ID filter, DPL filter, and I-DPL filter (from left to right).

Figure 3.6 also shows that, as we expect, the difference between the DPL filter and the I-DPL filter is very small. For SPLASH2, the DPL filter and the I-DPL filter reduce the number of comparisons, on average, to 29% and 32%, respectively. For Specweb2005, the

DPL filter and the I-DPL filter reduce the number of comparisons, on average, to 48% and 56%, respectively. The DPL filter consists of the I-DPL and the D-DPL filter. In the previous section, we show that the D-DPL filter filters out a very small amount of data directory lookups, so the number of comparisons avoided due to the D-DPL filter is also small. When using the DPL filter, the number of comparisons avoided are mainly comparisons performed in the instruction directory by instruction directory lookups filtered out by the I-DPL filter.

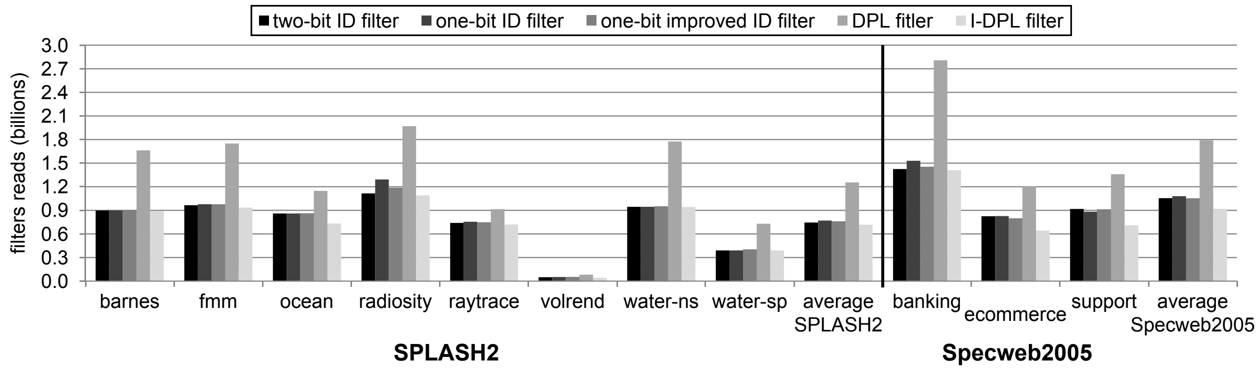
3.3.3 Operations performed by the filter

Figure 3.7 shows the number of reads and writes performed in the filter structure. Filter reads 3.7(a) are performed before any directory lookup to decide whether the lookup is useful or not. Filter writes 3.7(b) are necessary to keep the filtering information up-to-date. For the two-bit, one-bit and one-bit improved ID filters, these operations are performed over the filter bits allocated in the shared cache tag array structure. For the DPL filter and the I-DPL filter, these operations are performed over the filter structure that is decoupled from the shared cache organization.

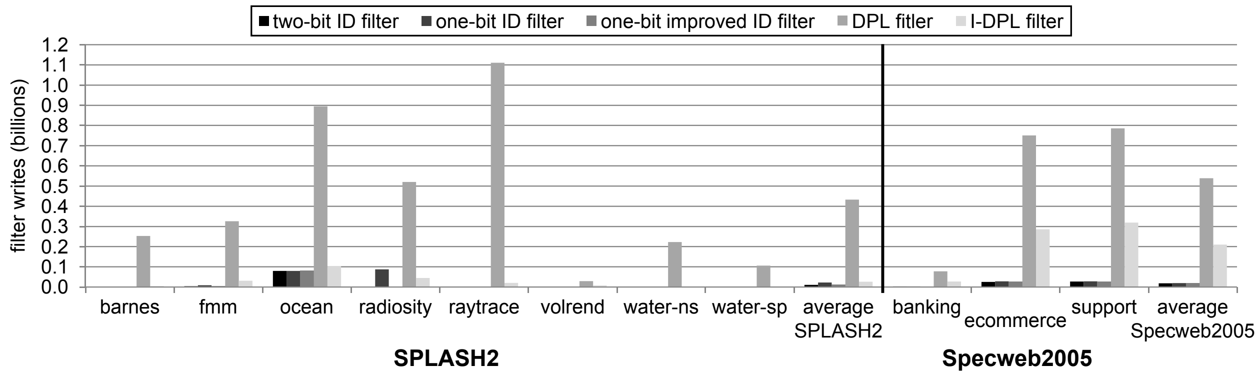
Both Figure 3.7(a) and 3.7(b) show five bars for each benchmark. From left to right, the first three bars correspond to the ID filters: two-bit, one-bit, and one-bit improved ID filters. The two last bars corresponds to the DPL filter and the I-DPL filter.

We can see, in Figure 3.7(a) that the DPL filter requires more filter operations than the rest of the proposed filters since it is the only proposed filter that requires to access two different structures: the D-DPL filter and the I-DPL filter. For example, a store, when using any of the ID filters, requires to read the filter bits attached to the shared cache block tag to get the block type of the target block. Based on the block type, a directory lookup in each directory is performed or avoided. In the same way, when using the I-DPL filter, one membership test is performed over the I-DPL filter structure and it determines whether the instruction directory is performed or not. A data directory lookup is always performed. However, when using the DPL filter, two membership tests are performed: one over the D-DPL filter and another one over the I-DPL filter. Based on the membership test performed over the D-DPL (I-DPL) filter, a data (instruction) directory lookup is performed or not. For the DPL filter, the only memory operations that only require to perform one membership test either over the D-DPL or over the I-DPL are load-misses and ifetch-misses. However, these operations represent a small percentage of the total number of memory operations (26% for SPLASH2, and 17% for Specweb2005).

Figure 3.7(b) shows that the DPL filter is also the proposed filter that requires more filter writes, that is, it is the filter that most often has to update its structure. ID filters update the filter bits only when a new block is allocated in the shared cache and when a block changes its type. Both situations are rare since the shared cache miss rate is low and, in general, data and instruction streams are disjoint. As a result, ID filters rarely require updates. However, the DPL filter requires to update its bloom filter any time a block is allocated, evicted or invalidated from the shared cache in any local cache. One block is allocated an another one is evicted from a local cache when a load-miss or an ifetch-miss access the shared cache. Though the number of load-misses and ifetch-misses represents a small fraction of the memory operations, it is far higher than the number of times a block



(a) filter reads



(b) filter writes

Figure 3.7: Billions of operations performed in the filter structure (reads and writes) along the execution of each benchmark to filter directory lookups out (reads) and to keep the filter structure up-to-date (writes).

changes its type in an ID filter.

The I-DPL filter suffers from the same problem as the DPL filter since it has to be updated due to changes in the local caches, but it only needs to be updated with block changes in the local instruction caches. Thus, it requires to be updated any time an ifetch-miss access the shared cache or an invalidation from the shared cache is sent to a local instruction cache. The number of ifetch-misses and invalidations sent to local instruction caches is much smaller than the number of load-misses and invalidations sent to local data caches. As a result, the number of updates of the I-DPL filter is not as high as the number of updates of the DPL filter. For `ecommerce` and `support`, the number of ifetch-misses is bigger than for the rest of the benchmarks (Table 3.6). Therefore, for these benchmarks the I-DPL filter require more updates than for the rest of the benchmarks.

3.3.4 Invalidation messages

The two-bit ID filter, the DPL filter, and the I-DPL filter do not modify at all the coherence mechanism. Thus, the number of invalidation messages sent to the local caches is the same as in the system without filters. The one-bit and the one-bit improved ID filters modify the coherence protocol since the instruction/data exclusivity is maintained at shared cache block size granularity. When using these filters, load-misses and ifetch-misses can invalidate several consecutive blocks in the local caches. All invalidations required by a load-miss or an ifetch-miss can be encapsulated in one single invalidation message (like for an eviction from the shared cache). However, depending on the number of processors that have to invalidate a copy in their local caches, more channels of the crossbar will be used and the power consumption will be higher. As several consecutive blocks in the local cache are invalidated, it is likely that more processors get involved.

Figure 3.8 shows the percentage of invalidation messages sent to the local caches when using the one-bit and the one-bit improved ID filter with respect to the system without filters. The number of invalidation messages is computed as the total invalidation messages multiplied by the number of processors involved in each of them.

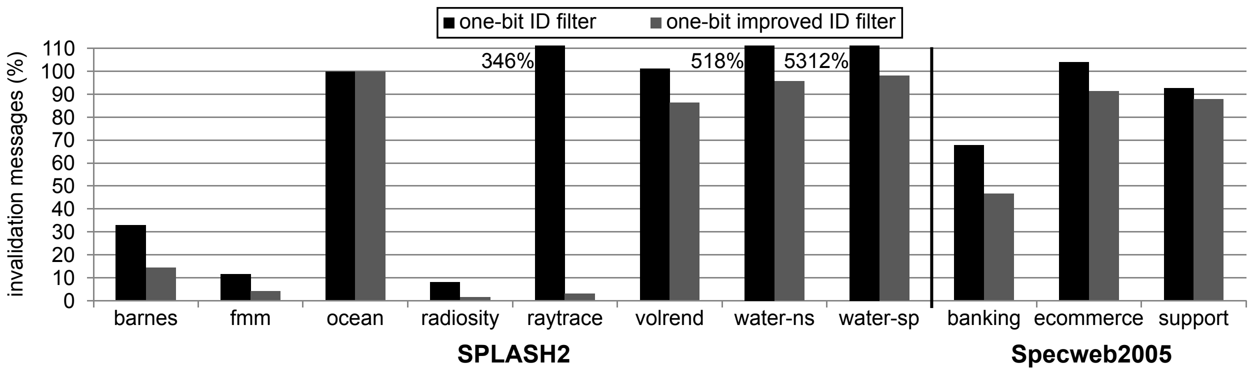


Figure 3.8: Percentage of invalidations messages sent (invalidation messages multiplied by the number of processors involved in each of them) for the one-bit and the one-bit improved ID filters with respect to the system without any filter.

When the one-bit ID filter is used, the number of invalidation messages increases for half of the benchmarks, reaching, for some of them, a huge value. Such an increase is due to the fact that several shared cache blocks highly accessed and shared by almost all processors contain simultaneously data and instruction. In the system without filters, as the data/instruction exclusivity was kept at local data cache block size, a part of these shared cache blocks was located in the data local caches and the other part in the instruction local caches and none of the memory operations performed requires to invalidate those copies.

The one-bit improved ID filter requires less invalidation messages than the one-bit ID filter since a shared cache block shared by instruction and data does not change its type continuously: it is classified as an *instruction* block and it is not allocated in the local data caches. Thus, the number of load-misses increases between 1% and 3% for the benchmarks that require a huge amount of invalidation messages when using the one-bit ID filter.

Figure 3.8 shows that when using the one-bit improved ID filter all benchmarks reduce the number of invalidation messages. As the data/instruction granularity is maintained at shared cache block size granularity, several consecutive local cache blocks can be invalidated together due to a load-miss or an ifetch-miss (all the invalidations are encapsulated in one invalidation message). Thus, accesses to the local cache blocks next to the block accessed by the load-miss or the ifetch-miss do not require any invalidation as invalidations have been already performed. As a result, invalidation messages are reduced.

3.3.5 Performance

The two-bit ID filter, the DPL filter, and the I-DPL filter do not modify the coherence protocol so benchmarks' performance is not altered. On the contrary, the one-bit and one-bit improved ID filters modify the coherence protocol forcing the instruction/data exclusivity at a shared cache block size granularity. It is then necessary to check that the performance of the benchmarks is the same as before.

Figure 3.9 shows the normalized execution time of the one bit and the one-bit improved ID filters with regard to the system without filtering. For Specweb2005, it is interesting to compare both the mean and the standard deviation of the different simulations that we have run (see Section 2.3.2) for the system with and without the ID filters. Because of this, we show three bars: the first one represents the mean and standard deviation for the system without filtering, and the other two correspond to the mean and the standard deviation in the system using the one-bit and the one-bit improved ID filter, respectively.

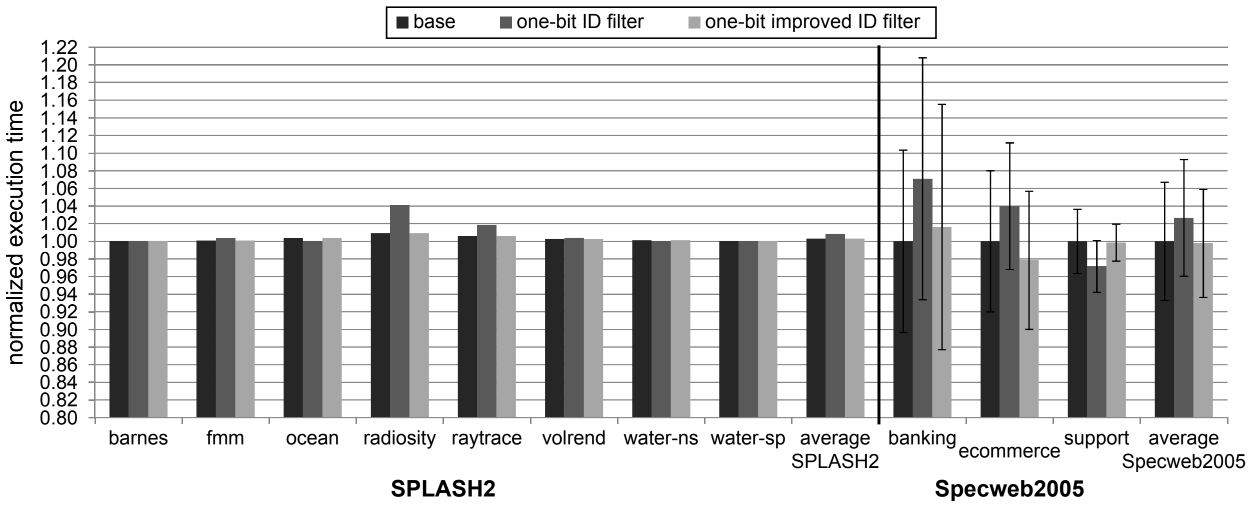


Figure 3.9: Normalized execution time of the one-bit and the one-bit improved ID filters with regard to the system without filtering.

For SPLASH2, Figure 3.9 shows that, on average, the system is 0.8% and 0.3% slower when using the one-bit and the one-bit improved ID filter, respectively. `Radiosity` is the benchmark with the worst performance. It has a performance loss of 4% and 0.9% for one-bit and one-bit improved ID filter, respectively. In this benchmark, the compiler locates frequently accessed constants in the code region. Therefore, 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 shared cache block size 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.

Figure 3.9 shows that in Specweb2005 we can differentiate two groups: for `banking` and `ecommerce` the mean execution time is bigger when using the one-bit ID filter than when using the one-bit improved ID filter; for `support` the mean execution time is bigger when using the one-bit improved ID filter. However, in both groups the confidence interval shows that the execution time is not statistically different.

For `banking`, the variability is higher because we cannot 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 (see Table 2.5).

3.3.6 Power reduction

We use CACTI [43] to estimate the dynamic energy and leakage power for the cache tag array, the directory, and the proposed filters (see Section 2.2 for more details).

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.

Figure 3.10 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 the data and the instruction directories. There are five bars for each benchmark. From left to right, the first three bars correspond to the ID filters: two-bit ID filter, one-bit ID filter, and one-bit improved ID filter. The last two bars correspond to the DPL filter and the I-DPL filter.

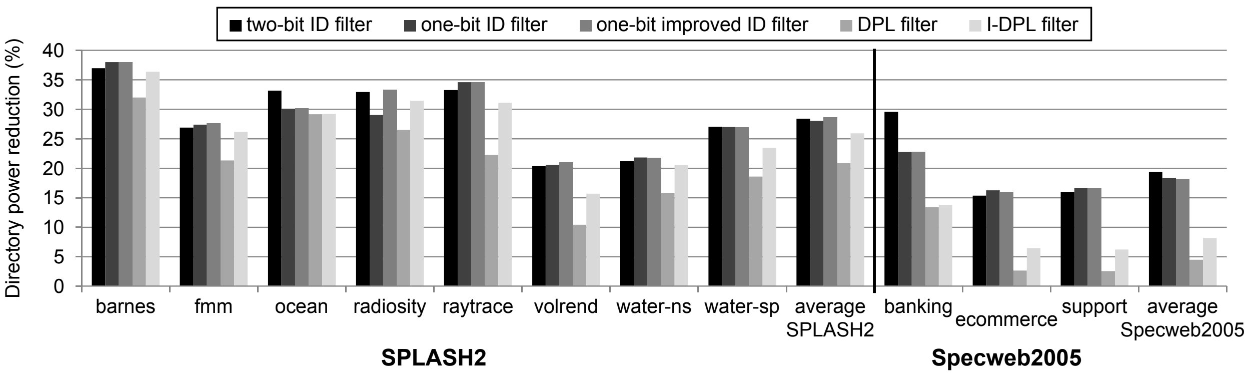


Figure 3.10: Percentage of power reduction in the directory when using the different proposed filters. For each benchmark, we show the power reduction for the two-bit ID filter, the one-bit ID filter, the one-bit improved ID filter, the DPL filter, and the I-DPL filter (from left to right).

The DPL filter shows, as we expect, the lower directory power reduction. The reason of the poor performance in power terms of the DPL filter is that it does not reduce the number of comparisons in the directory more than the rest of the proposed filters, but it requires to perform more filter operations for its correct operation. For the rest of the discussion in this section we do not take into account the DPL-filter.

For SPLASH2, on average, the power reduction is quite similar when using any of the proposed filters. The two-bit , the one-bit and the one-bit improved ID filters, on average, reduce the power consumption by 28%, 29%, and 29.5%, respectively. The I-DPL filter shows a slightly poorer performance in power terms. It reduces the directory power by 27%.

In contrast, for Specweb2005, there is an important difference between the reduction got by the ID filters and the I-DPL filter. Figure 3.5 shows that the I-DPL filter identifies less useless directory lookups than the other proposed filters. Thus, the directory power is only reduced by 9% when using the I-DPL filter. On the other hand, any of the ID filters, on average, reduce the directory power by 19%.

There are important differences on average power reduction between SPLASH2 (28%) and Specweb2005 (19%), though Figure 3.5 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 there is a small performance loss (Figure 3.9), but this performance loss is smaller than when using the one-bit ID filter.

3.3.7 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.

A DPL filter size is determined by three parameters of the bloom filter it uses: 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 mapped in the directory the DPL filter is associated to. Thus, an increase in the number of blocks mapped to the directory will increase the counter bits. The 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 represented by the bloom filter. As a result, the number of false positives of a DPL filter with a specific bloom filter size increases. It is then necessary to check that the accuracy loss in the DPL filter does not affect the results.

In Section 3.2.2, we decide to use an I-DPL filter implemented with a bloom filter of 128 entries and 2 hash functions. For SPLASH2, this I-DPL filter reduces the number of comparisons performed to 4.7% with respect to the system without filtering. When the size of the local caches is doubled, the number of comparisons is reduced to 7%, and when both the size of the local caches and shared cache are doubled, the number of comparisons is only reduced to 8.7%. We can see that the effectiveness of the I-DPL filter is significantly reduced. As a result, we decide to analyze the behavior of two different I-DPL filters when the size of the caches is modified: an I-DPL filter implemented with a bloom filter of 128 entries and 2 hash functions, and an I-DPL filter implemented with a bloom filter of 256 entries and 2 hash functions. An I-DPL filter implemented with a bloom filter of 256 entries and 2 hash functions reduces the number of comparisons to 2.3%, 2.7%, 3%, and 3.7% in our CMP model, when the size of the shared cache is doubled, when the size of the local caches is doubled, and when both the size of the local and shared caches is doubled, respectively.

For Specweb2005, an I-DPL filter implemented with a bloom filter of 128 entries and 2 hash functions reduces the number of comparisons performed to 32%, but it only reduces the number of comparisons to 53.4% when both the size of the local and shared caches is doubled. An I-DPL filter implemented with a bloom filter of 256 entries and 2 hash functions reduces the number of comparisons performed to 13.8%, 14.4%, 24.3%, and 26.1% in our CMP, when the size of the shared cache is doubled, when the size of the local caches is doubled, and when both the size of the local and shared caches is doubled, respectively.

Figure 3.11 shows the average percentage of power reduction for SPLASH2 and for Specweb2005 for different cache sizes for each proposed filter. There are five groups of bars for each benchmark suite. For both benchmark suites, the two-bit ID filter is used for the results in the first group, the one-bit ID filter is used in the second group, the one-bit improved ID filter is used in the third group, and the fourth and fifth group correspond to the results for an I-DPL filter with 128 entries and 2 hash functions and an I-DPL filter with 256 entries and 2 hash functions, respectively. There are 4 bars in each group. The first one shows the average numbers presented in Figure 3.10 that are for the memory hierarchy parameters defined in Table 2.1. For the next bars, the size of the local and shared caches is increased, but the rest of their parameters remain the same as before. In the second bar, only the shared cache size is doubled, in the third bar only the size of the local caches is 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. In Figure 3.11, we can see that our workloads are barely affected by this effect since the number of evictions with both shared cache sizes is small.

When the shared cache size is increased, the power consumption of the ID filters increases. We expect the percentage of power reduction to decrease. Figure 3.11 shows that the percentage of power reduction is only reduced for the two-bit ID filter. The one-bit and one-bit improved ID filters barely increase the leakage and power consumption of the tag array, so an increase in the shared cache size does not affect the percentage of directory power reduction.

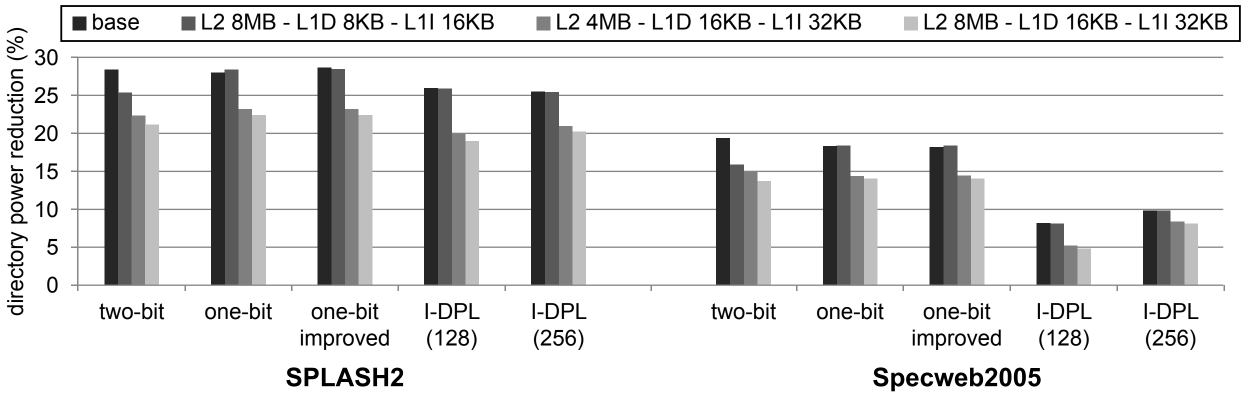


Figure 3.11: Percentage of power reduction in the directory for the different proposed filters when we vary the size of the caches.

Figure 3.11 also shows that the percentage of power reduction when using the I-DPL filters is not affected when the shared cache is doubled as it was expected.

For SPLASH2, we can see in Figure 3.11 that the I-DPL filter with a bloom filter of 128 entries is slightly better than the I-DPL filter with a bloom filter of 256 in our CMP model and also when the shared cache size is doubled. The effectiveness of the I-DPL filter with a bloom filter of 256 entries is better than the I-DPL filter with a bloom filter of 128 entries, but the energy consumed by the former is higher than the energy consumed by the latter. However, when the size of the local caches is doubled, the I-DPL filter with a bloom filter of 256 entries outperforms the other I-DPL filter. Therefore, it is important to perform effectiveness analysis when the memory hierarchy parameters change.

For Specweb2005, the I-DPL filter with a bloom filter of 256 entries always outperforms the I-DPL filter with a bloom filter of 128 entries since the effectiveness of the latter is not good enough. We explained before that the I-DPL filter was only tuned for SPLASH2 since in our evaluation we showed that, from a power consumption point of view, the performance of the I-DPL filter for SPLASH2 is not better than the performance of any of the ID filters.

When the local caches size is doubled, Figure 3.11 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. Figure 3.12 compares the percentage of power reduction when using 65nm and 22nm technologies. We use the memory hierarchy parameters described in Table 2.1 and an I-DPL filter that uses a bloom filter with 128 entries and 2 hash functions.

Figure 3.12 shows average numbers for SPLASH2 and Specweb2005. There are four groups of bars for each benchmark suite. For both benchmark suites, the two-bit ID filter is used for the results in the first group, the one-bit ID filter is used in the second group, the one-bit improved ID filter is used in the third group, and the fourth group corresponds to the results for the I-DPL filter with a bloom filter of 128 entries and 2 hash functions. There are 2 bars in each group. The first one shows numbers for 65nm technology with a target frequency of 1.2GHz and the second bar shows numbers for 22nm technology with a target frequency of 2.75GHz.

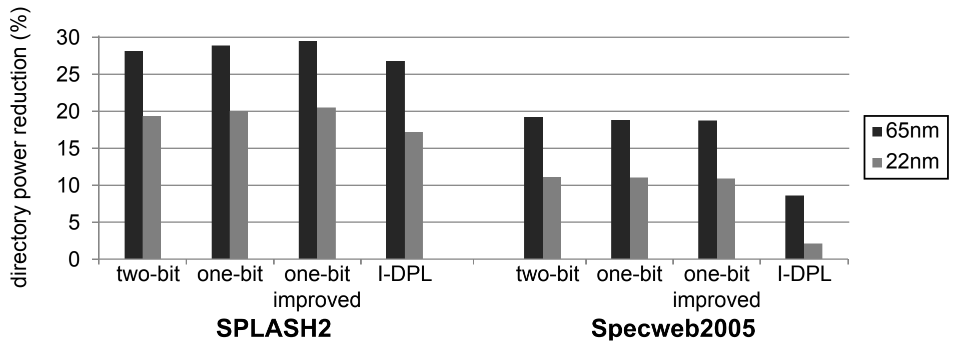


Figure 3.12: Percentage of power reduction in the directory modeling all the structures with a 65nm technology or a 22nm technology.

Figure 3.12 shows that the power reduction when using the different filters is smaller for a 22nm technology than for a 65nm technology. In SPLASH2, the power reduction attained is still quite interesting for all the proposed filters. The I-DPL 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 I-DPL filter does not get such good results. It only reduces the power by 2%.

3.4 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). The filters based on this implementation are called ID filters. In the second one, we keep the information of all blocks belonging to a stream together using a separate filter structure based on a bloom filter for each directory (data and instruction). The filters based on this second implementation are called DPL filters.

We propose several different filters based on the two implementations described. For the first implementation, we introduce three filters: the two-bit ID filter, the one-bit ID filter, and the one-bit improved ID filter. For the second implementation, we analyze two DPL filters: the DPL filter, and the I-DPL filter. Using any of these filters, on average, more than

60% of the directory lookups are avoided for SPLASH2. For Specweb2005, the ID filters reduce the directory lookups by more than 60%, but the DPL filters only reduces them by 45%. The ID filters achieve ∼28% and ∼19% reduction in directory power consumption for SPLASH2 and Specweb2005, respectively. The DPL filter achieves the lower directory power reduction of all the filters since it requires to perform more filter operations than the rest of them. It reduces the directory power consumption by 20% and 5% for SPLASH2 and Specweb2005, respectively. Finally, when using the I-DPL filter, the power consumption is reduced by 26% and 8% for SPLASH2 and Specweb2005, respectively.

The results shown in this chapter lead to the conclusion that ID filters perform better than DPL filters. The DPL filter requires more filter operations for its correct operation, so its energy consumption is higher than the rest of the proposed filters. The I-DPL has a similar energy consumption by construction to the ID filters, but ID filters are able to avoid more directory lookups. A good advantage of the I-DPL filter over ID filters is that its 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.

Chapter 4

Reducing Directory Lookup Associativity

This chapter describes a group of filters which reduce the associativity of the directory lookups performed in a CMP with write-through local caches. These filters are intended for a directory implemented as a duplicate of the local cache tags that require a high associative directory lookups. An important fraction of the target blocks of directory lookups are shared by a small number of processors so they are only located in a small subset of local caches or even in one specific local cache. Based on this, the proposed filters limit the associativity of any directory lookup.

We propose two different filter implementations: the Owner filter and the Array of Bloom Filters filter (ABF filter). The Owner filter keeps explicit information for every block in the shared cache identifying either the owner of the block (processor that keeps in its local cache the only local copy of the block) or the group of processors that share the block. The ABF filter keeps a superset of locations where the local copies of any target block can be located.

Our results show that, on average, the Owner filter reduces the number of comparisons performed by 97% and 93% for SPLASH2 and Specweb2005, respectively. As a result, the directory power is reduced, on average, by 31% and 22% for SPLASH2 and Specweb2005, respectively. On the other hand, the ABF filter does not get such good results since it has a good coverage but the energy consumed by the filter structure is too high compared to the energy consumed by the directory.

This chapter is organized as follows. Section 4.1 motivates why the proposed filtering mechanism works. Section 4.2 presents the filtering mechanism. Section 4.3 describes the Owner filter and evaluates its behavior. Section 4.4 details the different ABF filter proposals and evaluates them. Finally, Section 4.5 concludes this chapter.

4.1 Introduction

In a directory-based protocol, any store that access the shared cache requires a directory lookup in order to invalidate the copies of the target block in the local caches. Depending on the write policy of the local caches, the number of stores that access the shared cache varies. 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. Thus, the number of stores performed in the shared cache is bigger in a CMP with write-through local caches than in a CMP with write-back local caches. However, write-through local caches offer several advantages against write-back local caches.

The coherence protocol in a CMP with write-through local caches is simplified for several reasons. First of all, data is always up-to-date in the shared cache or the shared memory. Thus, any write request always access the shared cache or memory in the same way. This guarantees that any access to shared data is served in a specific amount of time. It does not depend on the number of local copies in the system, or the state of these copies. Moreover, it is easier to build bigger systems based on clustering several chips with write-through local caches as any request from a foreign chip can be solved directly from the shared cache. It would be only necessary to send the corresponding invalidations to the local caches, but writebacks from local caches are not necessary. In the same way, evictions from the shared cache in an inclusive system do not require to wait for any writeback from the local caches. They only need to send the corresponding invalidations to invalidate the local copies of the target block.

In a CMP with write-back local caches, the number of stores that access the shared cache or memory are smaller since only local data cache misses or ownership requests access the shared cache or memory. Any request performed by a specific processor to a block that only contains data private to that processor is solved locally. Only the first access to that block and the first store performed over it access the shared cache. After that, only if the block is evicted or invalidated, a new access to the shared cache would be necessary. However, an ownership request could either be served from the shared cache or require to access data kept in the local cache of other processor. For example, a store performed over a block that has been modified in advance by another processor. The value of the block has to be read from that local cache, perhaps be written in the shared cache or memory, and then served to the processor currently accessing it. As a result, a processor request timing depends on the state of the target block in the system. This complicates the coherence problem since more block states have to be taken into account and more race conditions can take place. Moreover, CMPs with write-back local caches can be seen its benefits reduced when the programs present false sharing.

The CMP model described in Section 2.1 has write-through local caches. In our workloads (Section 2.3) we found that only 1 out of 100,000 stores access true shared data. The rest of the stores are accessing data that is local to the processor performing the store. In a CMP with write-back local caches, most of these stores would be performed locally (as long as the processor has a copy in exclusive state). On the contrary, in a CMP with write-through local caches and a directory based coherence protocol, every store access the shared cache and performs a directory lookup. Most of the times, the only copy of the target block of the directory lookup performed by a store is located in the local caches of the processor performing the store. Therefore, most of the directory lookups performed by stores are useless and it is possible to improve directory energy-efficiency by filtering them out.

Figure 4.1 presents a detailed analysis of directory lookups performed by stores and evictions: from left to right, the first bar shows the number of directory lookups due to stores and evictions (`directory lookups`), the second bar shows the number of these directory lookups that find at least one copy of the cache block in any local cache (`at least one copy`), and the third bar shows the number of directory lookups due to stores that only find a copy of the cache block and this copy is located in the local cache of the processor performing the store (`only itself`).

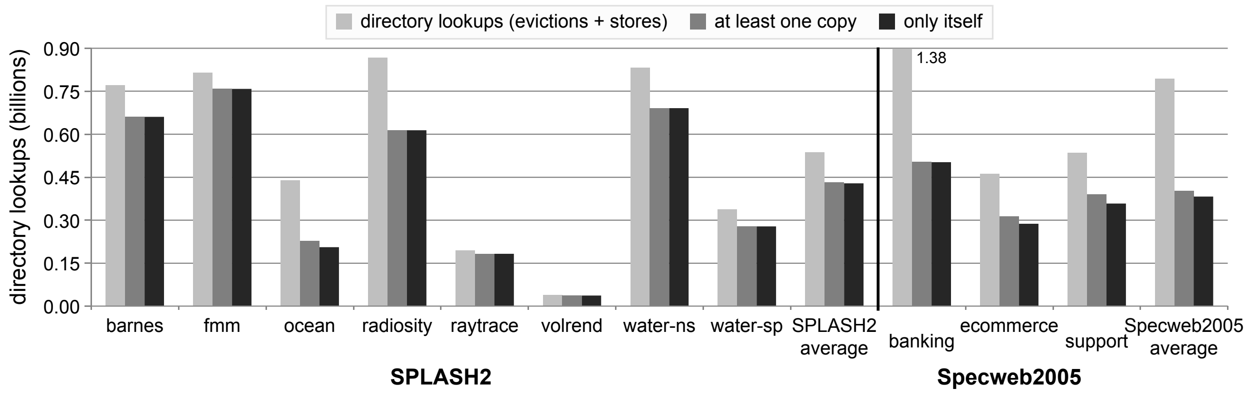


Figure 4.1: Number of directory lookups due to stores and evictions (`directory lookups`), number of directory lookups due to stores and evictions that find at least a copy of the cache block in any local cache (`at least one copy`), and number of directory lookups due to stores that only find a copy just in the local cache of the processor that performs the current store (`only itself`).

The difference between the first two bars in Figure 4.1 indicates the number of times that there are no copies of the cache block in any local cache. On average, this difference represents 20% and 50% of the directory lookups performed by SPLASH2 and Specweb2005, respectively. The difference between the second and the third bar represents all cases that require to invalidate local copies of the target block. These cases are: a) evictions performed over shared cache blocks which have local copies, and b) stores performed over shared cache blocks that are located at least in a local cache different from the local cache of the processor performing the store. On average, this difference represents less than 1% of the directory lookups for both SPLASH2 and Specweb2005. The remaining directory lookups (80% and 49% for SPLASH2 and Specweb2005, respectively) are performed by private stores, i. e. stores that access cache blocks without copies in any other processor's local cache.

We propose a filtering mechanism that filters out stores over private data, so that they do not perform expensive and useless directory lookups. Thus, the proposed mechanism reduces the associativity of the directory lookups shown in the third bar in Figure 4.1. 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 4.1, these directory lookups are represented by the difference between the first two bars.

This filtering mechanism is limited since it only reduces energy when specific situations take place: an owner exists, or there are no copies of the target block in any local cache. We propose a second filtering mechanism which can reduce the directory lookup associativity to the real number of local copies of the target block under any circumstance. This filter is based on an array of bloom filters.

4.2 Filtering mechanism

Based on the results in the previous section, we propose two filtering mechanisms to reduce the directory lookup associativity. Using the proposed filters, directory lookups require less comparisons than in the system without filtering and so, the energy consumed by the directory is reduced.

The first mechanism we will describe in this chapter is focused in filtering out directory lookups performed by stores over private data. We call it "Owner" filter. Figure 4.1 shows that, on average, 80% and 49% of the directory lookups in SPLASH2 and Specweb2005, respectively, are performed by stores that access blocks in the shared cache that are only replicated in the local cache of the processor performing those stores. These stores do not require to send invalidations to the rest of the processors in the system. However, it is necessary to inform the local cache of the local way that has to be updated. The lookup in the duplicate tag directory could be restricted to the copy of the tag array of the processor performing the store if we know beforehand that this processor has in its local cache the only copy of the block in the system.

Figure 4.2 shows how the Owner filter works. The filter is accessed before performing a directory lookup. The filter logic determines which part of the directory must be accessed based on the information read. If the filter identifies the owner of the target block, only the copy of the tag array of the local cache of the owner processor must be looked up. As a result, the associativity of the corresponding directory lookup is reduced.

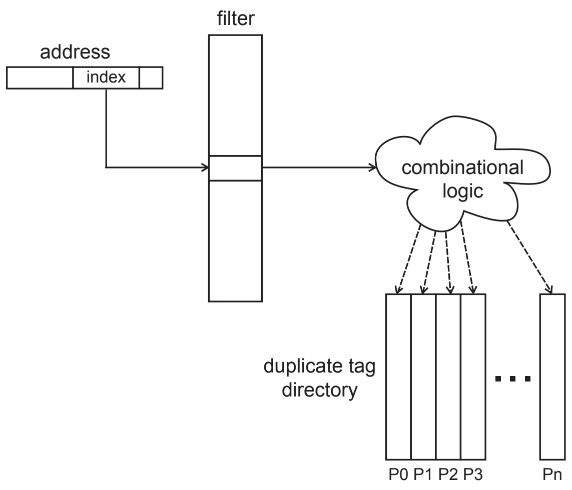


Figure 4.2: Scheme of the Owner filter

The Owner filter has as many entries as cache blocks in the shared cache, and each entry has $\log_2 P$ bits plus a valid bit, being P the number of processors in the system. As it is described until now, this filter is only useful when the target block has an owner (there is only one local copy). This is a common situation, but it does not happen always. For example, Figure 4.1 shows that 20% and 50% of the directory lookups for SPLASH2 and Specweb2005, respectively, are performed over blocks that have no local copies. Moreover, we will see later in Section 4.3 that the Owner filter keeps imprecise information since it does not always identifies the owner of a block even if it exists. To improve the performance of the filter, we propose to change the owner identifier bits (bits that identify the owner when the valid bit is set) when the valid bit is unset to a coarse granularity [25, 34]. Consequently, it is possible to identify other situations and get benefit from the filter even when the owner is unknown by the filter, or when more than one local cache keeps copies of the block (there is no owner).

The performance of the Owner filter is limited since it only reduces energy when specific situations take place: an owner exists, or there are no copies of the target block in any local cache. We propose a second filtering mechanism that can identify where the copies of a target block are located, and so, the filter reduces directory lookup associativity under any circumstances. The location of the local copies is maintained by means of an Array of Bloom Filters. We call this filter "ABF" filter.

The ABF filter is accessed before performing a directory lookup and it reduces the directory lookup associativity. Figure 4.3 shows an example of this type of filter in a system with 4 processors. In this case, the filter consists of a bloom filter per processor, that is, each bloom filter represents the blocks located in the local tag array cache of a specific processor. For every directory lookup, all the bloom filters are accessed and only when the result is positive, the corresponding part of the directory is looked up.

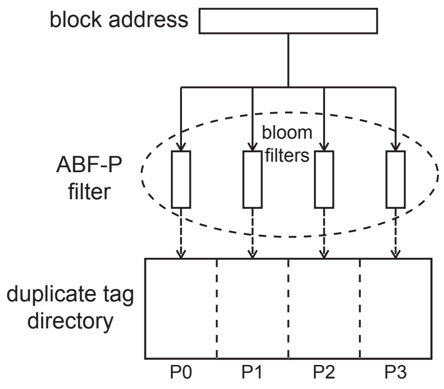


Figure 4.3: Scheme of an ABF filter that consists of a Bloom Filter per processor

Different ABF filter designs can be implemented depending on the part of the directory represented by each bloom filter. Section 4.4 describes the different designs and the benefits and drawbacks of each proposal.

4.3 Owner Filter implementation

In this section we describe the Owner filter implementation: how the number of directory lookups is reduced, which modifications are necessary in the coherence protocol, and the operations performed in the filter to maintain it up-to-date. Finally, we evaluate this filter.

Each line in the shared cache has associated one entry in the filter. For every memory operation, 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.

First, we describe the filter structure assuming that the local cache and the shared cache block sizes are the same. Then, we describe the specific characteristics of the filter for a CMP with different local cache and shared cache block sizes.

4.3.1 Owner Filter states

The CMP model described in Section 2.1 has 8 cores, so 3 bits are necessary to code the owner identifier. Table 4.1 shows how the owner identifier and the valid bit are used to encode information useful to avoid directory lookups or reduce the number of entries looked up in a directory lookup. Table 4.1 also includes the filter state name, which is used from now on to refer to each combination of the owner identifier and the valid bit values.

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)

Table 4.1: Filter states

When the valid bit is set, the owner identifier bits codify the identifier of the processor that keeps in its local data cache the only local copy of the target block. A directory lookup is restricted to the directory entries that correspond to the local data cache of the owner. When the valid bit is unset, the filter does not identify the owner either because it does not exist or because the filter update mechanism cannot identify it. Table 4.1 shows that, in this situation, we identify 7 different filter states and there is still one empty codification.

The first codification in Table 4.1 when the valid bit is unset, which is called *no copies*, identifies when there are no local copies of the target block. In this situation, a directory lookup is not performed independently of the memory operation performed in the shared cache.

The rest of the codifications identify, among other things, whether the local copies are located only in the local data caches or only in the local instruction caches. In the former case, any directory lookup will be only performed in the data directory while, in the latter case, any directory lookup will be only performed in the instruction directory.

Other thing that the filter tries to identify when it does not identify the owner is the group of processors that keep a local copy of the target block. We distinguish two groups of processors using the most significant bit of the identifier of a processor: *subgroup0* consists of the processors identified as 0, 1, 2, and 3, while *subgroup1* consists of the processors identified as 4, 5, 6, and 7. When the filter identifies that all the local copies are located in the local caches of the processors that belong to that subgroup, only the directory entries that correspond to the local caches of the processors in the identified subgroup are looked up when a directory lookup is performed.

Table 4.1 shows that for a *data* block we distinguish three possible states depending on whether the filter identifies a subgroup (*data block (subgroup0)* and *data block (subgroup1)*) or not (*data block (all)*). For an *instruction* block we also distinguish three different filter states.

In the proposed coding of the filter bits, we do not consider classifying a cache block as both data and instruction. This situation might happen, for instance, when compilers include read only data in instruction memory regions. However, since the baseline CMP model chosen already forces instruction/data exclusivity, we also force it in the proposed filter. In order to reduce directory lookups and invalidation messages, we prevent that a block classified as *instruction block* changes its type. When a load accesses a block classified as *instruction block*, the data is supplied to the local data cache, but the block is not allocated in to the local data cache. Hence, the number of load-misses can increase, but neither directory lookups nor invalidations are necessary.

4.3.2 Owner Filter update

A filter entry state is updated using only the following information: the memory operation type, the identifier of the processor performing the memory operation, and the previous filter state. We also know the evictions from local caches. Using them the filter information will be precise, but extra and costly directory lookups will be required. Consequently, we decide to use not precise filter information, that is, to only know a superset of the copies in the local caches.

The filter state *valid owner* is set on three cases: a) load-miss to a block in the shared cache without copies in the local caches (*no copies*), b) load-miss that also misses in the shared cache 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 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*, *instruction block*, or *data block (subgroupX)* where 'X' is not the subgroup the processor performing the store belongs to), and b) store that misses in the shared cache.

Both ifetch-misses and load-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 load-miss accesses a block whose filter state is *instruction block*, the filter state is not modified.

4.3.3 Owner Filter entry granularity

In the CMP modeled, the local cache and the shared cache block sizes are different. The shared cache block size is four times the local data cache block size and twice the local instruction cache block size. Keeping a filter entry per local cache block involves very high storage requirements, so we chose the size of the shared cache block as the filter granularity.

We do not want to modify the filter state diagram described in Section 4.3.2, so it is necessary to slightly modify the coherence protocol in order to: a) increase the number of invalidations a store can generate, and b) increase the granularity of the data/instruction exclusivity. Both modifications require that some memory operations perform more invalidations than before. All these invalidations only use one invalidation message per processor as it happens with evictions from the shared cache.

A store will invalidate not only the copies of the subblock accessed, but all the copies of the shared cache block that may exist in the local caches. As this may increase false sharing, we calculated how many processors, on average, share a subblock and also share the whole shared cache block. Our results show that more than 99.9% stores are performed to shared cache blocks which have copies of any of their subblocks in just one of the local caches. Less than 0.01% stores access a shared cache block that is shared by several processors while the specific subblock accessed by the store is not shared by the same amount of processors. Therefore, false sharing is not expected to increase.

4.3.4 Owner Filter operation

Any access or eviction performed in the shared cache reads the filter state (valid bit + owner bits) for the target block. This information is used to reduce the number of directory lookups required or to reduce the number of entries looked up by a directory lookup (directory lookup associativity).

The number of directory lookups is reduced when the filter state indicates that there are no copies of the target block in the data or instruction local cache, e.g., a *data block* (*instruction block*) that indicates that there are no copies in the instruction (data) local cache or a block labeled as *no copies* which has no local copies.

The directory lookup associativity is reduced when the filter state identifies either the owner (*valid owner*) or the subgroup of processors which can have a copy of the block in their local caches (*(subgroupX)*). In the former case, the associativity is divided by the number of processors. In the latter case, as processors are split into two subgroups, the associativity is

memory operation	valid bit	owner identifier	filter state	data directory	instruction directory
load-miss	1	xxx	valid owner		—
	0	0xx	data block or no copies		—
	0	101 or 110	instruction block (subgroupX)		—
	0	111	instruction block (all)		—
ifetch-miss	1	xxx	valid owner	(128/8)	
	0	000	no copies	—	
	0	001 or 010	data block (subgroupX)	(128/2)	
	0	011	data block (all)	(128)	
store	1	processor performing store	valid owner	(32/8)	—
	1	other processor	valid owner	(128/8)	—
	0	000	no copies	—	—
	0	001 or 010	data block (subgroupX)	(128/2)	—
	0	011	data block (all)	(4)	—
	0	101 or 110	instruction block (subgroupX)	—	(128/2)
	0	111	instruction block (all)	—	(128)
eviction	1	xxx	valid owner	(128/8)	—
	0	000	no copies	—	—
	0	001 or 010	data block (subgroupX)	(128/2)	—
	0	011	data block (all)	(128)	—
	0	101 or 110	instruction block (subgroupX)	—	(128/2)
	0	111	instruction block (all)	—	(128)

Table 4.2: Lookups performed in the directories for every memory operation that access the shared cache. The number of entries looked up is enclosed. When the number of entries is a fraction, it indicates that the number of directory lookups is not reduced, but the number of entries looked up in each lookup is reduced.

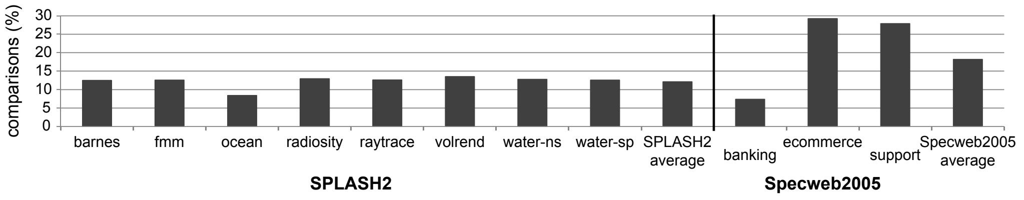
halved.

As the filter granularity is the shared cache block size, all the subblocks of the target shared cache block are looked up in every directory lookup. Consequently, all directory lookups involve 128 comparisons (either in the data or the instruction directory). Stores that access *valid owner* blocks are the only exception: as the directory lookup is performed to know the way of the subblock to update in the local data cache, the directory lookup is limited to 32 comparisons in the data directory.

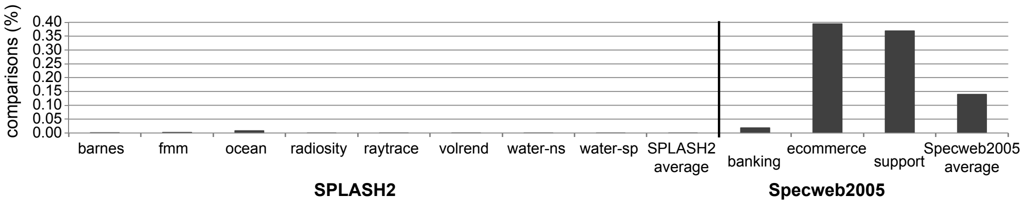
Table 4.2 summarizes the number of directory entries that are looked up for every memory operation that requires a lookup, depending on the filter state. Table entries with a dash indicate that a directory lookup is not required. For instance, comparing Table 2.2 and Table 4.2, we can conclude that a directory lookup performed by an eviction is eliminated when the filter state is *no copies.* Table entries labeled X/Y mean that the directory lookup involves X entries, but only a fraction of the entries equal to 1/Y are looked up. For example, an ifetch-miss to a *valid owner* block performs a directory lookup over 128/8 entries, that is, only the entries corresponding to the owner processor are looked up.

4.3.5 Owner Filter overhead

The filter proposed requires ($\texttt{1 + log}_2\texttt{P}$) number of bits per shared cache line, being P the number of cores in the CMP. For the CMP described in Section 2.1, as it has 8 cores, four extra bits per shared cache line are required. For each 512KB, 64B block-size shared cache bank, the filter implementation requires 4KB. This represents 12% of the shared cache bank tag array size (including the state bits in the tag array) and 0.7% of the total shared cache bank size (tag array + data array).



(a) data directory



(b) instruction directory

Figure 4.4: Percentage of directory comparisons performed by directory lookups when using the Owner filter compared to the system without filtering.

4.3.6 Owner Filter evaluation

In this section, we evaluate the coverage of the Owner filter, the number of filter operations that the filter requires for its correct operation, the number of messages sent through the crossbar when the filter is working, the performance of the system with the filter compared to the system without filtering, the reduction in the directory power consumption, and the directory power reduction when using other cache sizes and future technologies.

Coverage

The Owner filter tries to reduce the directory lookups associativity. Thus, we define the Owner filter coverage as the number of comparisons that are avoided with respect to the system without filtering.

Figure 4.4 shows the percentage of comparisons performed in the directory when using the proposed filter with respect to the comparisons performed in the system without filtering. Figure 4.4(a) shows the percentage of comparisons in the data directory and Figure 4.4(b) shows the percentage of comparisons in the instruction directory. Table 4.3 shows, for both directories, the number of comparisons performed in the system without filtering and the percentage due to each memory operation type.

Figure 4.4 shows that the number of comparisons performed by directory lookups is greatly reduced. The reduction is more important in the instruction directory than in the data directory: for all the benchmarks, more than 99% of the comparisons performed in the instruction directory are avoided while, on average, only 88% and 81% of the comparisons performed in the data directory are avoided for SPLASH2 and Specweb2005, respectively. Table 4.3 shows that the number of comparisons performed in the instruction directory is, in general, twice the number of comparisons performed in the data directory. As a result, the number of avoided comparisons in the instruction directory is much bigger than in the

benchmark	**data directory**				**instruction directory**			
	TOTAL (billions)	stores (%)	ifetch-miss (%)	eviction (%)	TOTAL (billions)	stores (%)	load-miss (%)	eviction (%)
barnes	24.87	99.13	0.05	0.81	57.34	85.99	13.65	0.35
fmm	26.54	97.87	0.37	1.75	61.56	84.42	14.82	0.75
ocean	21.73	52.88	0.13	46.98	53.38	43.05	37.81	19.12
radiosity	28.61	97.02	2.95	0.02	70.45	78.80	21.18	0.01
raytrace	6.23	99.51	0.48	0.00	47.29	26.22	73.77	0.00
volrend	1.28	98.86	1.12	0.01	3.23	78.85	21.13	0.00
water-ns	26.70	99.79	0.01	0.19	60.41	88.21	11.70	0.08
water-sp	10.82	99.97	0.02	0.00	25.03	86.43	13.56	0.00
banking	45.21	97.71	1.58	0.70	90.09	98.06	1.58	0.35
ecommerce	24.84	56.18	30.89	12.92	43.87	63.62	29.05	7.31
support	28.33	57.49	30.19	12.31	48.59	67.03	25.78	7.17

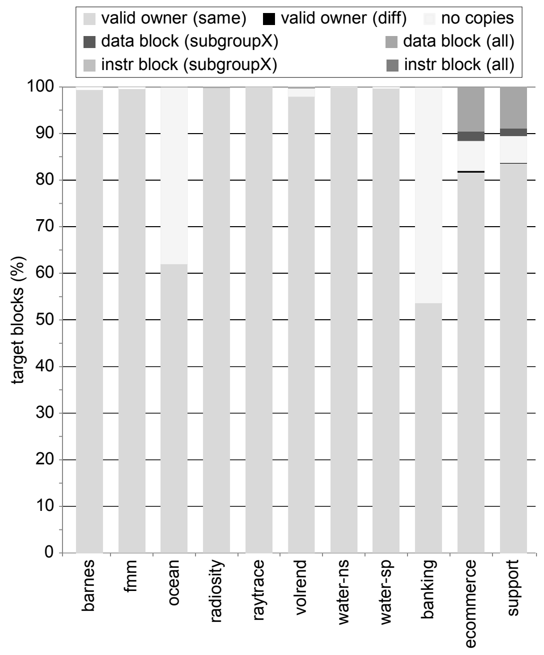
Table 4.3: Billions of data and instruction directory comparisons in a system without filtering performed by directory lookups and the percentage that corresponds to each memory operation.

data directory. This is due to the fact that the most common memory operation is a store (see Table 4.3) that requires 32 comparisons in the data directory and 64 comparisons in the instruction directory. Moreover, in general, the filter identifies the owner of the blocks accessed by stores so an instruction directory is completely avoided but a data directory lookup is required (though its associativity is smaller than the original). It is important to remember that even if the filter identifies the owner of a block that is accessed by a store and the owner is the processor performing the store, a data directory lookup is necessary to determine the local data cache way that has to be updated.

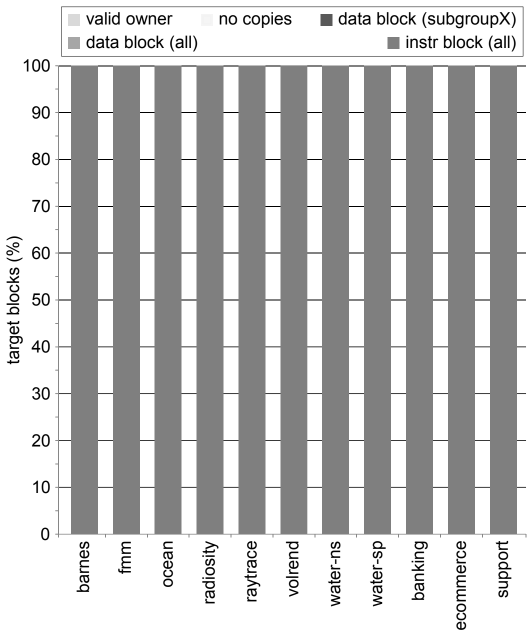
In general, the number of comparisons performed in the directories is reduced quite similarly for all benchmarks, except for `ecommerce` and `support`. The difference between these two benchmarks and the rest is not significant in the instruction directory since the number of comparisons is reduced to less than 0.4% for all the benchmarks. However, the difference is important in the data directory. The number of comparisons in the data directory is reduced to 29% and 28% for `ecommerce` and `support`, respectively, while it is reduced below 14% for the rest of the benchmarks.

Using Table 4.3, that shows how many comparisons in the directory are performed by each memory operation, and Figure 4.5, that shows, for each memory operation, the percentage of directory lookup target blocks that are in each filter state, we see that the difference between `ecommerce` and `support`, and the rest of the benchmarks is due to two reasons: a) blocks in `ecommerce` and `support` are shared for more processors than in the rest of the benchmarks, and b) more evictions from the shared cache are performed in `ecommerce` and `support` than in the rest of the benchmarks and an important fraction of the evictions performed (more than 25%) access blocks that are shared among several processors.

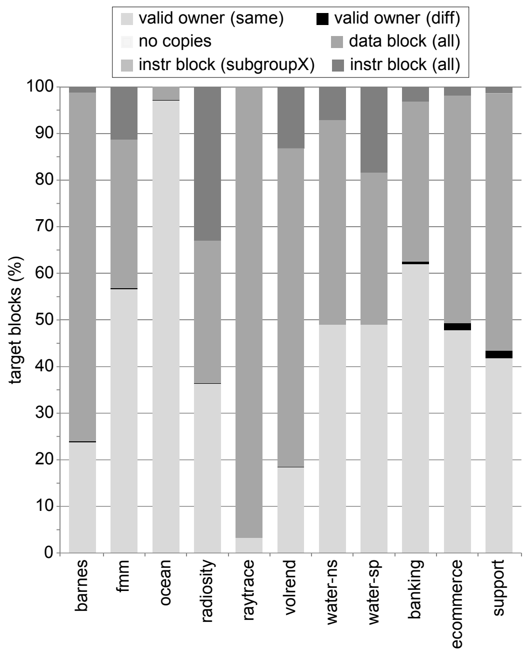
Table 4.3 shows that for every benchmark more than 50% of the comparisons performed in the directory are due to stores. Stores require a data directory lookup that performs 32 comparisons (8 processors x 4-way associative local data caches) and an instruction directory lookup that performs 64 comparisons (8 processors x 8-way associative local instruction cache). Figure 4.5(a) shows the percentage of target blocks accessed by stores that are in each filter state. We can see that `ecommerce` and `support` are the only benchmarks in which



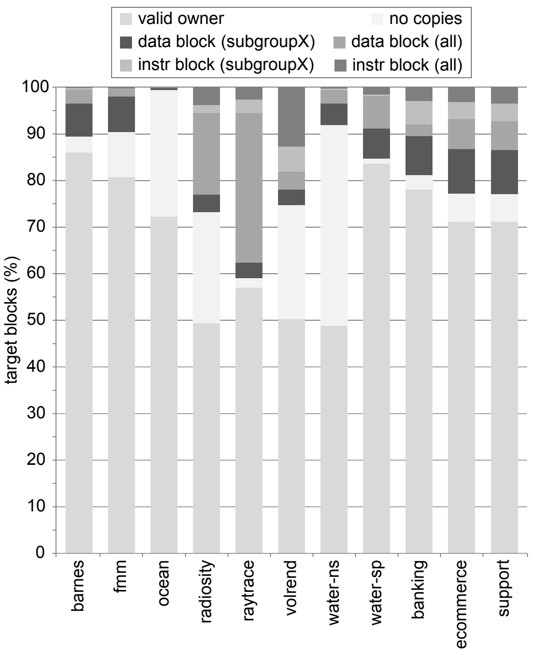
(a) store



(b) ifetch-miss



(c) load-miss



(d) eviction

Figure 4.5: Filter states

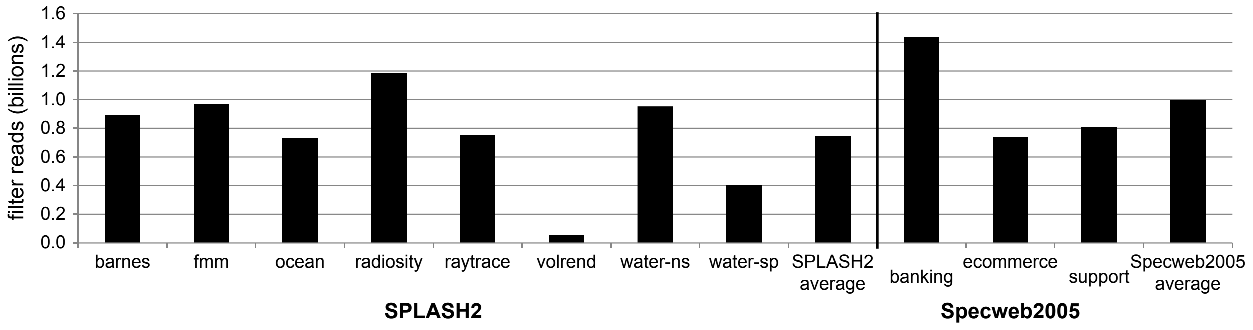
more than 10% of the blocks accessed by stores are identified as *data* blocks by the filter, that is, 10% of the stores have more than one local copy. For the rest of the benchmarks, the filter identifies that more than 99% of the stores are performed over blocks that have an owner and the owner is the processor performing the store (*valid owner*). As a result, `ecommerce` and `support` perform stores that require directory lookups with the original data directory lookup associativity (32 comparisons) while the rest of the benchmarks only require to perform a data directory lookup which associativity is the original associativity divided by P, being P the number of processors (4 comparisons). Therefore, for `ecommerce` and `support` the number of comparisons performed by stores is reduced less than for other benchmarks when using the Owner filter.

Figures 4.5(b) and 4.5(c) show that, for ifetch-misses and load-misses, all the benchmarks behave similarly since all of them identify target blocks as *instruction* (for ifetch-misses) or *data* (for load-misses) blocks. Figure 4.5(c) shows that target blocks are classified as *valid owner* (same or different), *no copies*, or *data block (all)*, but all of them have the same meaning: an instruction directory lookup is not required. Therefore, for ifetch-misses and load-misses, the filter reduces the number of comparisons performed by directory lookups in the same way for all the benchmarks.

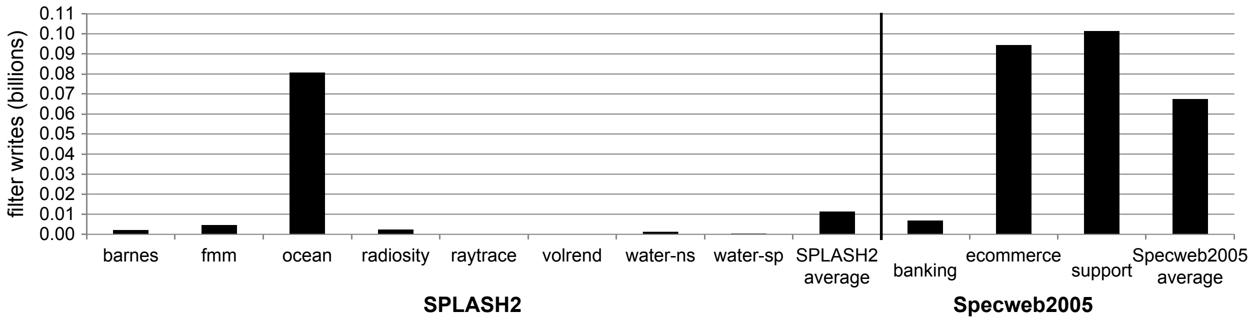
Finally, Figure 4.5(d) shows that for all the benchmarks an important fraction (from 10% to 40%) of evictions access blocks that are classified either as *data* or *instruction* blocks. In these cases, the number of comparisons performed by a directory lookup is only halved since only one directory is accessed. Table 4.3 shows that `ocean`, `ecommerce`, and `support` are the only benchmarks that perform a significant fraction of evictions, therefore, we can conclude that for evictions the filter does not work as nicely as for other memory operations, but, as it is not an important memory operation, in general, we do not notice it. For `ocean`, that performs an important number of evictions, Figure 4.5(d) shows that it is the only benchmark in which the fraction of *data* or *instruction* blocks is below 1% of the blocks accessed by evictions. `Ecommerce` and `support` perform more evictions than the rest of the benchmarks and so, they are affected by the low performance of the filter for evictions.

Operations performed by the filter

Figure 4.6 shows the number of reads (Figure 4.6(a)) and writes (Figure 4.6(b)) performed in the Owner filter structure. These operations are performed over the filter bits located in the shared cache tag array structure. Comparing Figure 4.6(a) and Figure 4.6(b) we can see that the number of reads is much bigger than the number of writes. Reads are performed for any memory operation, while writes are only performed in a shared cache miss (a new block is allocated in the shared cache) or when a block changes its state. In general, the number of filter writes is below 0.01 billions, except for ocean, ecommerce, and support which are the benchmarks with the highest shared cache miss rate. The number of filter reads is, on average, 0.7 billions for SPLASH2 and 1.0 billions for Specweb2005.



(a) filter reads



(b) filter writes

Figure 4.6: Billions of operations performed in the filter structure (reads and writes).

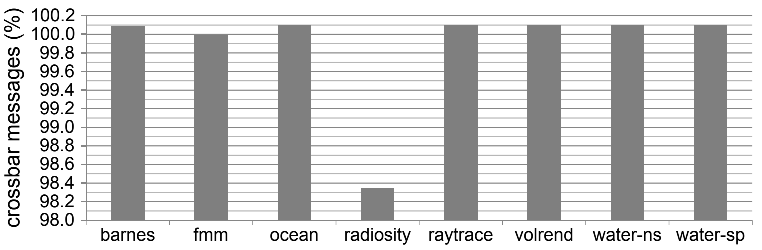
Crossbar messages

The Owner filter modifies the coherence mechanism since data/instruction exclusivity is maintained at the shared cache block size (64B) instead of keeping data/instruction exclusivity at the local data cache block size (16B). Moreover, a block classified by the filter as an *instruction* block never changes its state. Thus, a load-miss that access an *instruction* block never allocates that block in the local data cache of the processor performing the load. As a result, the number of invalidations messages performed by stores, load-misses, and ifetch-misses can be modified together with the number of load-misses and ifetch-misses performed in the shared cache.

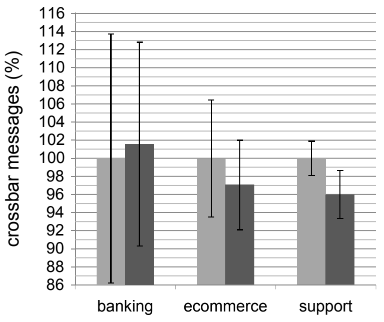
The number of invalidations messages performed by stores can be modified in two different directions since an invalidation involves all the subblocks (16B) of a shared cache block (64B): a) more subblocks are simultaneously invalidated so more caches can be involved and more messages are required, and b) just with one message several subblocks located in the same cache can be simultaneously invalidated so next stores to those subblocks do not require to send invalidations.

The number of invalidation messages performed by load-misses and ifetch-misses together with the number of load-misses and ifetch-misses is modified due to two different reasons: a) since an *instruction* block never modifies its filter state, the continuous change of type of several blocks that contain simultaneously data and instructions is eliminated, reducing the number of invalidation messages necessary and modifying the number of load-misses and ifetch-misses that access the shared cache, and b) to maintain data/instruction exclusivity at the shared cache block size the number of invalidation messages is modified like the number of invalidations messages performed by stores and this can increase the number of load-misses in the shared cache.

Figures 4.7(a) and 4.7(b) show the percentage of messages to and from the shared cache when using the Owner filter with respect to the system without filtering for SPLASH2 and Specweb2005, respectively. Figure 4.7(a) shows that, for SPLASH2, the number of messages is less than 0.1% bigger for all benchmarks, except for `radiosity`. `Radiosity` has several blocks that contain simultaneously data and instruction. In the system without filtering, when an ifetch-miss access the shared cache, the local data copies are invalidated and a local copy is located in the local instruction cache. Then, a subsequent load-miss invalidates the local copy in the instruction cache. As a result, there are a lot of invalidation messages performed by ifetch-misses and load-misses. When using the Owner filter, this situation is avoided since a block with such characteristics is only kept in the local instruction caches so invalidations are not required and the number of ifetch-misses that access that block are reduced.



(a) SPLASH2



(b) Specweb2005

Figure 4.7: Percentage of messages in the crossbar when using an owner filter with respect to the system without filtering.

Figure 4.7(b) shows the number of messages in the crossbar for Specweb2005. For all the workloads of Specweb2005 we run several simulations to minimize variability (see Section 2.3.2) and so, in Figure 4.7(b), we show the average percentage and the standard deviation with respect to the system without filtering for all the workloads. The number of messages is, on average, increased for `banking` and reduced for `ecommerce` and `support`. However, statistically the number of messages in the system with the Owner filter and the system without filtering is the same.

Performance

The Owner filter modifies the coherence protocol forcing the instruction/data exclusivity at the shared cache block size granularity. Thus, we need to check that the performance remains unchanged. Figure 4.8 shows the normalized execution time of the CMP with the proposed filter with respect to the baseline CMP. For Specweb2005, it is interesting to compare both the mean and the standard deviation of the different simulations that we run (see Section 2.3.2) for the system with and without the Owner filter. Due to this, for Specweb2005, we show two bars: the first one represents the mean and standard deviation for the system without filtering, and the other corresponds to the mean and the standard deviation in the system using the Owner filter. 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.

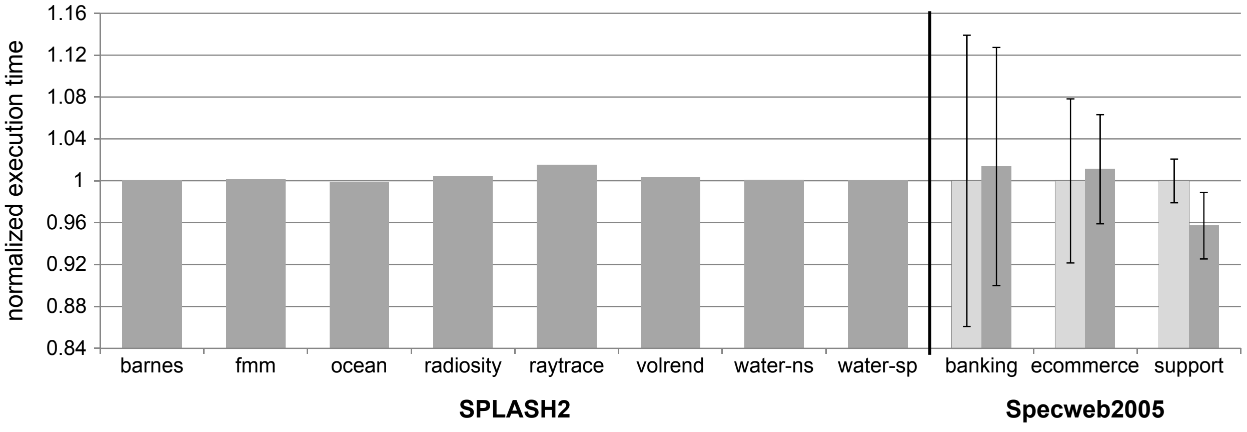


Figure 4.8: Normalized execution time

Power reduction

We use CACTI [43] to estimate the dynamic energy and leakage power for the cache tag array, and the directory (see Section 2.2 for more details). The Owner filter is embedded in the shared cache tag array since it is implemented by increasing the number of bits of each shared cache tag array entry. As a result, it is not necessary to model the filter structure (only to increase the size of the shared cache tag array).

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 4.9 shows the percentage of power reduction in the directory using the Owner filter. It takes into account the power consumption reduced in the directory as well as additional power 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 Owner 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.

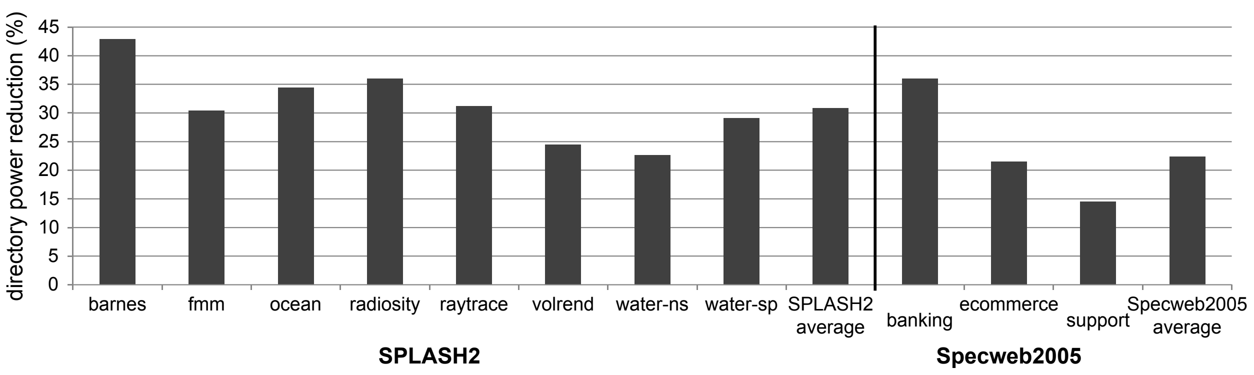


Figure 4.9: Percentage of power reduction in the directory.

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 sizes and new generation technologies

The size of the Owner 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 memory operations performed in the shared cache. If the size of the local caches is increased, the memory operations performed are modified. Thus, we decide to analyze the reduction of power for different shared and local cache sizes. We simulate a CMP in which the sizes of the shared cache and the local caches are doubled.

Figure 4.10 shows the percentage of power reduction with different local and shared cache sizes for each benchmark and the average for SPLASH2 and Specweb2005. 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. Figure 4.11 shows the percentage of power reduction when using 65nm and 22nm technologies. On average, when using a 22nm technology, the percentage of power reduction is 19.5% for SPLASH2 and 10.5% for Specweb2005.

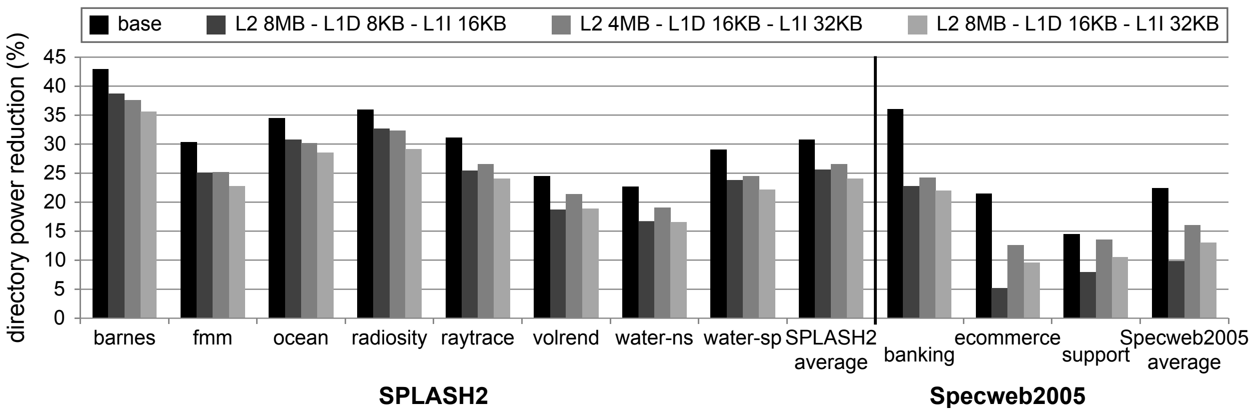


Figure 4.10: Percentage of power reduction in the directory with different local and shared cache sizes.

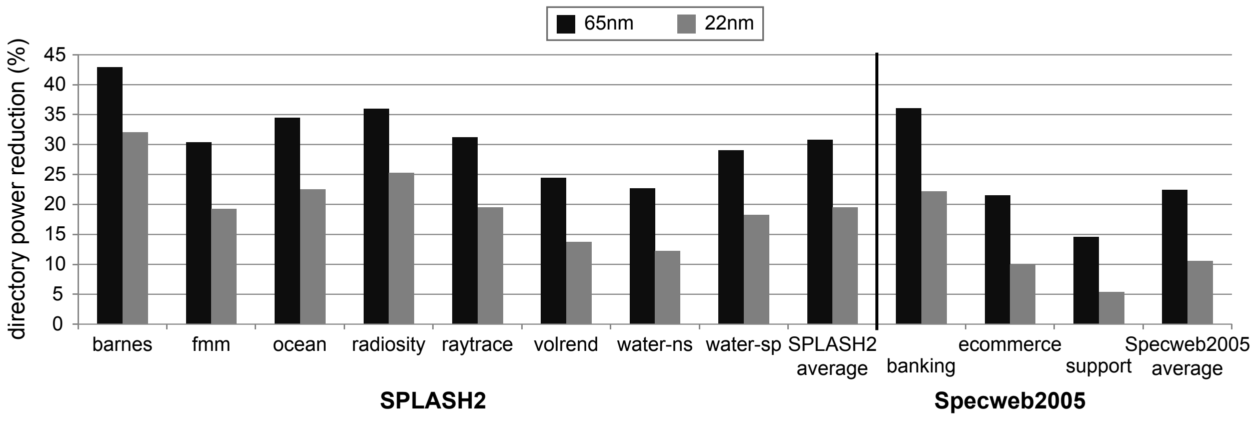
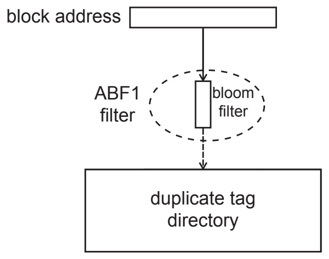


Figure 4.11: Percentage of power reduction in the directory when using 65nm and 22nm technologies.

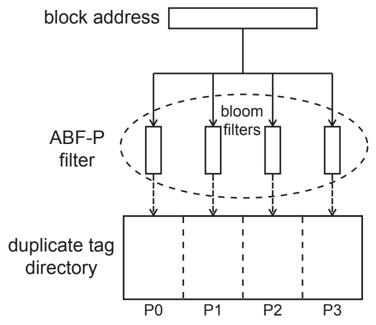
4.4 ABF Filter implementation

In this section we introduce four different implementations of the ABF filter. The difference among them is the number of bloom filters used and the specific part of the directory that is represented by each bloom filter. The four implementations described in this section are: a) using one bloom filter for each directory (data or instruction) (ABF1 filter), b) using one bloom filter for each directory and processor (ABF-P filter), c) using one bloom filter per directory, processor, and local cache way (ABF-PW filter), and d) using one bloom Filter per directory, processor, and local cache set (ABF-PS filter).

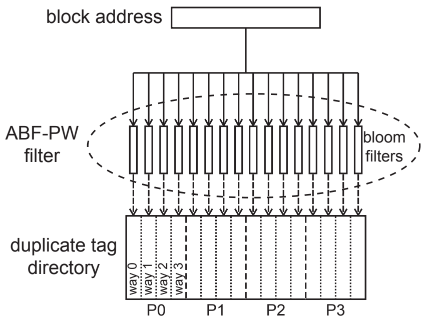
Along this section we analyze that an ABF filter has a good coverage. ABF filters can greatly reduce the number of comparisons performed by directory lookups. In fact, the number of comparisons performed by some designs exactly corresponds to the number of useful comparisons, that is, the number of comparisons that have a positive result (the target block is in the directory entry checked). However, as we will see later, it is not a power efficient design since some proposals require significantly big structures due to the large number of bloom filters required by the design or the size of the bloom filters required to get a good coverage. These structures have a high energy consumption.



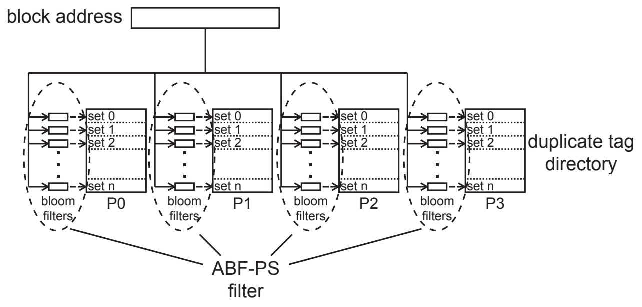
(a) ABF1 filter



(b) ABF-P filter



(c) ABF-PW filter



(d) ABF-PS filter

Figure 4.12: ABF filter designs

4.4.1 ABF Filter Designs

An ABF filter consists of several counting bloom filters and each of these bloom filters represents the blocks located in a specific part of the directory. The set of blocks represented by a bloom filter is disjoint from the set of blocks represented by any other bloom filter inside the ABF filter. All the bloom filters together represent the whole directory. Different ABF filter designs can be made depending on how we split the directory. There is always an ABF filter for the data directory (D-ABF filter) and another one for the instruction directory (I-ABF filter). Before performing a data (instruction) directory lookup, the D-ABF (I-ABF) filter is accessed, that is, a membership test of the local cache block address is performed in the corresponding bloom filters. Any positive result in the membership test of a bloom filter indicates that the part of the directory represented by that bloom filter must be accessed by the directory lookup. [1]

We propose four different ABF filters both for the data directory (D-ABF filters) and the instruction directory (I-ABF filters) based on how we split the directory among the bloom filters. From now on, we will talk about ABF filters to refer both to D-ABF filters and I-ABF filters.

Below we introduce the different designs proposed:

- **ABF1 filter** In this design, there is just one bloom filter for the whole directory

[1]The bloom filters used to implement the ABF filters are Segmented Bloom filters [22], like the bloom filters used for the DPL filter that we introduced in Section 3.2.2.

(data or instruction) (Figure 4.12(a)). The bloom filter represents the total number of blocks in the directory (`NP x LA x NLS`, being `NP` the number of processors, `LA` the local cache associativity, and `NLS` the number of local cache sets mapped to each shared cache bank).

For any positive membership test, a directory lookup of the whole directory is required. If there were no false positives, the only directory lookups performed would be the useful ones, but the directory lookup associativity would remain the same as in a system without filtering.

This filter is a DPL filter (see Section 3.2.2), but we also introduce it here to compare with the rest of the ABF filter designs.

- **ABF-P filter** The ABF-P filter requires one bloom filter per each processor (Figure 4.12(b)). Each bloom filter represents the blocks located in the local cache of a specific processor that are mapped to the shared cache bank (`LA x NLS`, being `LA` the local cache associativity, and `NLS` the number of local cache sets mapped to each shared cache bank). If we compare the ABF1 filter (Figure 4.12(a)) with the ABF-P filter (Figure 4.12(b)), we see that more bloom filters are necessary, but less blocks are represented by each of them. As a result, the bloom filters will be smaller than in the ABF1 filter to get the same effectiveness (same probability of a false positive) [17].

  An access to the ABF-P filter requires to perform a membership test in all its bloom filters. A positive membership test in a bloom filter indicates that the processor that corresponds to that bloom filter could be sharing the target block and a directory lookup is performed. The directory lookup associativity depends on the number of positive results in an access to the ABF-P filter since only the directory entries that correspond to the processors that the ABF-P filter have identified as sharers are accessed.

  This design is similar to Jetty [41]. Jetty is a filter intended to reduce the energy consumed by snoop requests in snoopy bus-based SMPs. The include-Jetty filter contains information to identify which blocks are located in the local cache of a processor by means of a Counting Bloom Filter [17]. The ABF-P filter keeps information about the blocks located in a local cache that are mapped in a shared cache bank.

- **ABF-PW filter** The ABF-PW filter consists of one bloom filter per each processor and local cache way (Figure 4.12(c)). Each bloom filter represents all the blocks located in a specific way of the local cache of a specific processor and mapped to the shared cache bank (`NLS`, being `NLS` the number of local cache sets mapped to each shared cache bank). Comparing the ABF-PW filter (Figure 4.12(c)) with the ABF-P and the ABF1 filters (Figures 4.12(b) and 4.12(a)), we can see that the ABF-PW filter is the one that requires more bloom filters, but the number of blocks represented by each of them is smaller. So, the bloom filters required in the ABF-PW to keep the same effectiveness as in ABF1 and ABF-P filters are smaller than the ones used in ABF1 and ABF-P filters.

  A membership test is performed in all the bloom filters that form a part of the ABF-PW filter. A positive membership test in a bloom filter indicates that a local copy could be located in the specific local way and processor corresponding to that bloom filter, that is, a positive membership test determines a specific directory entry in which a local copy could be located (a target block can be located only in a specific local cache set). The associativity of the directory lookups performed when using the ABF-PW is the number of positive membership tests got from the different bloom filters for the target block. If there were no false positives, all the directory entries determined by the ABF-PW filter would keep the target block. This design is the one that could exactly adjust the directory lookup associativity to the real number of local copies of the target block.

Ghosh et al. [21] proposed a similar mechanism to reduce the lookup associativity in a large set-associative cache. They propose to use a segmented counting bloom filter for each way of a cache. Before performing a cache lookup, a membership test is performed on the bloom filters attached to each cache way. The cache lookup associativity is then reduced since only the cache ways that correspond to the bloom filters with a positive result are accessed.

- **ABF-PS filter** The ABF-PS filter consists of one bloom filter per processor and local cache set mapped to the shared cache bank (Figure 4.12(d)). Each bloom filter represents the blocks located in a set of the local cache of a specific processor (`LA`, being LA the local cache associativity). Compared with the rest of the designs, the ABF-PS filter (Figure 4.12(d)) is the one that requires more bloom filters, so less blocks than in the rest are represented by each bloom filter.

  This is the only design in which an access to the filter does not perform a membership test in every bloom filter since the target block can only be located in a specific local cache set. As long as the local and shared cache block sizes are the same, for every memory operation, only the bloom filters that represent blocks of the local cache set in which the target block is located are accessed. Thus, the number of bloom filters accessed is the same as in the ABF-P filter (the number of processors). Any positive membership test indicates that a directory lookup is required. Like in the ABF-P filter, the directory lookup associativity depends on the number of positive membership tests.

  This design is similar to the one used for the Tagless Directory [61]. The Tagless Directory is a directory structure that uses an implicit and conservative representation of the blocks located in the local caches instead of the explicit representation used in conventional directory schemes. The structure organization is like a duplicate tag directory but the blocks located in each set of the local caches are represented using a bloom filter (without counters). The directory is a structure smaller than in conventional directory organizations, so it scales nicely with the number of cores but it requires to introduce several extensions to a base coherence protocol. Moreover, to maintain the directory up-to-date, it is necessary to add new coherence messages or at least to add more information to the coherence messages performed by a directory-based protocol since the Tagless Directory does not use a Counting Bloom Filter

Table 4.4 summarizes the main characteristics of the ABF filter designs described: the number of bloom filters, the number of directory entries represented by a bloom filter, the number of bloom filters accessed before performing a directory lookup, and the number of comparisons a directory lookup would perform if there were no false positives. We assume that the local cache and the shared cache block sizes are the same.

	number bloom filters	directory entries per bloom	bloom filter checked per directory	directory entries looked up per positive result
ABF1 filter	1	NP x LA x NLS	all	all
ABF-P filter	NP	LA x NLS	all	LA
ABF-PW filter	NP x LA	NLS	all	1
ABF-PS filter	NP x NLS	LA	NP	LA

Table 4.4: Number of bloom filters, number of directory entries represented by a bloom filter, number of bloom filters accessed before performing a directory lookup, and number of comparisons a directory lookup would perform if there were no false positives for each proposed design. (`NP` is the number of processors in the system, `LA` is the local cache associativity, and `NLS` is the number of local cache sets that are mapped to a shared cache bank.)

Table 4.5 shows the same information as Table 4.4 but for the CMP model we are using (Section 2.1). As we mention before, in our CMP model, we can distinguish a data and an instruction directory and an ABF filter is necessary for both of them. The ABF filter for the data directory is called D-ABF filter and the one for the instruction directory is called I-ABF filter. In this CMP model, the local and shared cache block sizes are different and so, the number of comparisons performed by a directory lookup depends on the memory operation, (Section 2.1). In Table 4.5, we assume that the directory lookup is performed by a store which is the memory operation that requires less comparisons to perform a directory lookup.

	number bloom filters	directory entries per bloom	bloom filter checked per directory	directory entries looked up per positive result
D-ABF1 filter	1	512	1	32
D-ABF-P filter	8	64	8	4
D-ABF-PW filter	32	16	32	1
D-ABF-PS filter	128	4	8	4
I-ABF1 filter	1	512	1	64
I-ABF-P filter	8	64	8	8
I-ABF-PW filter	64	8	64	1
I-ABF-PS filter	64	8	8	8

Table 4.5: Number of bloom filters, number of directory entries represented by a bloom filter, number of bloom filters accessed before performing a directory lookup, and number of comparisons a directory lookup would perform for a store if there were no false positives for each ABF filter for the data (D-ABF filter) and the instruction (I-ABF filter) directory of the CMP model used.

4.4.2 ABF Filters update

We said that we use Counting Bloom filters [17] in order to be able to remove elements from the ABF filter. The counters must be updated every time that the blocks located in the local caches change: a) a new block is allocated in the local cache, b) a block is evicted from the local cache, and c) a block is invalidated in a local cache by the shared cache. In any of these situations, only one bloom filter is affected, so only one counter has to be modified (we assume that the local and the shared cache blocks size are the same).

In the modeled CMP, the shared cache is informed simultaneously of the allocation of a new block and the eviction of another in a local cache. The only problem is that only the address of the new block is known since the local cache only indicates where the block evicted was located. The block evicted was located in the same set where the new block will be allocated, so both blocks share a part of the address. However, these bits are not enough to access the ABF filter (otherwise, both blocks would access exactly the same locations of the bloom filters that form a part of the ABF filter). The update of the evicted block should be delayed until the corresponding directory entry is read. In the system without filters, the directory is only accessed to write the information of the new block allocated. Now, it will be necessary to read first the data of the evicted block and then to write the data corresponding to the new block.

After performing a directory lookup it is known which local cache blocks will be invalidated. In the system with an ABF filter, despite of sending the corresponding invalidations to the local caches, the bloom filter counters must be updated. An invalidation only affects a specific bloom filter in any ABF filter. However, any directory lookup can determine more than one invalidation. Depending on the design chosen, all the invalidations can affect only one bloom filter or several of them. At least, all invalidations that affect the same bloom filter, need to modify exactly the same bloom filter entry (the block invalidated is always the same), so it will be necessary to be able to modify the counters by any amount.

4.4.3 ABF Filters granularity

The local cache and the shared cache block sizes are different in the modeled CMP. The shared cache block size is four times the local data cache block size and twice the local instruction cache block size. As a result, several memory operations involve more than one local cache block (for example, an eviction from the shared cache). These memory operations require several consecutive accesses to the ABF filter (one for each local cache block) and the invalidations performed by them in the local caches affect more than just one bloom filter.

As an example of the problem, Table 4.6 shows how many bloom filters are checked and the maximum number of bloom filters that have to be modified for any memory operation in a D-ABF filter. It is also included the number of entries accessed of each bloom filter, and with which number can be modified the counter of the bloom filter entry accessed.

A load-miss does not require any membership test since it does not perform a data directory lookup. However, it is necessary to include the new block and delete the evicted block from the corresponding bloom filter. As these two blocks share the same local way and set, and the modified copies are located in the same processor, these operations are always performed in the same bloom filter regardless of the ABF filter design. As a result, any counter bloom filter requires, at least, two ports.

An ifetch-miss requires to eliminate any copy of the block from the local data caches. As the local instruction block size is twice the local data block size, two different local data cache blocks have to be checked. Depending on the design, both blocks can belong to the same bloom filter (ABF-P filter) or to different bloom filters (ABF-PS).

A store only requires to eliminate the copies of the target local data cache block.

		checked		modified		
		bloom filters	entries per bloom filter	bloom filters	entries per bloom filter	value reduced
D-ABF1 filter	load-miss	0	0	1	2	1
	ifetch-miss	1	2	1	2	8
	store	1	1	1	1	7
	eviction	1	4	1	1	8
D-ABF-P filter	load-miss	0	0	1	2	1
	ifetch-miss	8	2	8	2	1
	store	8	1	7	1	1
	eviction	8	4	8	4	1
D-ABF-PW filter	load-miss	0	0	1	2	1
	ifetch-miss	32	2	16	1	1
	store	32	1	7	1	1
	eviction	32	4	32	4	1
D-ABF-PS filter	load-miss	0	0	1	2	1
	ifetch-miss	16	1	16	1	1
	store	8	1	7	1	1
	eviction	32	1	32	1	1

Table 4.6: Number of bloom filter checked and modified by the different memory operation in each D-ABF filter. It is also included the number of entries accessed and the value by which the counter bloom filter can be modified.

The number of bloom filters to be checked and modified depends on the ABF filter design. However, only one entry per bloom filter needs to be accessed.

As the shared cache is inclusive, an eviction from the shared cache requires to eliminate all the local copies of the target block. The shared cache block size is four times the local data cache block size. As a result, four different local data cache blocks have to be checked in the ABF filter, and later, if the local copies exist, the same bloom filters should be modified. Depending on the design, all the blocks can belong to the same bloom filter or to different bloom filters.

In Table 4.6 we can observe that if we use the local cache block address to access the ABF1, the ABF-P, or the ABF-PW filters, more than one bloom filter entry has to be accessed for some memory operations either to perform membership tests or to update the information in the bloom filter. In order to do this, we have two solutions: a) to implement ABF filters with several ports, or b) to allow the design to perform several consecutive accesses to the ABF filter for one memory operation (delaying the directory lookup and future memory operations). Both solutions increase the power consumed by the ABF filter.

We propose to use the shared cache block address to access the ABF filters. By doing this, the number of bloom filter entries accessed in each access is limited to 1. The only drawback is that the filter accuracy could be reduced since the number of false positives could increase, for example, if there is a local copy of any subblock of a shared cache block in a processor, any check of any other subblock of the same shared cache block will produce a positive result.

Table 4.7 shows the same information as Table 4.6 when using the shared cache block address to access the D-ABF filters. We do not include the D-ABF-PS filter since it is the only one that can not be benefitted from using the shared cache block to access the ABF filter.

		checked		modified		
		bloom filters	entries per bloom filter	bloom filters	entries per bloom filter	value reduced
ABF1 filter	load-miss	0	0	1	2	1
	ifetch-miss	1	1	1	1	16
	store	1	1	1	1	7
	eviction	1	1	1	1	32
ABF-P filter	load-miss	0	0	1	2	1
	ifetch-miss	8	1	8	1	2
	store	8	1	7	1	1
	eviction	8	1	8	1	4
ABF-PW filter	load-miss	0	0	1	2	1
	ifetch-miss	32	1	16	1	2
	store	32	1	7	1	1
	eviction	32	1	32	1	4

Table 4.7: Number of bloom filter checked and modified by the different memory operation when using the shared cache block address to index the different D-ABF filter designs (except for the D-ABF-PS filter). It is also included the number of entries accessed and the value by which the counter bloom filter can be modified.

4.4.4 ABF Filters effectiveness

The effectiveness of any of the proposed designs depends on keeping the number of false positives low. A false positive wrongly reports that the directory part represented by a bloom filter contains a copy of the block. Thus, the directory lookup has to check that part of the directory in which there is no copy of the target block. If the number of false positives is high, the ABF filters would be useless since they would not reduce the number of comparisons performed by directory lookups.

For all the ABF filters, we try several bloom filters of different sizes and with different hash functions (type and number) in order to analyze their effectiveness. We choose two types of hash functions: based on bit shifts and based on XOR (exclusive or) operation [61]. 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 exclusive or 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*).

Figure 4.13 shows the percentage of comparisons performed by directory lookups in the data directory for each workload of the SPLASH2 suite when using the different D-ABF filters compared to the system without filtering. Each line in each graph corresponds to a benchmark in the SPLASH2 suite. For the D-ABF1, the D-ABF-P, and the D-ABF-PW filters, there are two graphs: one for the results when accessing the D-ABF filter with the local cache block address and one for the results when accessing the D-ABF filter with the

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

Table 4.8: Different combination of hash functions used to analyze the effectiveness of the ABF filters.

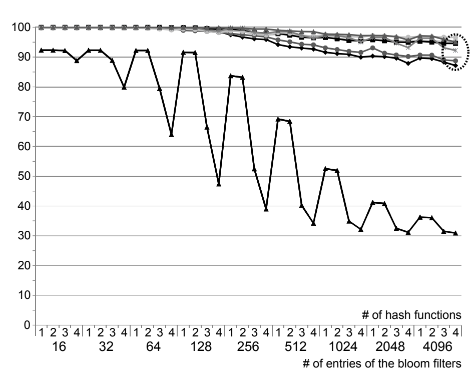
shared cache block address. For the D-ABF-PS filter, there is just one graph since this filter does not get any benefit from using the shared cache block address to access it. We use different bloom filter sizes and hash functions for every proposed D-ABF filter. The number of entries of the bloom filters used varies from 16 to 4096. The number of hash functions varies between 1 and 4 (Table 4.8 shows how we combine the hash functions when we use 1, 2, 3, or 4 hash functions.). 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 filters 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.

In Figure 4.13, we can observe that all benchmarks have a similar behavior except `ocean` when using the D-ABF1 filter. Table 4.3 shows that, in general, most of the comparisons performed in the data directory are due to stores. By contrast, in `ocean`, stores and evictions perform 53% and 47% of the comparisons in the data directory, respectively. A membership test of the bloom filter of a D-ABF1 filter of a target block accessed by a store is positive since most of the stores are performed over private data (see Figure 4.1). Thus, the data directory lookups performed by stores cannot be filtered out (even if there is just one copy of the target block). A membership test of a block accessed by an eviction, in `ocean`, generally produces negative results, and so, lookups performed by evictions might be avoided using the D-ABF1 filter. Therefore, the behavior of `ocean` when using the D-ABF1 filter is different from the rest of the benchmarks.

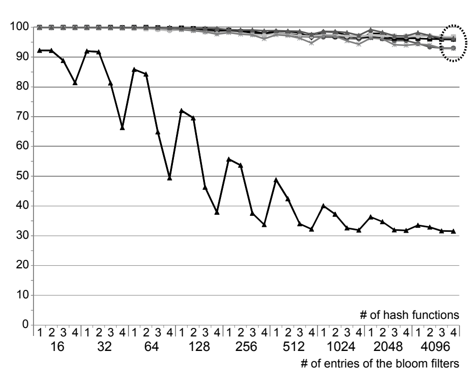
The D-ABF1 filter (Figures 4.13(a) and 4.13(b)) is not useful since it only reduces the number of comparisons performed for `ocean`. For the rest of the benchmarks, even using significantly big bloom filters, the number of comparisons performed is reduced to 85% of the comparisons performed in the system without filters. The rest of the D-ABF filters reduce the number of comparisons below 15% for any benchmark.

In general, it is better to use D-ABF filters with bloom filters that use three or four hash functions instead of increasing the bloom filter size by three or four. As we expect, the D-ABF filters that use bloom filters that represent a bigger fraction of the directory, require to use bigger bloom filters to get the same effectiveness (reduce the number of comparisons in the same percentage). For example, the D-ABF-P filter using bloom filters of 2048 entries and 4 hash functions reduces the number of data directory comparisons to 10%, on average, while the D-ABF-PS filter using bloom filters of 128 entries and 4 hash functions reduces the number of data directory comparisons to 10.7%, on average. Each bloom filter of a D-ABF-P filter represents 64 blocks while each bloom filter of a D-ABF-PS filter only represents 4 blocks.

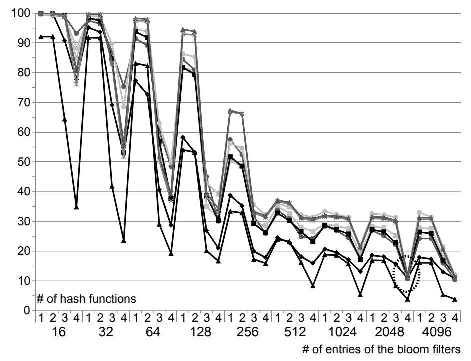
If we compare each D-ABF filter when using the shared or the local cache block address



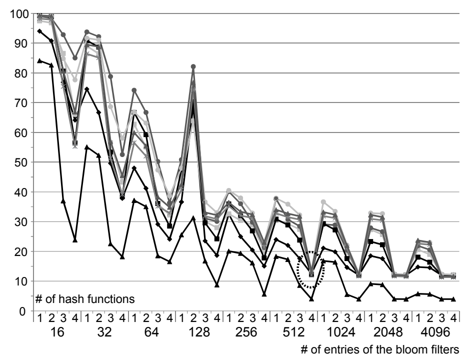
(a) ABF1 filter accessed by the local cache block address



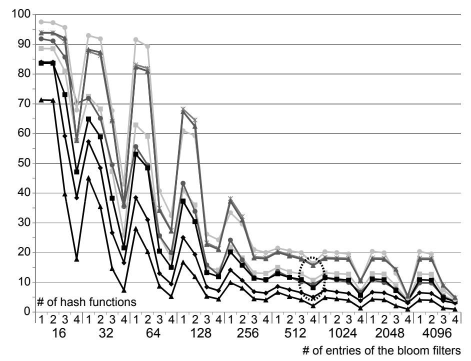
(b) ABF1 filter accessed by the shared cache block address



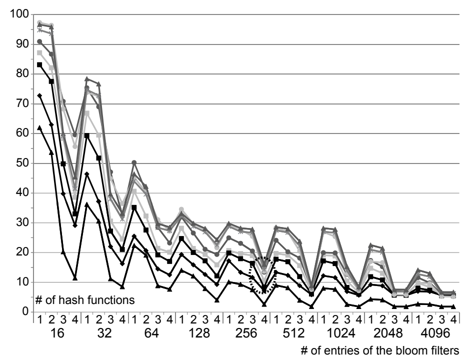
(c) ABF-P filter accessed by the local cache block address



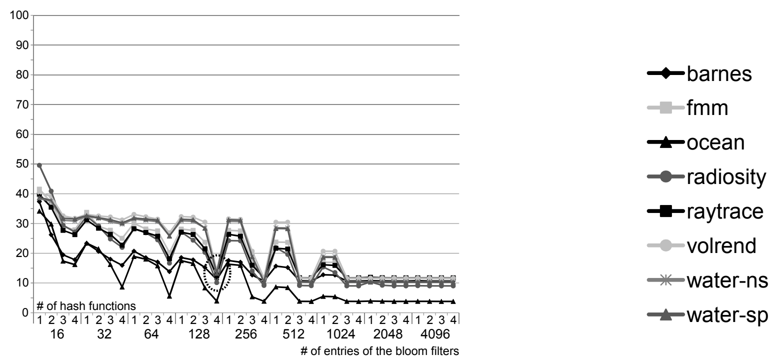
(d) ABF-P filter accessed by the shared cache block address



(e) ABF-PW filter accessed by the local cache block address



(f) ABF-PW filter accessed by the shared cache block address



(g) ABF-PS filter accessed by the local cache block address

Figure 4.13: Percentage of comparisons performed in the data directory when using the different D-ABF filters compared to the system without filtering.

to access the filter, we observe that the number of comparisons performed for the same bloom filter size is smaller when using the shared cache block address (Figure 4.13(d) compared to Figure 4.13(c) or Figure 4.13(f) compared to Figure 4.13(e)). Therefore, the D-ABF filters accessed using the shared cache block address require smaller bloom filters. The reason is that when using the shared cache block address, the number of comparisons that have to be performed per positive result in any bloom filter is sometimes bigger, however, the number of times that any bloom filter is checked is reduced. For example, in the D-ABF-PW filter, an ifetch-miss requires two accesses to each bloom filter when using the local data cache block address (the local instruction cache block size is twice the local data cache block size). For every positive result, a directory entry has to be accessed to check if the target block is located there. If the shared cache block address is used to index the bloom filters, each bloom filter of the D-ABF-PW filter is accessed just once. However, for every positive result, two directory entries have to be checked. So, though the number of comparisons can not be reduced in the same way, when using the shared cache block address to index the bloom filters, the energy consumed by the filter will be lower. Moreover, as the number of blocks represented by any bloom filter is sometimes reduced (different subblocks of the same shared cache block are kept in the local caches), the conflicts in the bloom filters are reduced, so the D-ABF filter effectiveness increases.

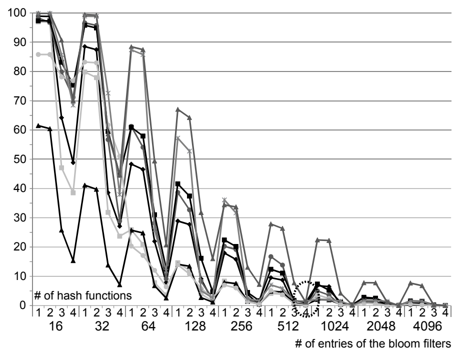
Figure 4.14 shows the same information as Figure 4.13 for the I-ABF filters. We can see that all the I-ABF filters reduce the number of comparisons performed in the instruction directory, unlike the D-ABF1 filter that was not useful to reduce the number of comparisons in the data directory.

Figure 4.14 shows that, like in the data directory, those filters in which a bloom filter represents less blocks of the directory require smaller bloom filters. For example, the I-ABF-P filter requires bloom filters with 512 entries and 4 hash functions (see Figure 4.14(c)) to reduce the number of comparisons in the instruction directory to 0.2% with respect to the system without filtering. The I-ABF-PS and the I-ABF-PW filters using bloom filters with 64 entries and 4 hash functions also reduce the number of comparisons in the instruction directory to 0.2%. Like for the D-ABF filters, when using the shared cache block address to access the I-ABF filters, the effectiveness of the filter improves.

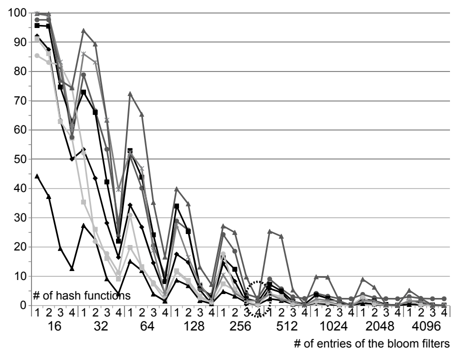
We decide to choose the bloom filter sizes and the number of hash functions that either reduce the number of comparisons to the lower limit or that taking a bigger bloom filter barely reduces more the number of comparisons. In Figures 4.13 and 4.14, we indicate the bloom filter size chosen for each ABF filter.

4.4.5 ABF Filters overhead

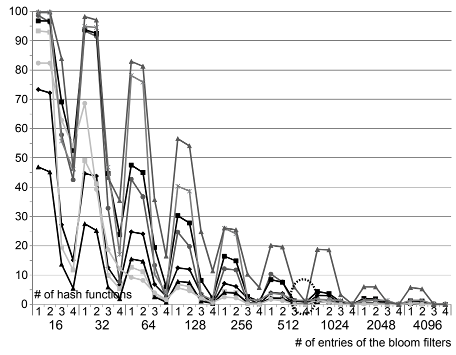
The bloom filters used in the different ABF filters consist of a bit vector and a counter array. The size of the whole bit vector of any ABF filter is the number of bloom filters used multiplied by the number of entries they have and the number of hash functions used. The size of the whole counter array is the size of the bit vector multiplied by the number of bits that the counter requires ($log_2(number\ of\ blocks\ represented\ by\ the\ bloom\ filter) + 1 valid bit$). Table 4.9 indicates, for each ABF filter, the bloom filter size and number of hash functions chosen (Figures 4.13 and 4.14), and the size of the bit vector and the counter array



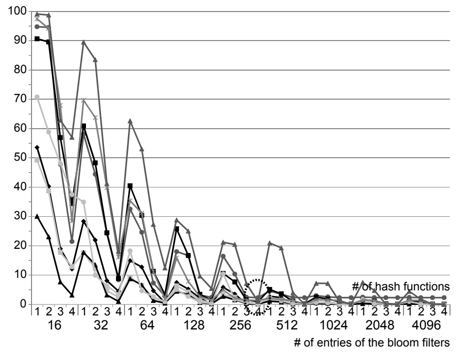
(a) I-ABF1 filter accessed by the local cache block address



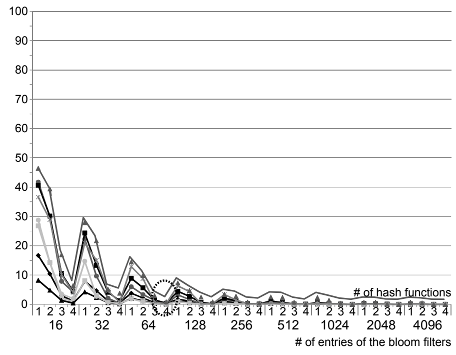
(b) I-ABF1 filter accessed by the shared cache block address



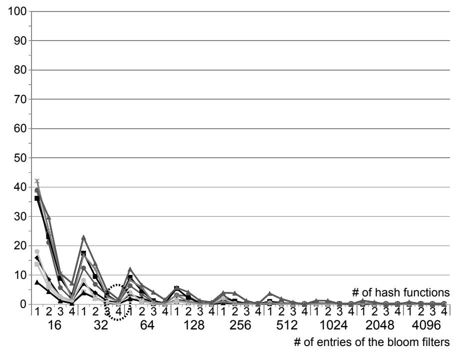
(c) I-ABF-P filter accessed by the local cache block address



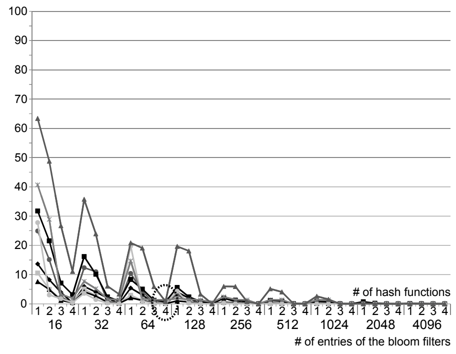
(d) I-ABF-P filter accessed by the shared cache block address



(e) I-ABF-PW filter accessed by the local cache block address



(f) I-ABF-PW filter accessed by the shared cache block address



(g) I-ABF-PS filter accessed by the local cache block address

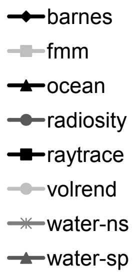


Figure 4.14: Percentage of comparisons performed in the instruction directory when using the different I-ABF filters compared to the system without filtering.

that form a part of any ABF filter.

	cache block address used	number of bloom filters	number of entries	number of hash functions	size (Bytes) bit vector	size (Bytes) counter array
D-ABF1 filter	local	1	4096	4	2KB	20KB
D-ABF1 filter	shared	1	4096	4	2KB	20KB
D-ABF-P filter	local	8	2048	4	8KB	56KB
D-ABF-P filter	shared	8	512	4	2KB	14KB
D-ABF-PW filter	local	32	512	4	8KB	40KB
D-ABF-PW filter	shared	32	256	4	4KB	20KB
D-ABF-PS filter	local	128	128	4	8KB	24KB
I-ABF1 filter	local	1	512	4	256B	2.5KB
I-ABF1 filter	shared	1	256	4	128B	1.25KB
I-ABF-P filter	local	8	512	4	2KB	14KB
I-ABF-P filter	shared	8	256	4	1KB	7KB
I-ABF-PW filter	local	64	64	4	2KB	8KB
I-ABF-PW filter	shared	64	32	4	1KB	4KB
I-ABF-PS filter	local	64	64	4	2KB	6KB

Table 4.9: Number of bloom filters of each ABF filter, bloom filters size and hash functions chosen in the previous section for each design, and the size in bytes of the bit vector and the counter array of each ABF filter per shared cache bank.

Table 4.9 shows that the size of both the counter array and the bit vector is always bigger than the data or instruction directory. (The size of the data or instruction directory is 1KB per shared cache bank.) The only exception is the I-ABF1 filter when it is accessed by the shared cache block address. The size of the counter array of the I-ABF1 filter is approximately the size of the instruction directory and the bit vector is the smallest one. The rest of the filters are significantly big.

It is interesting to notice that, except the ABF1 filter, the bit vector of all the ABF filters have the same size if they are accessed in the same way (either with the local cache block address or the shared cache block address). When the number of bloom filters in a filter is bigger that in other, the number of blocks represented by each bloom filter is reduced, so a bloom filter with less entries keeps the effectiveness of the ABF filter.

To decide which design is better we can consider not only the size it requires, but the number of bloom filters that have to be accessed each time. For example, when using the ABF-P filter only 8 bloom filters might be accessed per directory lookup. However, when using the D-ABF-PW (I-ABF-PW) filter, 32 (64) bloom filters must be accessed. Moreover, it is more interesting to use the filters that are accessed using the shared cache block address since they achieve the same effectiveness with smaller bloom filters.

We do not expect these designs to reduce the power in the directory since the size of the structures is significantly big. All the ABF filters filter out approximately the same amount of comparisons and their sizes are of the same order. Thus, in the next section, we measure the power reduction attained with one of the ABF filters as an example of the power reduction that we could get with the rest of the designs. For these preliminary results, for the data directory, we choose the D-ABF-P filter accessed by the shared cache block address, and, for the instruction directory, we choose the I-ABF1 filter accessed using the shared cache block address.

4.4.6 ABF Filters power reduction

We use CACTI [43] to estimate the dynamic energy and leakage power for the cache tag array, the directory, and the proposed ABF filters (see Section 2.2 for more details).

As we said before, in this section, we measure the power reduction attained with one of the ABF filters as an example of the results we can get with these kind of filters. For the instruction directory, we choose the I-ABF1 filter. This design was evaluated previously in Section 3.3.6. The only difference was the hash functions used. In Section 3.3.6, we combine the functions *s3* and *s4*, reducing the size of the bloom filter and the number of hash functions used. As it was a better hash function combination, we do not repeat here the same results.

For the data directory, we choose the D-ABF-P filter accessed by the shared cache block address. Figure 4.15 shows the percentage of the data directory consumed when using the D-ABF-P filter compared to the system without filtering. To simplify the design, for the D-ABF-P filter we only count the power consumed by the bit vector and we do not measure the power consumed by the counter array (which is a significantly bigger structure).

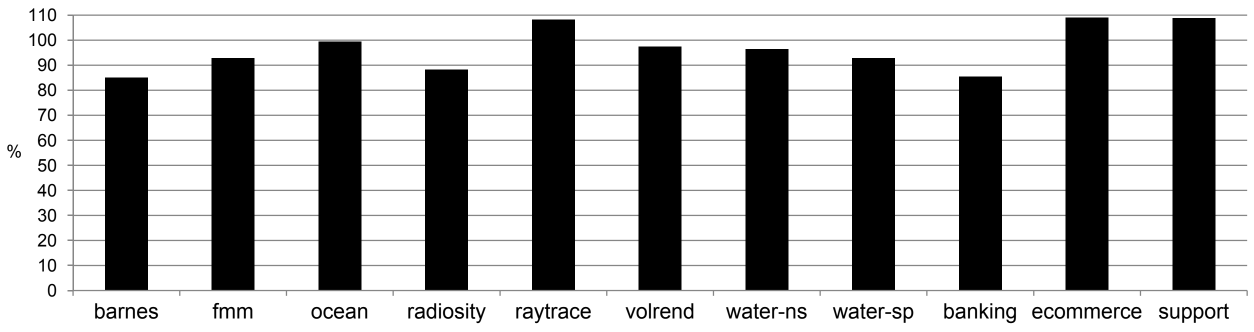


Figure 4.15: Percentage of data directory power when using the D-ABF-P filter accessed using the shared cache block address compared to the system without filtering. For the D-ABF-P filter, we only count the power consumed by the bit vector.

Figure 4.15 shows that, on average, the data directory power consumption is only reduced, on average, to 96% compared to the system without filtering. This reduction is too small. Moreover, in some benchmarks, this filter increases the power consumption. Numbers in Figure 4.15 only take into account the power consumed by the bit vector of the filter. We do not measure the power consumption of the counter array which is an structure bigger than the bit vector. If we count the power consumed by the whole filter (bit vector and counter array), the filter will increase the power consumed by the directory for all the benchmarks.

4.5 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 is private to the processor performing 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 Owner filter. This 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 Owner 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 performed by directory lookups 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.

The performance of the Owner filter is limited since it only reduces energy when specific situations take place: an owner exists, or there are no copies in any local cache. We propose to extend the filter so it can determine which local caches keep copies of a target block. In order to keep the filter structure small, we keep a superset of the local copies located in the local caches. We propose to use a filter implemented as an Array of Bloom Filters: ABF filter. Each bloom filter represents a part of the directory. For every directory lookup, the ABF filter is accessed and only when there is a positive result in the bloom filters that form a part of the ABF filter, the corresponding part of the directory is looked up. We propose different designs depending on the part of the directory represented by each bloom filter. We conclude that the ABF filters coverage is really good, but the energy consumed by the directory is not reduced since the filter structures are significantly big.

Chapter 5

Conclusions

This chapter summarizes the main contributions of this thesis and describes the publications derived from this thesis' research. It also includes some of the future lines of research arising from this thesis.

Along this thesis we have analyzed different solutions to reduce the power consumed by a coherence directory implemented as a duplicate tag directory in a CMP with write-through local caches. We have focused in two basic filtering mechanisms. The first filtering mechanism reduces the number of directory lookups performed while the second one reduces the directory lookup associativity.

We propose the first filtering mechanism because we realize that an important fraction of directory lookups are useless since 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. To reduce the energy wasted by useless directory lookups, we propose to use a filter before accessing the directory which is able to identify in advance whether a lookup is useless or not.

In a CMP with local caches split in data and instruction and a coherence directory implemented as a duplicate tag, we can distinguish a data and an instruction directory. Any store that access the shared cache or any eviction from the shared cache (inclusive shared cache) require to perform a data and an instruction directory lookup. However, the sets of memory addresses of instructions and data are, in general, disjoint. As a result, the local copies of a target block are located either in the local data caches or the local instruction caches, but rarely in both of them. As the data and instruction directory are a duplicate of the local cache tag arrays, only the data directory or the instruction directory can be useful for the same target block.

We propose two implementations of the first filtering mechanism: the ID filter and the

DPL filter. The ID filter exploits the inclusion property of the shared cache to label each block in the shared cache with the stream it belongs to (data or instruction). A directory lookup performed over a block labeled as data (instruction) only performs a data (instruction) directory lookup. The DPL filter keeps the information of all blocks belonging to a stream together using a filter structure decoupled from the shared cache size. A DPL filter consists of an I-DPL filter that keeps a superset of the blocks located in the instruction directory and a D-DPL filter that keeps a superset of the blocks located in the data directory. Both the I-DPL filter and the D-DPL filter are based on a bloom filter. Before performing a directory lookup, the I-DPL filter and the D-DPL filter are accessed and they determine if a data or instruction directory lookup must be performed (instruction fetches (loads) that miss in the local instruction (data) cache only require to access the D-DPL (I-DPL) filter).

We propose several ID filter implementations and any of these ID filters reduce the power consumed by the directory by 28% and 19% for SPLASH2 and Specweb2005, respectively, taking into account both the dynamic energy and the leakage power of the directory and the filter. A DPL filter that uses both an I-DPL filter and a D-DPL filter is not an interesting design since the D-DPL filter has a low coverage. Due to that, we decide to implement a design using only an I-DPL filter (it does not filter out data directory lookups). The I-DPL filter reduces the power consumed by the whole directory (data and instruction directories) by 27% and 9% for SPLASH2 and Specweb2005, respectively.

We describe and analyze the ID filter in the following paper:

- Ana Bosque, Víctor Viñals, Pablo Ibáñez, and José M. Llabería, "Filtering Directory Lookups in CMPs", in Proceedings of the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD), pp. 207-216, September 2010.

The DPL filter is described and analyzed together with a deeper analysis of the ID filter in the following paper:

- Ana Bosque, Víctor Viñals, Pablo Ibáñez, and José M. Llabería, "Filtering Directory Lookups in CMPs", in Microprocessors and Microsystems (accepted for publication).

The second filtering mechanism is proposed because we realize that, in general, the directory lookups that are useful are performed over blocks that are shared by a small number of processors. A coherence directory implemented as a duplicate tag require a high associative directory lookup that consumes a significant amount of energy. If we identify which local caches are sharing a target block before accessing the directory, the directory lookup associativity would be much smaller, reducing the energy consumed by the directory.

We propose two implementations of the second filtering mechanism: the Owner filter and the ABF filter. The Owner filter keeps explicit information for every block in the shared cache identifying the owner of the block (processor that keeps in its local cache the only local copy of the block) whenever it exists. If the owner of a target block is known before performing a directory lookup, only the directory entries that correspond to that processor are accessed.

The ABF filter, using an Array of Bloom Filters, keeps a superset of the local copies of any block located in any specific part of the directory. Before performing a directory lookup, the ABF filter is accessed and it determines which directory entries must be checked during the directory lookup. The number of directory entries determined by the ABF filter is much smaller than the total number of entries where the local copies of the target block can be located.

The Owner filter reduces the power consumed by the directory by 31% and 22% for SPLASH2 and for Specweb2005, respectively, taking into account both the dynamic energy and the leakage power of the directory and the filter. We propose several ABF filter designs. All of them have a good coverage, however, they are not power efficient designs due to the large size of their filtering structures.

We describe and evaluate the Owner filter in the following paper:

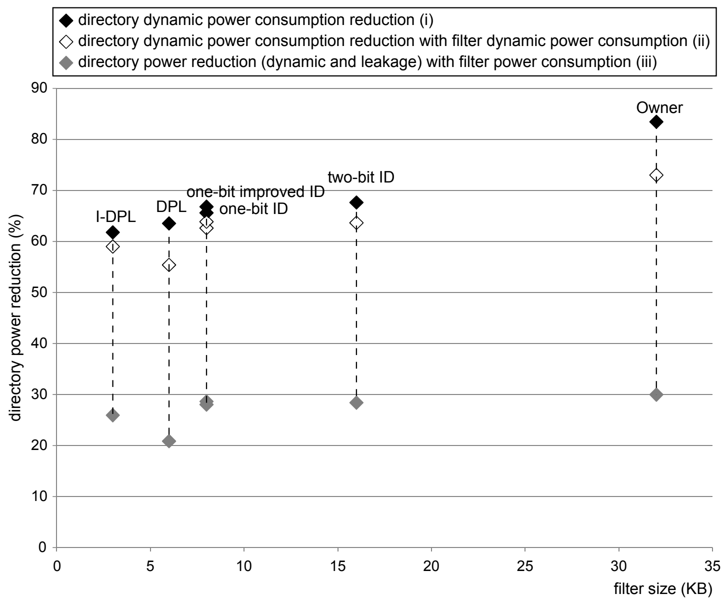
- Ana Bosque, Víctor Viñals, Pablo Ibáñez, and José M. Llabería, "Filtering Directory Lookups in CMPs with Write-through Caches", in Proceedings of the 2011 Euro-Par Conference, pp. 267-279, August 2011.

Before implementing and evaluating the proposed filtering mechanisms, we had to develop a simulator detailed enough for the kind of mechanisms we wanted to evaluate. We also needed to analyze in detail the benchmarks we would use to evaluate our proposals. SPLASH2 was analyzed with great detail by Woo et al. [50], but Specweb2005 has not been analyzed. We analyzed it in the following paper:

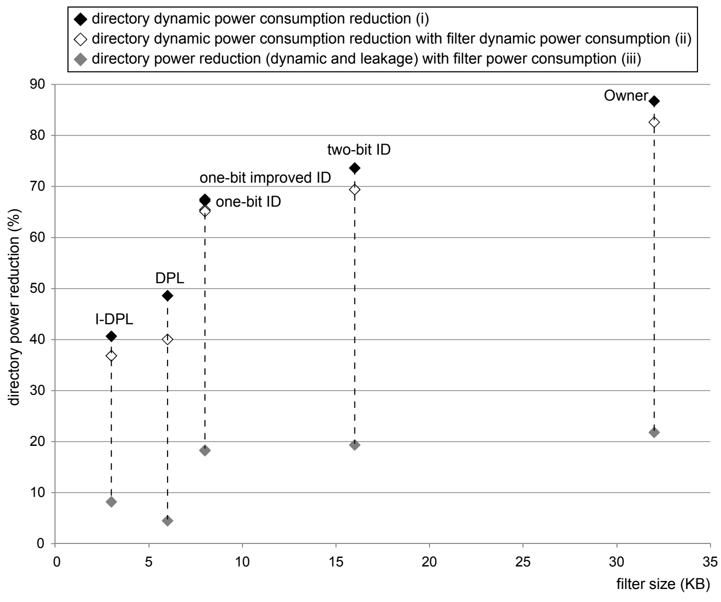
- Ana Bosque, Pablo Ibáñez, Víctor Viñals, Per Stenström, and José M. Llabería, "Characterization of Apache web server with Specweb2005", in Proceedings of the 2007 workshop on MEmory performance (MEDEA '07), pp. 65-72, September 2007.

Now, we will compare the proposed filtering mechanisms. Figure 5.1 shows the average power reduction achieved by each proposed filter and the filter size for the whole system (adding the filter size in each shared cache bank). We show three points for each proposed filter: i) the directory dynamic power reduction without taking into account the filter power consumption (dynamic and leakage), ii) the directory dynamic power reduction considering the dynamic power consumed by the filter, and iii) the directory power reduction (dynamic and leakage) taking into account the power consumed by the filter (dynamic and leakage). Figure 5.1 does not show the power reduction for the ABF filters since the numbers shown in this thesis for those filters are only an optimistic approximation. Moreover, the size of any ABF filter design is much bigger than the rest of the proposals and it does not achieve a directory power reduction as good as other filters.

We can see in Figure 5.1 that the Owner filter achieves the biggest directory power reduction. However, the difference between the Owner filter and the rest of the proposed filters is small when we take into account the filter power consumption and the leakage power (iii). This is due to the big size of the Owner filter. It is also interesting to observe that the two-bit ID filter reduces more than the one-bit or the one-bit improved ID filters the dynamic power consumption of the directory. However, when we take into account the filter power consumption and the leakage (iii), the one-bit and the one-bit improved ID filters reduce more the directory power consumption since the two-bit ID filter size is twice the one-bit ID filter size. In the same way, the DPL filter reduces more the directory dynamic power consumption, but the I-DPL filter achieves better results when we take into account the filter power consumption.



(a) SPLASH2



(b) Specweb2005

Figure 5.1: Percentage of power reduction achieved by the proposed filters and the filter size. (a) shows the average power reduction for SPLASH2 and (b) shows the average power reduction for Specweb2005. All the points linked by a dot line correspond to the same filter.

A future line of research is to reduce the power consumed by the coherence protocols in many-core systems, that is, in systems with hundreds or even thousands of cores. As have been mentioned along this thesis, a practical implementation of a many-core system would be to organize it in small clusters and perform a two-level cache coherency: one inside each cluster and another one among clusters. Each of these clusters would be the CMP modeled in this thesis, which uses write-through local caches. The shared cache inside each cluster would be write-back and private for that particular cluster. The second coherency level needs to be implemented to keep coherence among clusters.

Filters proposed in this thesis could therefore be implemented inside each cluster, but it would be necessary to asses their performance in such a new environment. The coherence protocol among clusters will probably have a different behavior compared to the coherence protocol inside the cluster, so further analysis would be necessary in order to propose a filter that reduces the energy consumption of the coherence protocol among clusters.

Bibliography

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *ISCA-15*, pages 280–289, 1988.

[2] N. Agarwal, L.-S. Peh, and N. Jha. In-Network Coherence Filtering: Snoopy coherence without broadcasts. pages 232 –243, 2009.

[3] A. R. Alameldeen and D. A. Wood. Variability in Architectural Simulations of Multi-Threaded Workloads. In *HPCA-9*, page 7, 2003.

[4] J. Archibald and J. L. Baer. An economical solution to the cache coherence problem. In *ISCA-11*, pages 355–362, 1984.

[5] 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*, pages 60–69, 2008.

[6] Barroso, Luiz André et. al. Piranha: a Scalable Architecture Based on Single-Chip Multiprocessing. In *ISCA-27*, pages 282–293, 2000.

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

[8] J. Cantin, M. Lipasti, and J. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *ISCA-32*, pages 246–257, June 2005.

[9] L. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *Computers, IEEE Transactions on*, C-27(12):1112–1118, Dec. 1978.

[10] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *ASPLOS-4*, pages 224–234, 1991.

[11] A. Charlesworth, N. Aneshansley, M. Haakmeester, D. Drogichen, G. Gilbert, R. Williams, and A. Phelps. The Starfire SMP Interconnect. pages 37 – 37, 1997.

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

[13] M. Dubois and et al. The Detection and Elimination of Useless Misses in Multiprocessors. In *ISCA-20*, pages 88–97, 1993.

[14] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *ISCA-13*, pages 434–442, 1986.

[15] 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.

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

[17] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. 8:281–293, June 2000.

[18] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo directory: A scalable directory for many-core systems. In *HPCA-17*, pages 169–180, 2011.

[19] S. J. Frank. Tightly Coupled Multiprocessor System Speeds Memory-Access Times. *Electronics*, 57:164–169, January 1984.

[20] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA-17*, pages 15–26, 1990.

[21] M. Ghosh, E. Ozer, S. Ford, S. Biles, and H.-H. S. Lee. Way guard: a segmented counting bloom filter approach to reducing energy for set-associative caches. In *ISLPED '09*, pages 165–170, 2009.

[22] M. Ghosh, E. Özer, S. Biles, and H. hsin S. Lee. Efficient System-on-Chip energy management with a segmented Bloom filter. In *ARCS-19*, pages 283–297, 2006.

[23] J. R. Gooadman. Cache Consistency and Sequential Consistency. Technical report, University of Wisconsin-Madison, 1989.

[24] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *ISCA-10*, pages 124–131, 1983.

[25] A. Gupta, W. dietrich Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *ICPP'90*, pages 312–321, 1990.

[26] http. //httpd.apache.org/docs/2.0/.

[27] http. //www.spec.org/web2005/.

[28] N. Jerger. SigNet: Network-on-chip filtering for coarse vector directories. pages 1378 –1383, 2010.

[29] T. Johnson and U. Nawathe. An 8-core, 64-thread, 64-bit Power Efficient SPARC SOC (niagara2). In *ISPD '07*, pages 2–2, 2007.

[30] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. Power7: IBM's Next-Generation Server Processor. *IEEE Micro*, 30:7–15, 2010.

[31] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *ISCA-12*, pages 276–283, 1985.

[32] S. Kottapalli and J. Baxter. Nehalem-EX CPU Architecture. 2009.

[33] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28:690–691, September 1979.

[34] J. Laudon and D. Lenoski. The SGI Origin: A ccnuma Highly Scalable Server. pages 241 –251, 1997.

[35] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO-42*, pages 469 –480, 2009.

[36] 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*, 35(2):50–58, Feb 2002.

[37] T. Maruyama, T. Yoshida, R. Kan, I. Yamazaki, S. Yamamura, N. Takahashi, M. Hondou, and H. Okano. Sparc64 VIIIfx: A New-Generation Octocore Processor for Petascale Computing. *IEEE Micro*, 30:30–40, 2010.

[38] E. McCreight. The Dragon computer system: An early overview. Technical report, Xerox Corp., 1984.

[39] M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi. How to Simulate 1000 Cores. *SIGARCH Comput. Archit. News*, 37(2):10–19, 2009.

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

[41] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers. In *HPCA-7*, pages 85–96, 2001.

[42] S. S. Mukherjee and M. D. Hill. An evaluation of directory protocols for medium-scale shared-memory multiprocessors. In *ICS-8*, pages 64–74, 1994.

[43] N. Muralimanohar and R. Balasubramonian. CACTI 6.0: A Tool to Model Large Caches, 2009.

[44] Nanda, A. K. et. al. High-throughput coherence control and hardware messaging in Everest. *IBM Journal of Research and Development*, 45(2):229 –243, 2001.

[45] B. O'Krafka and A. Newton. An empirical evaluation of two memory-efficient directory methods. pages 138 –147, 1990.

[46] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA-11*, pages 348–354, 1984.

[47] S. Patel, S. Phillips, and A. Strong. Rainbow Falls: Sun's Next Generation CMT Processor. 2009.

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

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

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

[51] SPEC. Specweb2005 release 1.10 benchmark design document. *Technical Whitepaper*, 2006.

[52] M. B. Steinman, G. J. Harris, A. Kocev, V. C. Lamere, R. D, and Pannell. The AlphaServer 4100 Cached Processor Module Architecture and Design, 1996.

[53] K. Strauss, X. Shen, and J. Torrellas. Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors. *SIGARCH Comput. Archit. News*, 34(2):327–338, 2006.

[54] Sun Microsystems, Inc. *The SPARC Architecture Manual, Version 8*, January 1991.

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

[56] C. K. Tang. Cache System Design in the Tightly Coupled Multiprocessor System. In *AFIPS '76*, pages 749–753, 1976.

[57] M. Thapar. Interleaved dual tag directory scheme for cache coherence. pages 546 –553, 1994.

[58] D. Wallin and et al. Vasa: A simulator infrastructure with adjustable fidelity. In *PDCS-2005*, pages 554–563, 2005.

[59] D. Weaver and T. Germond. *The SPARC Architecture Manual, Version 9*. SPARC International, Inc., 1994.

[60] W. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. *SIGARCH Comput. Archit. News*, 17(2):243–256, 1989.

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