Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks

Blas Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José Duato
Department of Computer Engineering
Universitat Politècnica de València
Camino de Vera, s/n, 46021, Valencia, Spain
{blacuesa, aros, megomez, arobles, jduato}@gap.upv.es

ABSTRACT

To meet the demand for more powerful high-performance shared-memory servers, multiprocessor systems must incorporate efficient and scalable cache coherence protocols, such as those based on directory caches. However, the limited directory cache size of the increasingly larger systems may cause frequent evictions of directory entries and, consequently, invalidations of cached blocks, which severely degrades system performance.

A significant percentage of the referred memory blocks are only accessed by one processor (even in parallel applications) and, therefore, do not require coherence maintenance. Taking advantage of techniques that dynamically identify those private blocks, we propose to deactivate the coherence protocol for them and to treat them as uniprocessor systems do. The protocol deactivation allows directory caches to omit the tracking of an appreciable quantity of blocks, which reduces their load and increases their effective size. Since the operating system collaborates on the detection of private blocks, our proposal only requires minor modifications.

Simulation results show that, thanks to our proposal, directory caches can avoid the tracking of about 57% of the accessed blocks and their capacity can be better exploited. This contributes either to shorten the runtime of parallel applications by 15% while keeping directory cache size or to maintain system performance while using directory caches 8 times smaller.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); B.3.2 [**Memory Structures**]: Design Styles

General Terms

Management, Design, Experimentation, Performance

Keywords

Multiprocessor, cache coherence, directory cache, operating system, coherence deactivation, private block, efficiency

1. INTRODUCTION AND MOTIVATION

Over the last years, there has been an increasing demand for larger and more powerful high-performance shared-memory multiprocessors [7, 15, 23]. In these architectures, processors accelerate their accesses to memory by using one or more levels of private caches, which are made transparent to software by means of a cache coherence protocol.

Most shared-memory multiprocessors implement cache coherence protocols based on directories since they represent the most scalable approach. Traditional directories keep track of all memory blocks in the system, which allows processors to easily identify the cached blocks without generating large quantities of coherence traffic, as it happens in broadcast-based protocols. However, keeping track of all the blocks in main memory entails huge storage requirements. To avoid it, some recent proposals [20] and commodity systems, such as the current AMD Magny-Cours [7], only keep track of cached memory blocks. Thus, the directory information is only kept in small directory caches [22, 12]. Due to the lack of a full directory, the eviction of a directory cache entry entails the invalidation of all the cached copies of the associated block. Since systems are increasingly larger and the size of directory caches is quite limited, directory caches can suffer frequent evictions and, consequently, they may exhibit high miss rates (up to 70% as reported in some recent studies [20, 10]). As a result, the miss rate of processor caches may become excessively high, which results in serious performance degradation.

Enlarging the size of directory caches is not a reasonable solution because this negatively impacts on both directory access latency and area requirements. Instead, we aim at increasing the effectiveness of the available space for directory caches, assuming that it will be a scarce resource, especially in large systems. To achieve this goal, our proposal is based on the fact that a significant fraction of the memory blocks used by parallel applications are private, that is, accessed by just one processor [13]. Figure 1 shows the fraction of private and shared blocks for a wide range of parallel applications. According to this graph, most of the accessed blocks (about 75% on average) are private. Although private blocks do not require coherence maintenance, directory caches keep track of them. As a consequence, most of the information that they keep is unnecessary, which seriously jeopardizes their effectiveness. On the contrary, if they did

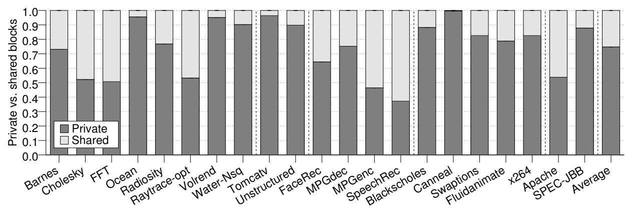

Figure 1: Fraction of private versus shared blocks.

not track private blocks, the availability of directory entries for the blocks that really need coherence (i.e., shared blocks) would increase spectacularly and their capacity could be exploited more efficiently.

Hence, in this work, we propose a mechanism that (1) classifies memory blocks into private and shared and (2) deactivates the coherence protocol for those sorted as private, which prevents directory caches from keeping track of them. Since a fine-grain block classification would require enormous storage resources, we propose to classify blocks at a page granularity, to which the operating system (OS) can help. This mechanism is as follows. By default, every new page loaded into main memory is considered as private. In turn, all blocks belonging to private pages are considered as private as well. The cache coherence protocol is deactivated for private blocks, that is, upon cache misses for private blocks, the requested data are retrieved from main memory without checking whether there are other cached copies and without keeping track of them in the directory cache. The OS detects when a private page must become shared. When this happens, the OS triggers a coherence recovery mechanism, which is in charge of restoring the coherence for all blocks within the page. From that moment on, the page is considered as shared and, in turn, all its blocks are considered as shared too, which will enforce their tracking.

This scheme only requires minor modifications in the OS and the memory controllers. Furthermore, it does not require dedicated hardware structures because it takes advantage of those already used by the OS and processors: *Translation Lookaside Buffers* (TLBs), page tables, and *Miss Status Holding Registers* (MSHRs).

We evaluate the impact of our proposal by simulating its implementation in a system similar to the AMD Magny-Cours processor. Simulation results show that, thanks to our proposal, directory caches can omit the tracking of about 57% (on average) of the accessed memory blocks. By not storing the directory information for those blocks, the number of evictions in directory caches and, therefore, the number of invalidations sent by memory controllers decreases about 70%. This results in reductions in the miss rate of processor caches of about 35%, which is translated into performance improvements of 15% (on average). On average, dynamic energy consumption is also reduced mainly due to the elimination of accesses to both directory caches (5%) and memory (20%) and the reduction in coherence traffic (15%). Additionally, this approach can be used to maintain performance while using smaller directory caches. In particular, simulations show that a system implementing our proposal can perform similarly to a system in which all cached blocks are tracked by directory caches eight times larger.

The rest of the paper is organized as follows. Section 2 discusses the related work. We present our proposal in Section 3. Section 4 focuses on the particular case of Magny-Cours processors. We describe the simulation environment in Section 5 and present the evaluation results in Section 6. Finally, Section 7 draws some conclusions.

2. RELATED WORK

Our proposal is based on the observation that most of the blocks referred to by parallel applications are private. We take advantage of this fact to propose a hybrid hardware-OS mechanism that avoids the tracking of those private blocks. In this section, we comment on some works that are somehow related to our proposal.

Like our approach, some authors use the OS to detect private and shared pages. Recently, Hardavellas et al. [13] used this detection to propose an efficient data placement policy for distributed shared caches (NUCA). While the mechanism for detecting private and shared pages is similar to ours, its application is completely different (data placement) and it does not consider coherence aspects. In contrast, we focus on how the detection of shared and private blocks can be used to increase directory effectiveness. On the other hand, Kim et al. [14] employ OS detection to reduce the fraction of snoops in a token-based protocol. That work is based on the fact that, although most referred blocks are private, the small fraction of shared blocks accounts for the majority of the cache misses. Hence, they propose a sophisticated mechanism that detects the shared blocks and their sharing degree so that broadcast messages can be replaced by multicast ones. Unfortunately, this proposal requires extra hardware (considerably larger TLBs) and adds OS overhead in order to be able to obtain significant improvements. Differently, our mechanism is much simpler and does not require complex hardware/OS modifications. Furthermore, we target the fraction of private blocks instead of the fraction of cache misses for shared blocks.

Our proposal can be used to reduce the number of directory entries while maintaining system performance. Some proposals achieve similar reductions by combining several directory entries into a single one as proposed in [24]. However, these proposals are orthogonal to ours and they can be used simultaneously.

Some works remove the unnecessary traffic of broadcast-based protocols by performing coarse-grain tracking of blocks at the expense of increasing the storage requirements. Moshovos et al. [21] and Cantin et al. [6] proposed RegionScout filters and Region Coherence Arrays, respectively, which provide different trade-offs between accuracy and implementation costs. Whereas RegionScout filters have lower storage requirements and they are less complex than Region Coherence Arrays, the latter are more accurate identifying shared regions and filter more unnecessary broadcast traffic. In turn, RegionTracker [26] provides a framework for coarse-grain optimizations that reduces the storage overhead and eliminates the imprecision of previous proposals. However, it requires considerable modifications in the cache design to facilitate region-level lookups. All these techniques share with the ours the idea of deactivating the coherence mechanism when it is not indispensable. However, there are two major differences. First, our proposal is aided by the OS, which significantly reduces the hardware overhead and complexity. Second, we do not aim at reducing broadcast traffic, but at avoiding to allocate in a directory cache data blocks that do not require coherence maintenance.

Similarly to our proposal, other works take advantage of

OS structures. Ekman et al. [8] propose a snoop-energy reduction technique for CMPs. This technique keeps a sharing vector within each TLB entry indicating which processors share a page. This sharing vector is broadcast on each snoop request and prevents processors not sharing the page from carrying out a tag-lookup in their caches. In turn, Enright-Jerger et al. [9] extend the region tracking structure proposed by Zebchuk et al. [26] to keep track of the current set of sharers of a region. Unfortunately, these techniques increase the storage requirements and entail important hardware modifications, which make them difficult to be implemented in real systems. Furthermore, our technique does not intend to keep the track of the sharers of a page, but it only maintains information about whether the page is shared or not (1 bit) and simply deactivates the tracking of blocks for private pages.

Other works also support cache coherence by means of a combination of software and hardware. Zeffer et al. [28] proposes a trap-based architecture (TMA), which detects fine-grained coherence violations in hardware, triggers a coherence trap when one occurs, and maintains coherence by software in coherence trap handlers. Like our mechanism, the trap-based architecture assumes a bit in the TLB and relies on the OS to detect when a private page moves to the shared state. However, in TMA, traps are associated with coherence violations in load/store operations, contrary to our mechanism, where they are associated with TLB misses. Additionally, TMA requires extra hardware support into each processor core to speed up the coherence trap handling. Alternatively, they propose a simple hardware mechanism that implements the inter-node coherence protocol in software [27]. To do this, two hardware modifications are required. First, the inter-node coherence has to check the need for invoking the software-coherence protocol. Second, the memory controller must handle dirty remote data that are evicted from the last level of cache. In this case, the hardware overhead is low, but opposite to our proposal, the software overhead is quite high.

Finally, Fensch et al. [11] propose a coherence protocol that does not require hardware support to enforce cache coherence. Rather, it avoids the possibility of incoherence by not allowing multiple writable shared copies of pages. Data are mapped to processor's caches at the granularity of pages under OS control and remote cache accesses are supported by hardware. However, that proposal requires release consistency, introduces extra overhead regarding hardwired systems, and is only suitable for CMPs due to the severe penalty caused by the remote cache access support.

3. COHERENCE DEACTIVATION

Directory caches keep track of cached memory blocks to avoid inconsistencies among the different copies of them. Although they indiscriminately track all cached blocks, a significant fraction of them cannot suffer from inconsistencies because they are private (i.e., accessed just by one processor). Therefore, the unnecessary use of the directory caches for private blocks decreases their effectiveness.

In order to address this problem, we propose a technique that, with the aid of the OS, dynamically detects private blocks and deactivates the coherence for them. Since a fine-grain detection (e.g., block-granularity) requires a huge amount of hardware resources, our proposal is based on a coarse-grain strategy (page granularity).

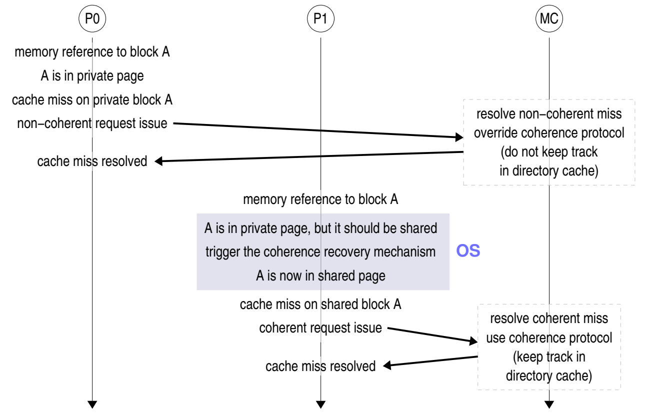
P0
P1
MC
memory reference to block A
A is in private page
cache miss on private block A
non−coherent request issue
resolve non−coherent miss
override coherence protocol
(do not keep track
in directory cache)
cache miss resolved
memory reference to block A
A is in private page, but it should be shared
trigger the coherence recovery mechanism
A is now in shared page
OS
cache miss on shared block A
coherent request issue
resolve coherent miss
use coherence protocol
(keep track in
directory cache)
cache miss resolved

Figure 2: Overview of the proposed mechanism. *P0* and *P1* are processors and *MC* is the memory controller. The shaded background indicates that the OS is in charge at that moment.

The general idea is that, by default, every new page loaded into main memory is considered as private. Cache misses for blocks belonging to private pages override the coherence protocol. As a result, directory caches do not keep track of them. When the OS detects that two different processors try to access blocks within the same private page, it triggers a hardware coherence recovery mechanism that restores the coherence state for every block within the private page and converts it into shared. From that moment on, the page will be considered as shared and the memory accesses to its blocks are resolved according to the coherence protocol, which entails that directory caches will keep track of them.

Figure 2 outlines our proposal. First, *P0* references the memory block *A*, which causes a cache miss. Assuming that *A* belongs to a private page, *P0* issues a non-coherent request, which is served by the home node (i.e., the memory controller or node where the memory block is mapped to) without storing any coherence information in its directory cache. Later, *P1* references the same memory block *A* and a TLB miss happens. While the OS is handling the TLB miss, it realizes that the page should become shared since another processor has already accessed it. Consequently, it triggers the coherence recovery mechanism. When the recovery process finishes, the page becomes shared and the access to the cache proceeds, resulting in a miss. Since the referred block belongs to a shared page, a coherent request is issued, which is processed as the assumed cache coherence protocol establishes.

Next sections explain our proposal in detail walking through different key aspects such as the generation and service of non-coherent requests (Section 3.1), the detection of shared pages (Section 3.2), and the coherence recovery mechanism that restores the coherence state (Section 3.3).

3.1 Non-Coherent Requests

On memory references, processors first access their TLB to translate virtual addresses into physical addresses. As shown in Figure 3, each TLB entry is made up of two components: the tag, which basically comprises the virtual address of the page, and the data, which contain the corresponding physical address along with several properties associated to the translation. Since the TLB entry data field often contains some reserved bits that are not used [4], we take advantage of

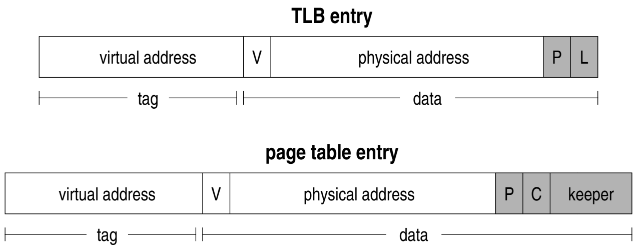

Figure 3: TLB and page table entry format. Shaded fields are additional fields required by our proposal. V is the valid bit, P is the private bit, L is the locked bit, and C is the cached-in-TLB bit.

two of them to include two new fields: the *private* bit (P), which is used to differentiate between private and shared pages, and the *locked* bit (L), which is used to avoid undesirable race conditions (as explained later in Section 3.3).

The P bit is only taken into account when a memory reference to a block belonging to the translated page causes a cache miss. Hence, if the cache miss is for a block belonging to a private page, a *non-coherent* request is issued. Otherwise, a *coherent* request is issued. Non-coherent requests override the coherence protocol and are always served by main memory. In addition, directory caches do not track them. This behaviour has two primary advantages. First, neither a lookup nor an insertion in the directory cache is required, which helps to reduce the latency of cache misses, the contention at memory controllers, and the energy consumption. Second, directory caches are less occupied and, therefore, they do a better use of their capacity to track blocks that really need coherence. Notice that, to instruct memory controllers to understand non-coherent requests, only minor modifications in their microcode will be required, but not hardware modifications.

3.2 Detection of Shared Pages

Similarly to other works [13, 14], we take advantage of OS capabilities to distinguish between private and shared pages. To accomplish this task, page table entries need three additional fields, as Figure 3 shows. The *private* bit (P) indicates whether the page is private or shared. If P is set, the *keeper* field will contain the identity of the first and single processor caching the page table entry in its TLB. The *cached-in-TLB* bit (C) indicates whether the keeper field is valid or not. Notice that these extra fields do not require dedicated hardware, but only extra OS storage requirements, which are very small. Particularly, the size of the extra fields is $2 + log_2(N)$ bits, where N is the number of processors in the system. Thus, assuming a system comprised of 8 processors, like the AMD Magny-Cours, only 5 extra bits per entry would be required.

On a page table fault, the OS allocates a new page table entry with the virtual to physical address translation. In addition, since every newly loaded page is considered as private, the P bit is set and C is cleared indicating that the entry is not cached in any TLB yet. When a TLB miss takes place, a search in the page table will be performed. In addition to caching the corresponding page table entry in the TLB, the new fields of the page table may be updated as indicated by the Algorithm 1. According to it, if C is clear, it means that the page table entry has not been cached yet and, therefore, the page is private. In this case, C is set and

Algorithm 1 Actions to perform in view of a TLB miss

```
if $C$ is clear then
    $C \leftarrow$ set
    $keeper \leftarrow$ requester
else
    if $P$ is set then
        if $keeper \neq requester$ then
            trigger coherence recovery mechanism
            $P \leftarrow clear$
        end if
    end if
end if
```

the identity of the processor requesting the page table entry is kept in the *keeper* field. If both C and P bits are set, it is necessary to check whether the page should be converted into shared. Thus, if the *keeper* field matches the requester, it means that the keeper processor suffered a TLB eviction and it requires such information again. Since only one processor is still accessing the page, the page continues to be private and no changes are necessary. On the contrary, if the *keeper* field does not match the requester, two different processors are trying to access blocks within the same private page and, consequently, the page must become shared. In this case, the coherence recovery mechanism is triggered and, when it finishes, P is cleared. Finally, if P is clear, it means that the page is shared and no modifications are required. Notice that, to avoid races, this algorithm is executed by the OS inside a critical section during the resolution of TLB misses.

Figure 4 depicts all the actions that can take place on memory operations.

3.3 Coherence Recovery Mechanism

From the point of view of directory caches, the main difference between the blocks considered as private and those considered as shared is that, whereas private blocks are not tracked, shared ones are. Therefore, when a private page must become shared, the recovery mechanism will have to ensure that the corresponding directory cache keeps proper track of each block within the page to recover. Notice that only the keeper may have valid copies of those page blocks since, until then, the page was private. To recover the coherent status of the corresponding blocks, we propose two different strategies: (1) evicting blocks from the keeper's cache (flushing-based recovery) or (2) updating the directory cache so that it keeps track of currently cached blocks (updating-based recovery). Next two sections explain these mechanisms in detail and the main differences between them.

3.3.1 Flushing-based Recovery Mechanism

The simplest way to restore the coherence status of the blocks belonging to a page that has been detected as shared is by flushing all the page blocks. This flushing-based recovery mechanism is as follows.

First, the initiator (node that triggers the coherence recovery mechanism) issues a *recovery request* for the involved page to its keeper (obtained from the page table).

Second, on the recovery request arrival, the keeper performs three operations. It firstly locks the corresponding TLB entry by setting the L bit. This prevents the keeper from issuing new requests for any of the page blocks. In

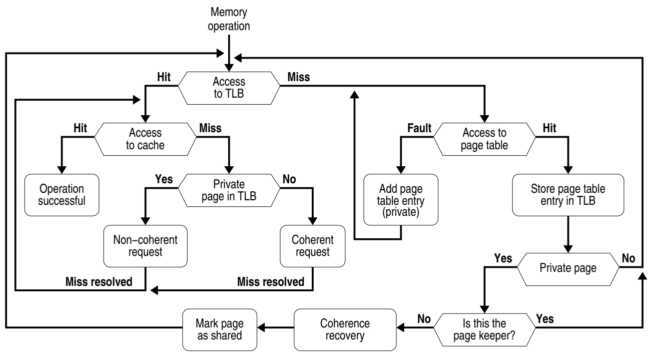

Figure 4: Block diagram of the general working scheme.

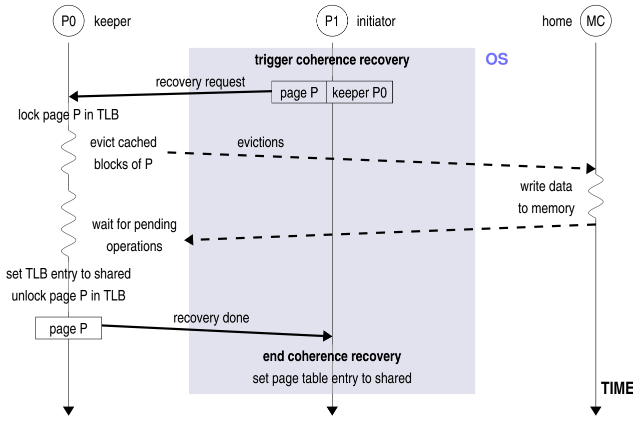

Figure 5: Flushing-based recovery mechanism. *P0* **and** *P1* **are processors and** *MC* **is the home node. Solid arrows are messages due to the recovery mechanism, whereas dashed arrows are messages due to the coherence protocol.**

case the TLB entry is not present, new operations for blocks within that page cannot be issued because it does not have the required information in its TLB and it cannot access the page table entry since the initiator is accessing it inside a critical section. Therefore, it would not be necessary to lock it. Secondly, the keeper performs a cache lookup and flushes every cached block within the involved page, writing-back data to main memory in case they have been modified. Thirdly, the keeper checks its MSHR structure to see the pending operations. While there is at least one pending miss or eviction for any of the page blocks, the keeper must wait for its completion. Once this happens, the involved blocks are evicted. After this, the keeper marks the corresponding TLB entry as shared, unlocks it, and informs the initiator by means of a *recovery done* message.

Third, when the initiator receives the recovery done message, the recovery mechanism finalizes and the page is set as shared in both the page table and the local TLB entry. Notice that, during this process, the OS has exclusive access to the involved page table entry and no other processor can access it so that race conditions cannot take place. Figure 5 illustrates an example of how this mechanism works.

After completing the flushing-based recovery mechanism for a page, we know for sure that the blocks belonging to such a page are not cached. Thus, from that moment on, since the page is marked as shared, every time a cache miss for any of its blocks happens, a coherent request will be issued and the directory cache will be able to keep proper track of them.

3.3.2 Updating-based Recovery Mechanism

The main advantage of the flushing-based recovery is that its implementation in real systems is quite feasible and straightforward, since the flushing operation is already supported by current systems. Unfortunately, the main drawback of this approach is that the flushing of all the blocks within a page may increase the miss rate of processor caches, which could degrade system performance (we raise this aspect in Section 6). To address this potential drawback, we propose an alternative implementation based on updating the directory cache information. This mechanism is as follows.

First, the initiator issues a *recovery request* to the corresponding page keeper.

Second, like in the flushing-based recovery mechanism, the page keeper locks the corresponding TLB entry on the arrival of the recovery request and looks for the blocks within the page that are present in its cache. The addresses of the found blocks are coded in a bit vector, which is included in a *recovery response* that will be sent to the home memory controller. To easily code the addresses of the cached blocks, a bit vector which contains as many bits as blocks within a page is used (e.g., 64 bits for systems using 4K pages with 64 bytes blocks). Thus, if the first bit in the vector is set, it means that the first block of the page is cached; if the second bit is set, the second block of the page is cached; and so on. After composing the bit vector and before sending the recovery response, the keeper checks whether there are outstanding operations for any of the page blocks. This is done by looking up the MSHR structure. If any of the pending operations is for a block belonging to the page being recovered, the node waits until they finish. Once these operations complete, the recovery response is sent.

Third, upon the receipt of a recovery response, the home memory controller proceeds to update its directory cache according to the received bit vector. To determine the address of each cached block, the memory controller uses the page address of the block (received in the response) and the position of the bit within the bit vector. For each bit set in the vector, the memory controller creates a new entry for the corresponding block in its directory cache, evicting an old entries when required. The sharing code of the new entry can be easily set because it knows that, at that moment, there is a single copy of the block cached by the keeper. When the directory cache updating finalizes, the home node sends a *recovery target done* message back to the page keeper informing about it.

Forth, when the keeper receives the recovery target done message, it marks the TLB entry corresponding to the page as shared, unlocks the corresponding TLB entry, and sends a *recovery done* message to the initiator, finalizing the recovery process. Figure 6 shows an example of how the updating-based recovery mechanism works.

After completing the updating-based recovery mechanism for a page, we know for sure that the directory cache will hold the track of the page blocks that are cached until then. Therefore, the next time a miss on any of those blocks happens, the coherence protocol will be able to use the information in the directory cache to appropriately resolve it.

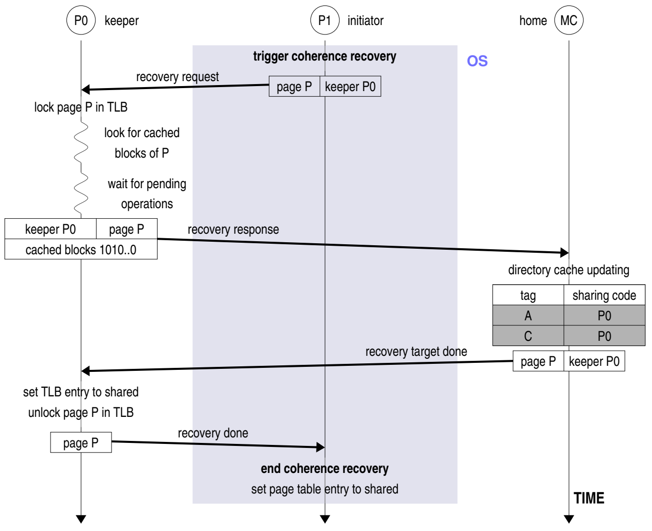

Figure 6: Updating-based recovery mechanism. *P0* **and** *P1* **are processors and** *MC* **is the home node.**

3.3.3 Discussion

In this section, we comment on some aspects of the recovery latency and the adaptation to systems with different features: page table walkers and memory interleaving.

Latency of the recovery: The recovery process may take a long time because its critical path includes a search in the keeper's cache and maybe several evictions from the keeper's cache or insertions in the directory cache. Despite its high latency, we have to take into account that the recovery process is only performed very few times. In fact, during the lifetime of a page in main memory, the page will probably have a large number of references (according to the locality principle) and it will be converted from private to shared at most once. Therefore, it is not unreasonable to expect the latency of the accesses to memory blocks to have much more impact on the overall performance than the latency of the recovery mechanism. In Section 6.3, we show that the recovery mechanism is only triggered about 3 times per 1000 cache misses (on average). Thus, the impact of the recovery mechanism on the protocol performance is indeed negligible since it is largely offset by the savings in cache misses and the reduction in their latency.

Page table walkers: Although we have linked the description of our proposal to traditional page tables, its application is also possible in systems that use hardware page table walkers. The adjustment to this context would be quite straightforward and simple. The page table will require the same fields as those assumed along this document. The single difference is that the responsibility of detecting shared pages will fall on hardware instead of the OS. Therefore, some additional extra hardware logic will be required to do it. However, since this class of system is out of the scope of this work, we have not carried out such an implementation.

Memory interleaving: The flushing-based recovery mechanism is independent of whether we assume a system with memory interleaving or not, but the updating-based recovery is slightly affected. In systems assuming memory interleaving, a page can be physically distributed across different memory controllers and, consequently, each page may have several home nodes. Therefore, the recovery responses conveying the addresses of the cached blocks will have to be sent to several home nodes. This can be easily done by using multicast messages instead of unicast ones. Thus, the destinations of a recovery response will be the set of home nodes of the blocks cached by the keeper. Furthermore, after sending the multicast recovery response and before sending the recovery done message, the keeper will have to collect as many recovery target done messages as the number of destinations in the multicast response.

Table 1: System parameters.

Memory Parameters	
Processor frequency	3.2 GHz
Cache block size	64 bytes
Processor cache	2MB (32K entries), 4-way
Processor cache access latency	2ns
Directory cache	256KB (64K entries), 4-way
Directory cache access latency	2ns
Directory cache coverage ratio	Typical 2×, worst-case 0.25×
Memory access latency (local bank)	60ns
Page size	4KB (64 blocks)
Network Parameters	
Network topology	Hypercube + extra channels
Data message size	68 and 72 bytes
Control message size	4 and 8 bytes
Network bandwidth	12.8GB/s
Inter-die link latency	2ns
Inter-processor link latency	20ns
Flit size	4 bytes
Link bandwidth	1 flit/cycle

4. TARGET SYSTEM: AMD MAGNY-COURS

We evaluate the benefits that our proposal could achieve in a system similar to AMD Magny-Cours [7]. We make this decision based on our belief that the proposed mechanism can be implemented in a commodity system without requiring great effort. To better understand the advantages that our proposal can provide, we briefly describe this system.

Magny-Cours processors comprise 2 dies, each of them with 6 processing cores which have private L1/L2 caches and share a large L3 cache (6MB). Cache coherence inside the dies is maintained by means of a broadcast-based protocol. A maximum of eight dies (four Magny-Cours processors) can be included in a board to compound a larger system. These processor dies are made coherent by using a directory-based cache coherence protocol that implements MOESI states. Directory information is stored in directory caches called *HT Assists* (HTAs) or *Probe Filters* and there is not a directory backup in memory. Each HTA is associated with a memory controller and it holds an entry for every block cached in the system that maps to its memory bank. The sharing code field of the HTA comprises just one pointer to the owner node (3 bits). Upon a miss on the HTA, a new entry must be allocated, which may require to replace an existing one. Due to the lack of a directory backup, before performing the replacement, all the cached copies of the block identified by the replaced entry must be invalidated.

Typically, each HTA has 256K entries and each die has 128K entries in its cache hierarchy. Therefore, the coverage ratio of the directory caches is 2×, i.e., there are at least twice as many directory entries in the system as blocks can be cached[1] [7]. This would be enough for tracking all the cached blocks if they were distributed uniformly among the HTAs. However, it may happen that some memory con-

[1]In fact, Gupta in [12] recommends a factor of 2 or 4.

Table 2: **Benchmarks and input sizes.**

Benchmarks	Input size
SPLASH 2 (8)	
Barnes	8192 bodies, 4 time steps
Cholesky	Input file tk15.O
FFT	64K complex doubles
Ocean	258 × 258 ocean
Radiosity	room, -ae 5000.0 -en 0.050 -bf 0.10
Raytrace-opt	Teapot
Volrend	Head
Waternsq	512 molecules, 4 time steps
Scientific benchmarks (2)	
Tomcatv	256 points, 5 time steps
Unstructured	Mesh.2K, 5 time steps
ALPBench (4)	
FaceRec	Script
MPGdec	525_tens_040.m2v
MPGenc	Output of MPGdec
SpeechRec	Script
PARSEC (4)	
Blackscholes	simmedium
Canneal	simmedium
Fluidanimate	simmedium
Swaptions	simmedium
x264	simsmall
Commercial Workloads (2)	
Apache	1000 HTTP transactions
SPEC-JBB	1600 transactions

trollers hold more cached blocks than others. The worst-case scenario appears when all the cached blocks belong to the same memory controller (known as hotspotting). In Magny-Cours, having up to 8 dies, the coverage ratio in this worst-case dramatically decreases down to 0.25×, which could cause a lot of replacements and, therefore, the invalidation of a large quantity of cached blocks, thereby underutilizing caches and leading to a significant performance degradation.

In Magny-Cours, memory blocks are distributed among memory banks without performing interleaving. This has two important consequences. First, all the blocks within a page map to the same memory bank and, therefore, to the same directory cache. Second, the coverage ratio can be dramatically reduced, especially for parallel applications, because they may allocate most of the accessed data in the node where the main thread of the application is executed, leading to hotspotting.

5. EVALUATION METHODOLOGY

We evaluate our proposals with full-system simulation using Virtutech Simics [18] running Solaris 10 and extended with the Wisconsin GEMS toolset [19], which enables detailed simulation of multiprocessor systems. For modeling the interconnection network, we have used GARNET [1], a detailed network simulator included in GEMS. Finally, we have also used the McPAT tool [17], assuming a 45nm process technology, to measure the savings in terms of energy consumption that our proposal can entail.

For the evaluation of our proposals, we have first modeled a cache coherent HyperTransport system optimized with directory caches (HTAs) similar to those of the AMD Magny-Cours. We simulate eight dies, which constitutes the maximum number of nodes supported by the Magny-Cours protocol. Although each Magny-Cours die has actually six cores, we are only able to simulate two of them due to time constraints. Moreover, since this paper does not focus on the intra-die broadcast-based coherence protocol and taking into

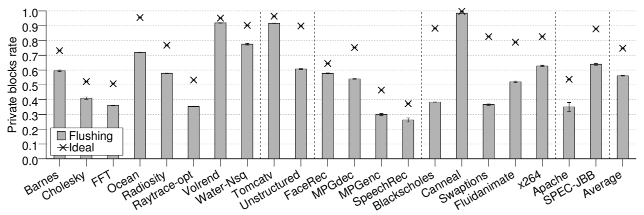

Figure 7: Fraction of actual private blocks versus detected private blocks.

account that such a protocol would considerably increase the simulation time, we do not model it. We consider the described system as the *base* architecture and its main parameters are shown in Table 1. Our proposals are implemented upon this system.

We evaluate our proposal with a wide variety (21) of parallel workloads from 3 suites (SPLASH-2 [25], ALPBenchs [16], and PARSEC [5]), two scientific benchmarks, and two commercial workloads [2], which are shown in Table 2. Due to time requirements, we are not able to simulate these benchmarks with large working sets. Consequently, as done in most works [6, 10, 11], we have simulated the applications assuming smaller data-sets. To avoid altering the results, we have reduced the size of both processor caches and directory caches accordingly. Particularly, our caches are four times smaller than the ones used by the original Magny-Cours processors. Notice that, since the size of all the simulated caches are proportionally reduced, the coverage ratio of directory caches is the same as in Magny-Cours (2×).

All the reported experimental results correspond to the parallel phase of benchmarks. We account for the variability in multi-threaded workloads [3] by doing multiple simulation runs for each benchmark and injecting small random perturbations in the timing of the memory system.

6. PERFORMANCE EVALUATION

In this section, we show how our proposal is able to reduce the amount of blocks tracked by directory caches. This results in energy saving and less processor cache misses, which leads to performance improvements. We also study how, thanks to our proposal, it is possible to maintain performance while dramatically reducing the directory cache size.

6.1 Private Blocks

Our proposal is based on the fact that a significant amount of the memory blocks accessed during the execution of parallel applications is private. Crosses in Figure 7 show the fraction of memory blocks referred to by processors that are only used by one of them (i.e., private). As observed, about 75% (on average) of the referred blocks are private. Our mechanism tries to identify such private blocks. However, since it works at a page granularity (instead of at block level), not all the private blocks can be identified as such because, when a page contains both private and shared blocks or just private blocks accessed by different processors, our mechanism will consider the whole page as shared and, in turn, all the blocks within that page will be considered as shared too. Thus, bars in Figure 7 show the fraction of blocks that our mechanism detects as private. Comparing the fraction of actual private blocks (75% on average) to the fraction of detected private blocks (57% on average), we can

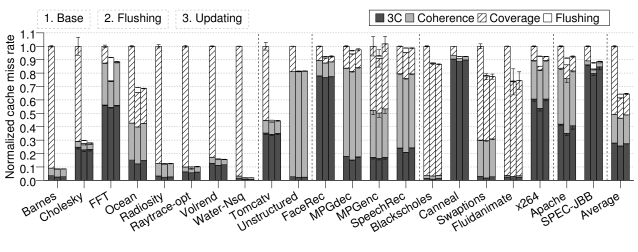

Figure 8: Normalized cache miss rate classification.

see that our proposal obtains a good approximation since only 18% of the blocks are misclassified. According to these data, a detection based on pages provides a good trade-off since it does not entail large overhead and it allows processors to detect most of the private blocks. Hence, Figure 7 shows that directory caches omit the tracking of 57% of the referred blocks.

6.2 Processor Cache Misses

Since directory caches do not track blocks belonging to private pages, less blocks contend for their entries. Consequently, they suffer fewer evictions and, therefore, less blocks are invalidated from processor caches. As a result, the processor cache miss rate is reduced by 35% (on average), as Figure 8 shows. This figure illustrates the fraction of the overall cache misses normalized with respect to the base system (first bar), when using coherence deactivation and the flushing-based recovery mechanism (second bar), and when using coherence deactivation and the updating-based recovery mechanism (third bar). We classify cache misses in four different groups: the *3C* misses comprise Cold, Capacity, and Conflict misses; the *Coherence* misses refer to those caused by invalidations due to write requests (store operations issued by other processors); the *Coverage* misses are those caused by the invalidations issued as a consequence of evictions in the directory caches due to their limited capacity; and the *flushing* misses are due to evictions performed by the recovery mechanism. Since our proposal aims at improving the effectiveness of directory caches, it mainly acts on the coverage misses. Thus, by avoiding the tracking of private blocks, a significant part of the coverage misses can be avoided (about 75% on average). The reduction of coverage misses depends to a large extent on the quantity of misclassified blocks. In applications like barnes, cholesky, radiosity, volrend, or tomcatv there are few misclassified blocks (see Figure 7) and, therefore, the coverage misses are highly reduced. On the contrary, applications like ocean, mpgenc, swaptions, or blackscholes present a high rate of misclassified blocks, which leads to lower reductions in the number of coverage misses. The reduction of coverage misses is an important achievement because, as shown in the figure and as reported in other studies [20, 10], their rate may be really important in some scenarios (more than 90%).

The recovery mechanism is triggered once per each page converted into shared. In case of assuming the flushing-based recovery mechanism, the invalidation of cached blocks may cause additional cache misses. For instance, in applications like FFT, the use of the recovery mechanism based on flushing increases the rate of cache misses with respect to that obtained with the recovery mechanism based on updating. This happens because the pages that are converted from private to shared have a lot of cached blocks at the moment of the conversion. Therefore, all those blocks will be invalidated and, when they are accessed again, new cache misses will be generated. Notice though that this is not the common situation. In fact, on average, the number of flushing misses is insignificant and their increase is largely offset by the reduction in 3C, Coherence, and Coverage misses. Thus, contrary to what might be thought, the use of the flushing-based recovery mechanism does not entail an increase of the total number of cache misses with respect to that of the updating-based implementation.

The reduction of both directory evictions and cache misses has a meaningful impact on the overall network traffic, as depicted in Figure 9. This figure shows the network traffic generated by the assumed cache coherence protocol using our proposal normalized to the base system. Each bar plots the number of flits transmitted through the interconnection network. Notice that those data include the traffic due to the coherence recovery mechanisms, but it is really insignificant (lower than 0.1% on average). As shown, the reduction in the number of cache misses is so impressive that the coherence traffic reduces drastically (about 40% on average). Besides, given that the cache miss rate is more or less similar regardless of the used coherence recovery mechanism, the reduction in network traffic does not change according to the recovery policy.

Our proposal is not only able to reduce the amount of cache misses, but also their average completion latency, as Figure 10 depicts. The latency of cache misses is split into 4 stages: *request* latency refers to the time elapsed from the beginning of the cache miss until the receipt of the corresponding request by the home memory controller; *waiting* is the time that requests remain in the home memory controller waiting for the beginning of their service; *memory* is the latency of the memory controller (which also includes the latency of the directory cache) in obtaining the requested data from memory; finally, *response* is the latency from either the sending of the memory response or the forwarding of the request to another processor until the completion of the miss. Since requests for private blocks do not need a directory cache lookup, non-coherent requests can be served more quickly and, therefore, the average latency of cache misses is lowered. Thus, the average cache miss latency is reduced by

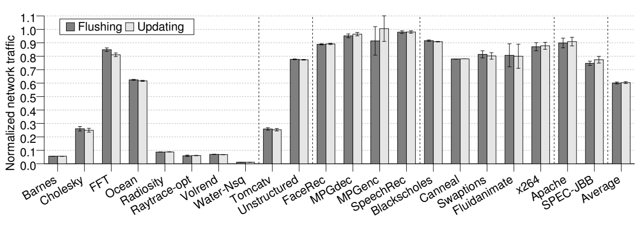

Figure 9: Normalized network traffic (in flits).

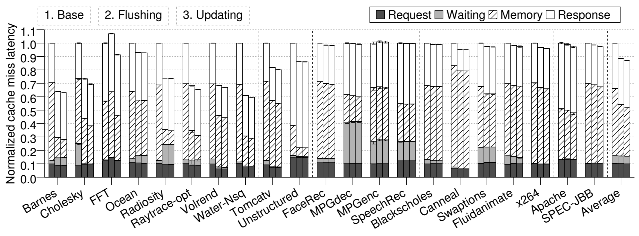

Figure 10: Normalized latency of solving cache misses.

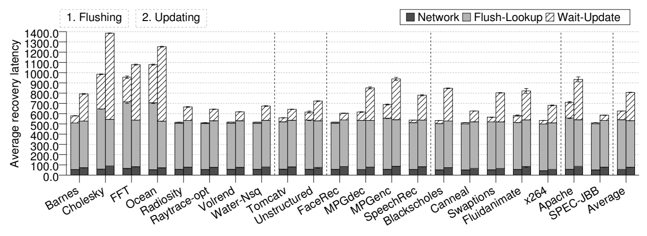

Figure 11: Average latency of the coherence recovery mechanisms (in clock cycles).

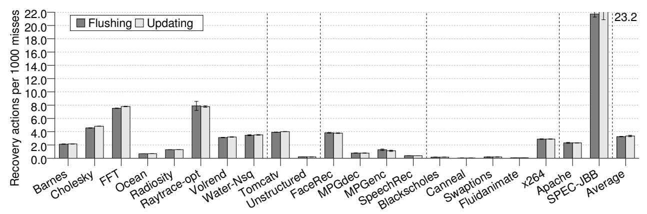

Figure 12: Times that the coherence recovery mechanism is triggered per 1000 cache misses.

about 13% on average when assuming the updating-based recovery mechanism and by 10% when using the flushing-based recovery. Notice that this reduction is reached mainly thanks to avoid doing a directory cache lookup and, besides, is linked to the proportion of cache misses treated as non-coherent. Consequently, the more cache misses for private blocks, the higher the reduction in the average latency of cache misses. On the other hand, for the FFT application and assuming the flushing-based recovery, the average latency of cache misses increases slightly with respect to that of the base system. This mainly happens because in that case there are a lot of flushing misses, which are treated as misses for shared blocks.

6.3 Coherence Recovery Mechanism

Figure 11 shows the average latency of the proposed coherence recovery mechanisms according to the timing parameters shown in Table 1. We have split their latency into several components: *network* is the transmission latency of the generated messages; *flush* is the latency required to perform the flushing of the corresponding page; *lookup* is the latency of looking up the cached blocks belonging to a page; *wait* is the time that nodes must wait until the evictions finish; and *update* is the latency of updating the directory cache. Thus, whereas the latency of the flushing-based mechanism is made up of *network*, *flush*, and *wait* latencies, the one of the updating-based mechanism is composed of *network*, *lookup*, and *update* latencies. Notice that the latency of the recovery mechanisms may be considerable because it includes a search in the cache of the page keeper. Besides, the latency of the updating-based mechanism also includes several accesses to the directory cache to add the required entries. As Figure 11 shows, the *network* and *flush/lookup* latencies are quite similar for both mechanisms. However, the *update* latency is appreciably higher than the *wait* latency. This is due to two reasons. First, in the flushing-based recovery mechanism, evictions are overlapped with the cache lookup, whereas in the updating-based mechanism, the bit-vector is not sent to the home memory controller until the lookup finishes. Second, the insertion of new directory cache entries (updating) may cause the eviction of other directory

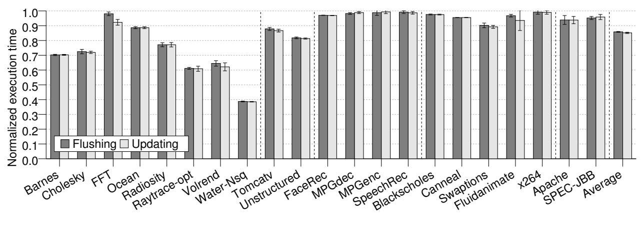

Figure 13: Normalized runtime maintaining the directory cache size.

cache entries, which may entail extra latency. This latency is specially high in Cholesky, Ocean, and FFT, because in Magny-Cours a controller can have at most 32 outstanding transactions (due to the limited number of tags to identify coherence transactions [7]). When a controller reaches the maximum number of simultaneous transactions, the new ones will have to wait for the completion of those already outstanding. Thus, when the recovery mechanism based on updating is working on a page with a lot of blocks cached in the keeper and the updating entails several evictions of directory cache entries, it may occur that the memory controller runs out of free tags and reaches the maximum number of outstanding transactions, which will slow down the updating. This is the reason why the *wait* latency is particularly high for Cholesky, Ocean, and FFT.

Although the recovery mechanism can exhibit a considerable latency, it is not often used. To illustrate this, we have estimated the number of times that the coherence recovery mechanism is triggered with respect to the total number of cache misses. As shown in Figure 12, on average, the coherence recovery mechanism is only triggered about 3 times per 1000 cache misses (up to 23 for the SPEC-JBB application). As a result, its impact on the whole runtime of applications is almost unnoticeable (less than 1% on average).

6.4 Execution Time

Due to the reduction in the number of cache misses and their latency, the runtime of applications can be significantly reduced, as depicted in Figure 13. This figure also illustrates that the coherence recovery mechanism has little impact on the overall performance since the results for the two evaluated mechanisms are almost identical. Thus, according to these results, the proposed technique can lead to improvements in application runtime of about 15% on average.

The systems where the storage requirements are critical can also take advantage of our proposal because, by means of it, the size of directory caches can be drastically reduced while obtaining good performance. This is illustrated in Figure 14 by the bars labeled as *DC:2*, *DC:4*, and *DC:8*, which represent three configurations with a half, a fourth, and an eighth of the base cache size, respectively. Since the results are quite similar for both recovery mechanisms, we only show the results for the mechanism based on flushing. We can see that our proposal allows to reduce the size of directory caches up to eight times while still maintaining the execution time of applications similar (on average) to that of the base system.

6.5 Energy Consumption

Thanks to the reduction in cache misses and network traffic, our proposal is also able to reduce system energy consumption. Figure 15 shows the dynamic energy consump-

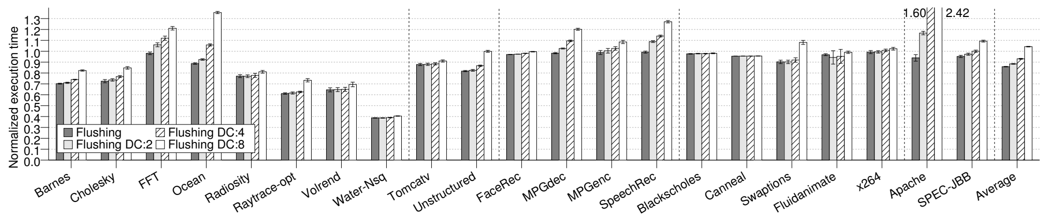

Figure 14: Execution time normalized to the base system. *DC:2*, *DC:4*, **and** *DC:8* **stand for directory caches with their size divided by 2, 4, and 8, respectively.**

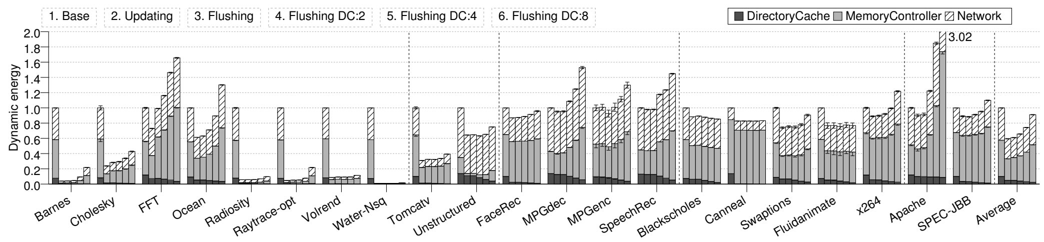

Figure 15: Dynamic energy consumption normalized to the base system. *DC:2*, *DC:4*, **and** *DC:8* **stand for directory caches with their size divided by 2, 4, and 8, respectively.**

tion of directory caches, memory controllers, and the interconnection network. Since non-coherent requests do not need to access directory caches, their consumption is reduced, which is shown by the *updating* and *flushing* bars. Furthermore, this reduction becomes more significant as the directory cache size decreases because, despite the fact that smaller directory caches suffer more accesses, their access latency is lower, which offsets such an increase of accesses.

Although our proposal decreases the number of memory accesses (due to the cache miss reduction, as shown in Figure 8), the flushing recovery mechanism may increase it (due to the eviction of cached blocks). On average, the referred reduction offsets this increase. As a result, the energy consumption of memory controllers is reduced by 20%, as the *flushing* bar shows. Notice that, for the FFT application, the number of flushed blocks is noticeable and, therefore, the energy consumption of memory controllers increases slightly. However, when the updating recovery mechanism is used (*updating* bar), since it does not cause additional cache misses, the energy consumption of memory controllers is reduced. On the other hand, as directory caches become smaller, the number of caches misses increases and, consequently, more accesses to memory controllers will be required, thereby increasing its consumption.

Finally, our proposal also entails savings in the energy consumption of the interconnection network due to the reduction in network traffic, as shown in Figure 9. Taking into account the overall consumption of directory caches, memory controllers, and the interconnection network, we can see that both *updating* and *flushing* are able to reduce the energy consumption by about 40% on average. As directory caches become smaller, the energy consumption increases mainly due to the increase in network traffic and the accesses to memory controllers. Despite this, the dynamic energy consumption of a system using our proposal remains lower than that of the base system using directory caches 8 times larger.

Regarding static energy consumption (not shown in Figure 15), notice that it is really tight to the execution time of applications. In particular, the reduction in static energy consumption of memory controllers and the network is directly proportional to the reduction in runtime (Figures 13 and 14). With respect to directory caches, their static energy reduction depends on both the application runtime and their size. Thus, when using directory caches 2, 4, and 8 times smaller than that in the base system, the static power consumption is reduced by 48%, 74%, and 86%, respectively.

7. CONCLUSIONS

In this paper we have proposed a simple approach which is able to remarkably increase the effectiveness of directory caches. It is based on the idea of avoiding the tracking of private blocks, since they do not need coherence. The OS is responsible for dynamically detecting shared blocks, which are handled as established by the applied cache coherence protocol. Avoiding the tracking of private blocks decreases the contention on directory caches. As a result, the number of blocks invalidated due to the lack of entries in directory caches can be drastically reduced (by about 57% on average). This advantage can be used not only for increasing the performance of a system (15% of reduction in application runtime), but also for continuing to obtain good performance having less storage requirements (systems with directory caches 8 times smaller).

We can also conclude that the coherence recovery mechanism does not have a significant impact on system performance and it only slightly affects energy consumption. Therefore, the flushing-based recovery mechanism seems to be the most suitable option since it offers similar performance results to the updating-based mechanism and it is much simpler to implement, being suitable for future commercial processors.

8. ACKNOWLEDGMENTS

This work has been supported by Generalitat Valenciana under Grant PROMETEO/2008/060, by Spanish Ministry of Ciencia e Innovación under grant "TIN2009-14475-C04-01", and by European Commission FEDER funds under grant "Consolider Ingenio-2010 CSD2006-00046".

9. REFERENCES

[1] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A detailed on-chip network model inside a full-system simulator. In *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 33–42, Apr. 2009.

[2] A. R. Alameldeen, C. J. Mauer, M. Xu, P. J. Harper, M. M. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Evaluating non-deterministic multi-threaded commercial workloads. In *5th Workshop On Computer Architecture Evaluation using Commercial Workloads (CAECW)*, pages 30–38, Feb. 2002.

[3] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *9th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 7–18, Feb. 2003.

[4] AMD. AMD64 architecture programmer's manual volume 2: System programming. Whitepaper, June 2010.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, Oct. 2008.

[6] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *32th Int'l Symp. on Computer Architecture (ISCA)*, pages 246–257, June 2005.

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

[8] M. Ekman, F. Dahlgren, and P. Stenström. TLB and snoop energy-reduction using virtual caches. In *Int'l Symp. on Low Power Electronics and Design (ISLPED)*, pages 243–246, Aug. 2002.

[9] N. D. Enright-Jerger, L.-S. Peh, and M. H. Lipasti. Virtual circuit tree multicasting: A case for on-chip hardware multicast support. In *35th Int'l Symp. on Computer Architecture (ISCA)*, pages 229–240, June 2008.

[10] N. D. Enright-Jerger, L.-S. Peh, and M. H. Lipasti. Virtual tree coherence: Leveraging regions and in-network multicast tree for scalable cache coherence. In *41th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 35–46, Nov. 2008.

[11] C. Fensch and M. Cintra. An OS-based alternative to full hardware coherence on tiled CMPs. In *14th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 355–366, Feb. 2008.

[12] A. Gupta, W.-D. Weber, and T. C. Mowry. Reducing memory traffic requirements for scalable directory-based cache coherence schemes. In *Int'l Conference on Parallel Processing (ICPP)*, pages 312–321, Aug. 1990.

[13] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *36th Int'l Symp. on Computer Architecture (ISCA)*, pages 184–195, June 2009.

[14] D. Kim, J. Ahn, J. Kim, and J. Huh. Subspace snooping: Filtering snoops with operating system suport. In *19th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, Sept. 2010.

[15] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, Nov. 2007.

[16] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. In *Int'l Symp. on Workload Characterization*, pages 34–45, Oct. 2005.

[17] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *42nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 469–480, Dec. 2009.

[18] P. S. Magnusson, M. Christensson, and J. Eskilson, et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[19] M. M. Martin, D. J. Sorin, and B. M. Beckmann, et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 33(4):92–99, Sept. 2005.

[20] M. R. Marty and M. D. Hill. Virtual hierarchies to support server consolidation. In *34th Int'l Symp. on Computer Architecture (ISCA)*, pages 46–56, June 2007.

[21] A. Moshovos. RegionScout: Exploiting coarse grain sharing in snoop-based coherence. In *32nd Int'l Symp. on Computer Architecture (ISCA)*, pages 234–245, June 2005.

[22] B. W. O'Krafka and A. R. Newton. An empirical evaluation of two memory-efficient directory methods. In *17th Int'l Symp. on Computer Architecture (ISCA)*, pages 138–147, June 1990.

[23] M. Shah, J. Barreh, and J. Brooks, et al. UltraSPARC T2: A highly-threaded, power-efficient, SPARC SoC. In *IEEE Asian Solid-State Circuits Conference*, pages 22–25, Nov. 2007.

[24] R. Simoni. *Cache Coherence Directories for Scalable Multiprocessors*. PhD thesis, Stanford University, 1992.

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Int'l Symp. on Computer Architecture (ISCA)*, pages 24–36, June 1995.

[26] J. Zebchuk, E. Safi, and A. Moshovos. A framework for coarse-grain optimizations in the on-chip memory hierarchy. In *40th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 314–327, Dec. 2007.

[27] H. Zeffer and E. Hagersten. A case for low-complexity MP architectures. In *ACM/IEEE Conference on Supercomputing (SC)*, pages 10–16, Nov. 2007.

[28] H. Zeffer, Z. Radović, M. Karlsson, and E. Hagersten. TMA: A trap-based memory architecture. In *20th Int'l Conference on Supercomputing (ICS)*, pages 259–268, June 2006.