Increasing the Effectiveness of Directory Caches by Avoiding the Tracking of Non-Coherent Memory Blocks

Blas Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José Duato

Abstract—A key aspect of the design of efficient multiprocessor systems is the cache coherence protocol. Although directory-based protocols constitute the most scalable approach, the limited size of the directory caches together with the growing size of systems may cause frequent evictions and, consequently, the invalidation of cached blocks, which jeopardizes system performance.
Directory caches keep track of every memory block stored in processor caches in order to provide coherent access to the shared memory. However, a significant fraction of the cached memory blocks do not require coherence maintenance (even in parallel applications) because they are either accessed by just one processor or they are never modified. In this paper, we propose to deactivate the coherence protocol for those blocks that do not require coherence. This deactivation allows directory caches not to keep track of non-coherent blocks, which reduces directory cache occupancy and increases its effectiveness. Since the detection of non-coherent blocks is carried out by the operating system, our proposal only requires minor hardware modifications.
Simulation results show that, thanks to our proposal, directory caches can avoid the tracking of about 66% of the blocks accessed by a wide range of applications, thereby improving the efficiency of directory caches. This contributes either to shorten the runtime of parallel applications by 15% while keeping directory cache size or to maintain performance while using directory caches 16 times smaller.

Index Terms—Multiprocessor, cache coherence, directory cache, operating system, coherence deactivation, non-coherent blocks, efficiency

1 Introduction and Motivation

Nowadays, larger and more powerful shared-memory multiprocessors [7], [15], [24] are increasingly demanded. The efficiency of high-performance shared-memory multiprocessor systems depends on the design of the cache coherence protocol. Directory cache coherence protocols represent the most scalable alternative. Unlike broadcast-based protocols, traditional directories keep track of every memory block in the system, which allows the protocol to easily locate the cached copies without generating large amounts of network traffic.

Since keeping track of every memory block in the system entails huge storage requirements, some recent proposals [21] and commodity systems, such as the current AMD Magny-Cours [7], only keep track of cached memory blocks. In this case, the directory information is only kept in small directory caches [23], [13]. Due to the lack of a full directory, the eviction of directory entries entail the invalidation of the cached copies of the corresponding block. Since the size of directory caches is limited and systems incorporate an increasingly number of processors and cores, directory caches may suffer frequent evictions and, consequently, they may exhibit high miss rates (up to 70% as reported in some recent studies [21], [11]). As a result, the miss rate of processor caches may become excessively high, which can lead to a serious performance degradation.

Although the number of directory evictions can be reduced by using larger directory caches, this is not a scalable solution since it entails both larger directory access latencies and higher directory memory overhead. Instead, we opt to increase the effectiveness of the available space for directory caches, assuming that it will commonly be a scarce resource, especially in large systems. We take advantage of the fact that a significant fraction of the memory blocks accessed by applications does not need coherence maintenance, that is, they are only accessed by one processor (private blocks) or not modified by any processor (read-only blocks). As Figure 1 shows, these blocks account for 82% (on average) of the memory blocks accessed during the execution of a wide range of parallel applications from different benchmark suites. Despite the fact that these blocks do not need coherence maintenance, traditional directory caches still keep track of them. As a consequence, most of the information that they keep is unnecessary, which reduces the effectiveness of the available area for directory caches. On the contrary, if directory caches avoid the tracking of both private and shared read-only blocks, the availability of directory entries for the blocks that actually need coherence (i.e., shared read-write blocks)

• *B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato are with the Department of Computer Engineering, Universitat Politècnica de València, Camino de Vera, s/n, 46021, Valencia, Spain.*
E-mail: {blacuesa, aros, megomez, arobles, jduato}@gap.upv.es

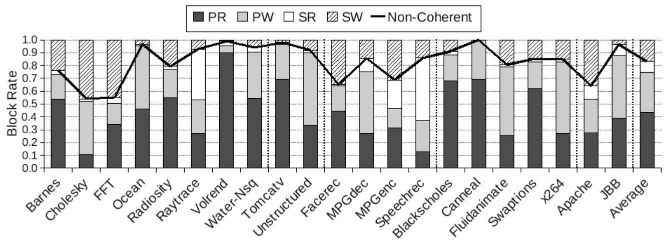

Fig. 1. Block classification. *PR* stands for **P**rivate **R**ead-only, *PW* for **P**rivate read-**W**rite, *SR* for **S**hared **R**ead-only, and *SW* for **S**hared read-**W**rite. *Non-coherent* marks the rate of blocks that do not require coherence.

will increase spectacularly and their capacity could be better exploited. This way, the number of cache misses caused by the blocks evicted from directory caches can be reduced while maintaining the size of directory caches, thereby improving system performance. Alternatively, it may be preferred to reduce the size of directory caches, while still maintaining system performance. This option could be especially intended for environments with severe silicon area constraints, such as systems on chip (SoCs).

To improve the use of directory caches, in [8] we propose to deactivate the coherence just for private blocks. Although most of the blocks are private (74% on average), some applications such as speechrec, raytrace, and, mpgenc present a significant percentage of shared read-only blocks (48.7%, 39.6%, and 22.4%, respectively). Furthermore, scenarios with thread migration could make private read-only blocks (43%) become shared read-only. Therefore, to improve the potential of that earlier approach and to address some of its weaknesses, in this work we extend the mechanism proposed in [8] by being able to additionally detect and deactivate the coherence for read-only blocks. Thus, the proposed mechanism prevents directory caches from tracking both private and read-only blocks. This mechanism (1) relies on the operating system (OS) to dynamically identify non-coherent memory blocks (i.e., both private and read-only blocks) at page granularity, (2) deactivates the coherence for the accesses to such blocks, and (3) triggers a coherence recovery mechanism when a block that has initially been identified as non-coherent becomes coherent.

This proposal 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, the proposed mechanism can avoid the tracking of 66% (on average) of the memory blocks accessed by the applications. By not storing coherence information for those blocks, the number of evictions and, therefore, the number of invalidations issued by memory controllers decreases by about 70%. This results in reductions in the miss rate of processor caches (about 40%), which is translated into performance improvements of 15%. Additionally, processors can maintain performance while using smaller directory caches. Results show that a system that implements our proposal achieves similar performance than a system that does not implement it and employs a directory cache 16 times larger. When compared to a system that deactivates coherence only for private blocks [8], our proposal obtains similar performance when it employs a directory cache 2 times smaller. Finally, dynamic energy consumption can be also reduced by about 40% on average mainly due to the elimination of accesses to both directory caches (5%) and the memory controller (18%), and the reduction in coherence traffic (18%).

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 presents our proposal. We describe the simulation environment in Section 5 and we present the evaluation results in Section 6. Finally, Section 7 draws some conclusions.

2 Related Work

The proposal of this paper is based on the observation that most of the blocks referred to by parallel applications do not require coherence maintenance. We take advantage of this fact to propose a hybrid hardware-OS mechanism that avoids the tracking of those non-coherent blocks. In this section, we comment on some works that are somehow related to our proposal.

Our proposal, like some previous approaches, uses the OS to detect private and read-only blocks. Hardavellas et al. [14] use this detection to propose an efficient data placement policy for distributed shared caches (NUCA). While the mechanism for classifying the pages is similar to ours, its application is completely different (data placement) since it does not consider coherence aspects. In contrast, we focus on how the detection of private/read-only blocks can be used to increase directory effectiveness. On the other hand, Kim et al. [16] employ OS detection to reduce the fraction of snoops in a token-based protocol. Unlike our proposal, that work is based on the fact that, although most referred blocks are private or read-only, the small fraction of shared-written blocks accounts for the majority of the cache misses. Hence, they propose a sophisticated mechanism that detects the sharing degree of blocks so that broadcast messages can be replaced by multicast ones. Unfortunately, this technique requires large TLBs and important hardware and OS modifications. Furthermore, neither [14] nor [16] detects shared read-only data pages. Differently, our mechanism is much simpler and does not require complex hardware/OS modifications. In addition, we target the fraction of private and read-only 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 [25]. 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. [22] 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 [27] 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. [9] 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. [10] extend the region tracking structure proposed by Zebchuk et al. [27] 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 private/shared read-only/read-written (2 bits) and simply deactivates the tracking of blocks for non shared read-written pages.

Other works also support cache coherence by means of a combination of software and hardware. Zeffer et al. [29] 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

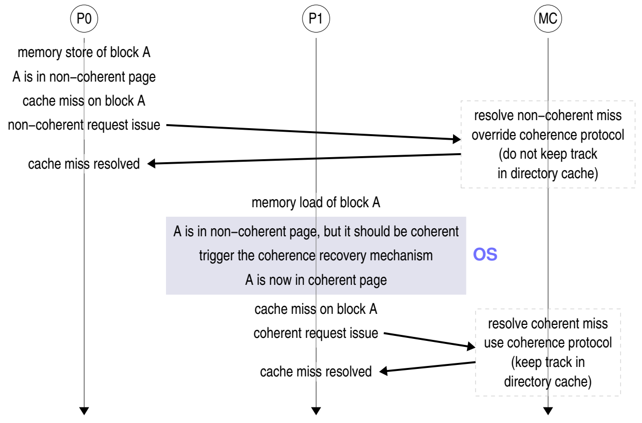

Fig. 2. Overview of the proposed mechanism. *P0* and *P1* are processors and *MC* is the memory controller. The shaded background indicates that the OS is in charge at that moment.

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 [28]. 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. [12] 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

Cache coherence protocols avoid inconsistencies among the different cached copies of memory blocks. Although they indiscriminately act on all the referred memory blocks, a significant part of them cannot suffer from inconsistencies. In particular, both the blocks accessed only by one processor (i.e., private blocks) and those that are never modified (i.e., read-only blocks) cannot suffer from inconsistencies. The unnecessary use of the directory cache for maintaining the tracking of those blocks increases its overload and makes coherence protocols less effective.

We propose a technique that, with the help of the OS, dynamically identifies both private and read-only blocks and deactivates the coherence for them. Since a

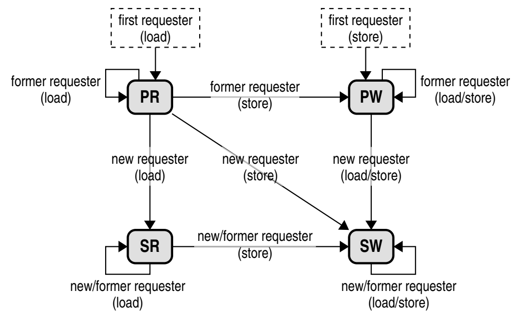
first requester (load)
first requester (store)
former requester (load)
PR
former requester (store)
PW
former requester (load/store)
new requester (load)
new requester (store)
new requester (load/store)
SR
new/former requester (store)
SW
new/former requester (load)
new/former requester (load/store)

Fig. 3. State transition diagram in view of new operations.

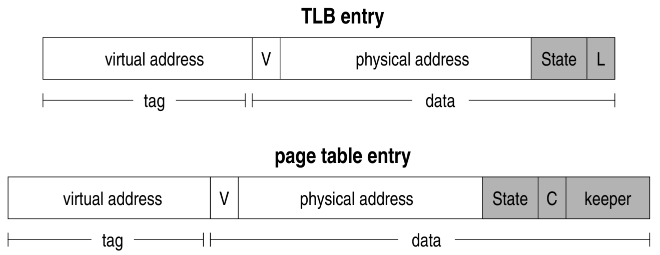

Fig. 4. TLB and page table entry format. Shaded fields are additional fields required by our proposal. V is the valid bit, L is the locked bit, and C is the cached-in-TLB bit.

fine-grain detection (e.g., block granularity) may require a huge amount of hardware resources, our proposal is based on a coarse-grain strategy (page granularity).

The general idea is that, by default, every new page loaded into main memory is classified as non-coherent. The cache misses for the blocks belonging to non-coherent pages are resolved without taking into account the coherence protocol. As a result, directory caches do not track the accesses to non-coherent blocks. As the OS detects subsequent memory accesses, depending on the requester and the access type, the page may evolve to coherent, which requires a coherence recovery process. This process is triggered by the OS and is in charge of restoring the coherence for every block within the involved page. After the recovery process completion, the page is considered as coherent and the memory accesses to its blocks will be tracked by the directory caches.

Figure 2 outlines our proposal. First, *P0* issues a store operation on the memory block *A*, which causes a cache miss. Assuming that *A* belongs to a non-coherent page, *P0* issues a non-coherent request, which is served by the home node (i.e., the memory controller or node where a memory block is mapped to) and no track is kept in the directory cache of main memory (*MC*). Later, another node, for instance *P1*, issues a load operation on the same memory block *A* and a TLB miss happens. While the OS is handling the TLB miss, it realizes that the page should be coherent instead of non-coherent. Consequently, it triggers the coherence recovery mechanism. When it finishes, the page becomes coherent and the access to the cache proceeds, resulting in a miss. Since the block belongs to a coherent 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 page classification (Section 3.1), the behaviour of non-coherent requests (Section 3.2), the updating of the page type (Section 3.3), the TLB-updating (Section 3.4) mechanism, and the coherence recovery mechanism (Section 3.5).

3.1 Page Classification

In order to distinguish the memory pages whose blocks require coherence from those whose blocks do not, we classify them in four types:

- *PR (**P**rivate **R**ead-only) page:* Only one processor accesses its blocks. All the accesses are loads.
- *PW (**P**rivate read-**W**rite) page:* Only one processor accesses its blocks. At least one of the accesses is a store.
- *SR (**S**hared **R**ead-only) page:* At least two processors access its blocks. All the accesses are loads.
- *SW (**S**hared read-**W**rite) page:* At least two processors access its blocks. At least one of the accesses is a store.

According to this classification, blocks within PR, PW, and SR pages do not require coherence, whereas blocks in SW pages could require it. Notice that the type (or state) of a page is not static, but it dynamically evolves as the OS detects new accesses to its blocks. The state transition of pages is illustrated in Figure 3.

3.2 Non-Coherent Requests

On memory references, processors first access their TLB to translate virtual addresses into physical addresses. As shown in Figure 4, 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 three of them to include two new fields: the *state* field (2 bits), which indicates the page state (PR, PW, SR, or SW), and the *locked* field (1 bit), which is used to avoid undesirable race conditions (as explained later in Section 3.5).

The page state is taken into account when a memory reference to one of its blocks causes a cache miss. Hence, if the cache miss is for a block within a PR, PW, or SR page, a *non-coherent* request is issued. Otherwise, a *coherent* request is sent out. 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,

TABLE 1
Updating the page table (PT) and TLBs and use of the TLB-updating and coherence recovery mechanism. *r/k/o-TLB* stands for the requester/keeper/others' TLB.

	I (*C* is clear)	PR	PW	SR	SW
load (new requester)	**PT**:*PR*+C +Keeper **r-TLB**:*PR*	**TLB-updating** **PT**:*SR* **r/k-TLB**:*SR*	**Coherence Recovery** **PT**:*SW* **r/k-TLB**:*SW*	**r-TLB**:*SR*	-
store (new requester)	**PT**:*PW*+C +Keeper **r-TLB**:*PW*	**Coherence Recovery** **PT**:*SW* **r/k-TLB**:*SW*	**Coherence Recovery** **PT**:*SW* **r/k-TLB**:*SW*	**Coherence Recovery** **PT**:*SW* **r/k/o-TLB**:*SW*	-
load (former requester)		-	-	-	-
store (former requester)		**PT**:*PW* **k-TLB**:*PW*	-	**Coherence Recovery** ***PT**:*SW* **r/k/o-TLBs**:*SW*	-

and the power 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.

3.3 Updating the Page State

Similarly to TLBs, page tables also need to keep the state of pages. However, in this case, three additional fields are required, as shown in Figure 4. The *state* field indicates the page type (PR, PW, SR, or SW). The *keeper* field contains the identity of the first processor that cached the page table entry in its TLB. The *cached-in-TLB* bit (C) indicates whether the keeper field is valid or not, that is, the page has been cached in any TLB. 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 $3 + log_2(N)$ bits, where N is the number of nodes in the system. Thus, assuming a system comprised of 8 processors, like the AMD Magny-Cours, only 6 extra bits per entry would be required.

On a page table fault, the OS allocates a new page table entry holding the virtual to physical address translation. The *C* field of this entry is cleared, indicating that it has not been cached in any TLB yet. When a TLB miss takes place and once the page table entry has been cached in a TLB, the *state*, *C*, and *keeper* fields of the entry may require to be updated as indicated in Table 1. Let us analyse each case separately.

If *C* is clear, no processor has accessed the page blocks and, consequently, a load/store to that page will cause a TLB miss. During the resolution of the TLB miss, the page will be set to *PR*/*PW*, respectively, in both the page table and the requester's TLB. Furthermore, the requester's identity will be stored in the keeper field of the page table and *C* will be set.

If *C* is set and the page is labelled as PR, upon a load/store from other processor than the keeper, a TLB

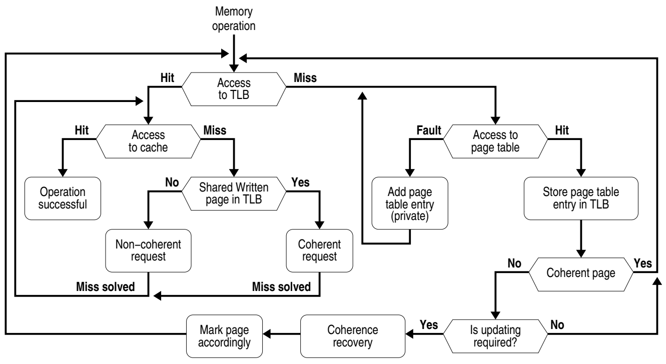

Fig. 5. Block diagram of the general working scheme.

miss will take place. During its resolution, the page state will have to be updated to SR/SW, respectively, in the page table and in the requester's and keeper's TLBs. In case the page transitions to a coherent state (SW), the coherence recovery mechanism will be triggered (see Section 3.5). This mechanism, which is initiated by the new requester during the TLB miss handling, is in charge of evicting all page blocks cached by the keeper and updating the corresponding entry of the keeper's TLB. On the other hand, if the page is going to transition to SR, although it remains in a non-coherent state, it is necessary to update the keeper's TLB. Since the requester does not have direct access to the keeper's TLB, the updating is performed by means of the TLB-updating mechanism (see Section 3.4).

If a load is issued by the keeper for a block within a *PR* page, the page will remain in the same state and no actions will be performed. In turn, if a store is issued, the page state will have to be updated in both the page table and the requester's TLB. Notice that, in this case, a TLB miss may not have occurred because the keeper may already have an entry in its TLB. Therefore, the updating of the page table could incur additional and considerable delay. To avoid it, the updating of the page table is postponed (action marked in gray) until another node tries to access one of the page blocks (the page becomes shared). Notice that this temporal inconsistency between page table and keeper's TLB has no effect while another node does not try to share the page.

When the page state is *SR*, upon a load from a new requester, it just caches the corresponding page table entry in its TLB. However, under stores, the page table and all the sharers' TLBs will have to be updated. In addition, since the page will transition to a coherent state, the coherence of all the blocks within the involved page needs to be restored. To this end, the coherence recovery mechanism is used. Particularly, since nobody keeps the list of sharers, the recovery mechanism will have to perform a broadcast to evict from caches all copies of blocks within the corresponding page and updating all the sharers' TLBs. This process is also done in case the page is *SR* and a former requester wants to store one of its blocks. However, in this case, since a TLB miss may not happen, an especial exception (marked as

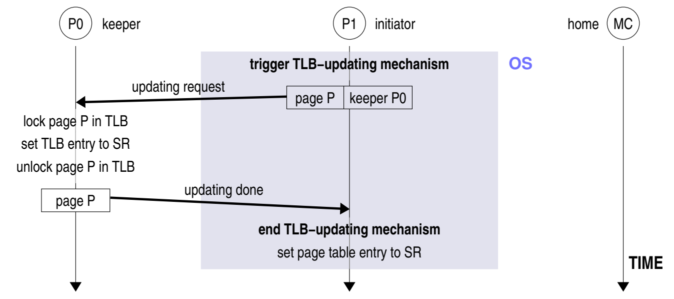

Fig. 6. TLB-updating mechanism. *P0* and *P1* are processors and *MC* is the home node.

*) is forced to update all the TLBs and the page table. This forced exception will incur into additional delay. However, as we will see in the evaluation, this exception is not very frequent and, furthermore, its delay will be largely offset by the advantages of avoiding the tracking of blocks that do not require coherence.

Finally, when the page is *PW*, new loads or stores from the keeper will cause no change. However, both loads or stores from new requesters will cause the page to transition to *SW*. In this case, during the TLB miss resolution, the page table and the keeper's TLB are updated and the coherence is restored by means of the coherence recovery mechanism.

Pages marked as *SW* do not require any transition because, once they become coherent, they remain in that state. Figure 5 outlines the interactions among system components to solve memory operations.

3.4 TLB-Updating Mechanism

The TLB-updating mechanism is triggered when a page transitions from PR to SR. Since the page state changes, this mechanism is in charge of updating the keeper's TLB. However, as the page remains being non-coherent, the page blocks do not need to be evicted to recover coherence. Figure 6 shows a detailed example of this mechanism. The initiator (node that triggers the mechanism) sends an *updating request* to the page keeper, which has been obtained from the page table on processing its TLB miss. Upon its receipt, the keeper updates the corresponding TLB entry (if present) and informs the initiator by an *updating done* message. When the initiator receives it, the mechanism finishes.

3.5 Coherence Recovery Mechanism

When a page initially considered as non-coherent becomes coherent, the coherence recovery mechanism must be triggered. This mechanism ensures that from that moment the directory cache will hold proper track of all cached blocks within the page. Since these blocks have not been tracked so far, we propose the simple strategy of just evicting them from caches (flushing-based recovery [8]). After the recovery process completion, since the page will be marked as coherent, the directory cache will be able to keep correct track of each of the page

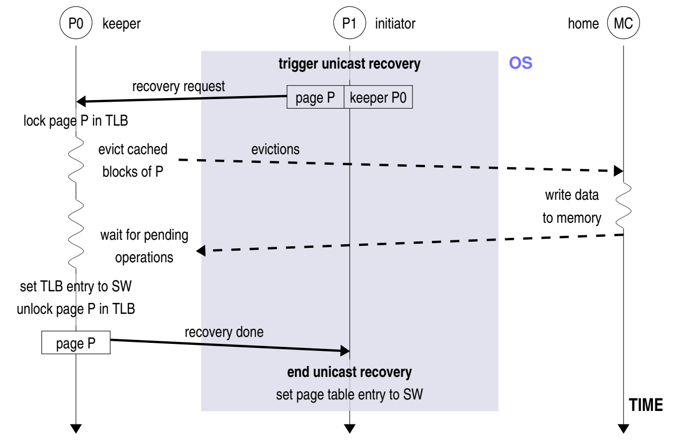

Fig. 7. Coherence recovery mechanism for a private page. *P0* and *P1* are processors and *MC* is the home node.

blocks. The coherence recovery mechanism, which is triggered while managing either the corresponding TLB miss or the exception forced when a former requester tries to stored a block within a SR page (as commented in Section 3.3), works as follows.

First, the initiator issues a *recovery request* (with the address of the page to recover) to the page keeper, whose identity was obtained from the corresponding page table entry.

Second, on the recovery request arrival, the keeper locks the corresponding TLB entry (*L* bit). This prevents the keeper from issuing new requests for the blocks within the page. In case the TLB entry is not present, it is not necessary to lock it since new requests will not be able to be issued (the initiator is accessing the page table entry inside a critical section). After this, if the page to recover is SR, the keeper broadcasts a *recovery probe* for the page because other nodes may have cached page blocks. On the contrary, if the page is PW or PR, this broadcast is not necessary because only the keeper may have cached copies.

Third, the possible receivers of the probe (if any) lock the page in their TLBs and both they and the keeper perform a cache lookup and flush every cached block of the involved page. When finishing, they check their MSHRs (Miss Status Holding Registers) which keep track of outstanding cache misses. While there is at least one pending cache miss for some of the page blocks, they wait for its completion. The blocks for the outstanding misses are not cached when the recovery mechanism is ongoing. Therefore, once the pending misses for the involved page are resolved, the corresponding TLB entry is set to SW. After this, all the non-keeper nodes inform the keeper by means of a *recovery target done* message.

Fourth, when the keeper has collected all the recovery target done messages (this step is required only if a recovery probe was broadcast), it unlocks the TLB entry and sends a *recovery done* message to the initiator.

Fifth, when the initiator receives the recovery done message, the recovery mechanism finalizes and the page

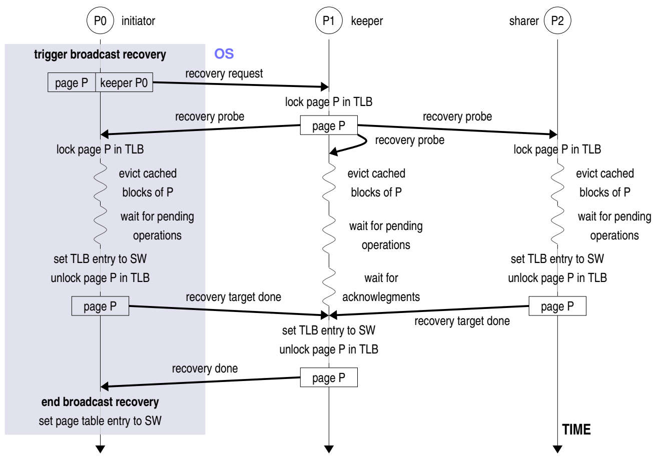

Fig. 8. Coherence recovery mechanism for a shared page. *P0*, *P1*, and *P2* are processors.

can be set to SW in both the initiator's TLB and the page table. 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.

Figures 7 and 8 illustrate the main differences between recoveries for private pages (unicast-based mechanism) and for shared pages (broadcast-based mechanism), respectively. After completing the execution of the recovery mechanism for a page, we know for sure that the blocks belonging to it are not cached. Therefore, the next time a processor references one of those blocks, a coherent request will be issued and, since the page is considered as coherent, the directory cache will be able to keep proper track of it.

3.6 Discussion on Coherence Recovery

In this section, we discuss about the latency of the coherence recovery mechanism and their adaptation to systems with hardware page table walkers.

The recovery process may take a long time because its critical path may include (1) a search in the keeper's cache and a search in the sharer's caches when the page is SR and (2) several evictions. Despite its high latency, it must be taken into account that the recovery process is only performed very few times. In particular, during the lifetime of a page in main memory, at most one recovery mechanism and one TLB-updating mechanism could be triggered. However, during that time, the page will probably have a large number of references according to the locality principle. 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 coherence recovery mechanism. In Section 6, we show quantitative data of this and we observe that the recovery and the TLB-updating mechanisms together are triggered less than 5 times per 1000 cache misses (on average). Thus, the impact of the recovery mechanisms on the protocol performance is indeed negligible since it is largely offset by the savings in cache misses and the reduction in their latency.

Although in this work we link the description of our proposal to traditional page tables, its application is also possible in systems that use hardware page table walkers. Indeed, the adjustment to that 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 coherent 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 do not carry out such an implementation.

4 Contributions of Shared Read-only Blocks

The main difference between the proposal made in this work and that in [8] is the deactivation of the coherence protocol for the accesses to SR memory blocks. In this section, we deal with the pros and cons of this new proposal.

On one hand, the detection and coherence deactivation of SR pages may present the following drawbacks:

- The detection of SR memory blocks requires additional resources. In particular, TLBs and page tables need one additional bit to code 4 states (PR, PW, SR, and SW). Although one additional bit does not significantly increase the storage requirements of this proposal, it makes the state transition and the logic a little more complex.
- The coherence recovery process when detecting SR memory blocks is more subtle. Furthermore, an additional TLB-updating mechanism is necessary to update the keeper's TLB on detecting a page state transitions from PR to SR. The latency of this mechanism (analyzed in Section 6.1.3) is quite low and, although it may increase the latency of some TLB misses, it avoids the triggering of the coherence recovery mechanism (which is much slower), as done in [8].
- If a processor holds a valid TLB entry for a certain page and the state of such a page transitions from SR to SW, in absence of a TLB miss, an additional OS exception will be forced to initiate the coherence recovery mechanism for the page. Although this additional exception is rarely required, it incurs considerable delay.
- Since SR pages can be shared across several processors and no track is kept about it, a coherence recovery mechanism based on broadcast is required. In [8] a unicast recovery mechanism in only required. This adds complexity to the recovery process and increases the recovery traffic. However, compared to the coherence traffic, it is unnoticeable.
- Due to the complexity of the recovery process, the implementation of a coherence recovery mechanism

based on updating like the one presented in [8] (instead of flushing) is discarded. Nevertheless, as analyzed in [8], the flushing-based recovery is recommended as it is more easily implementable on actual systems.

- Although the deactivation of the coherence for the shared read-only blocks does not jeopardize the correctness, it may cause inefficiencies. If SR blocks are treated as coherent, cache misses for them will be probably served by the owner processor. However, when we consider them as non-coherent, they will always be served from main memory. As a result, their latency may increase considerably. Nevertheless, as we see in Section 6, the advantages of avoiding the tracking of SR blocks outweigh this possible drawback.

On the other hand, the detection and coherence deactivation of SR pages provides many additional advantages that clearly offset the drawbacks pointed out above (as later analyzed in Section 6):

- The coherence protocol can be deactivated for a considerably larger number of memory blocks. As a result, the beneficial features of the proposed technique increase, thereby leading to significant improvements in performance and to greater scalability.
- The classification of memory pages in PR, PW, SR, and SW decreases the number of blocks misclassified as coherent due to the use of a coarse-grain detection. As a result, the detection mechanism is more accurate and their advantages can be better exploited.
- Since more pages are detected as non-coherent, the coherence recovery mechanism is triggered less times, thereby causing less overhead.
- It partially addresses the problem that the proposal in [8] has with respect to thread migration. Using that proposal, all blocks privately accessed by a thread will be identified as shared after it migrates and the coherence cannot be deactivated for them. On the contrary, in this proposal, the private read-only blocks (more than 40% on average according to data in Figure 1) of a thread after migrating will be able to be detected as shared read-only blocks and, as a result, they will be able to be considered as non-coherent. Notice, though, that this proposal does not tackle the problem of thread migration for PW blocks.

Following sections show quantitative data of the advantages of deactivating the coherence for SR blocks.

5 Evaluation Methodology

We evaluate our proposal with full-system simulation using Virtutech Simics [19] running Solaris 10 and extended with the Wisconsin GEMS toolset [20], which enables detailed simulation of multiprocessor systems.

TABLE 2
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 $2\times$, worst-case $0.25\times$
Memory access latency (local bank)	60ns
Page size	4KB (64 blocks)

Network Parameters	
Network topology	Hypercube with 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

For modeling the interconnection network, we use GARNET [1], which is a detailed network simulator included in GEMS. Finally, we also use the McPAT tool [18], assuming a 45nm process technology, to measure the savings in terms of energy consumption of our proposal.

For the evaluation of our proposal, we first model a cache coherent HyperTransport system optimized with directory caches (PFs) 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 only are 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 account that such a protocol would considerably increase the simulation time, we do not model it either.

In Magny-Cours, dies are made coherent by using a directory-based cache coherence protocol that implements MOESI states. Each PF 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 PF comprises just one pointer to the owner node (3 bits). Typically, each PF has 256K entries and each die has 128K entries in its cache hierarchy. Therefore, the coverage ratio of PFs is $2\times$ (i.e., PFs have twice as many entries as blocks can be cached). This would be enough for tracking all the cached blocks if they were distributed uniformly among all the PFs. However, cached blocks may not be distributed uniformly. Thus, the worst-case scenario appears when all the cached blocks belong to the same memory controller (known as hotspotting), in which the coverage ratio dramatically decreases down to $0.25\times$.

We consider the described system as the *base* architecture and its main parameters are shown in Table 2. Our proposal is implemented upon this system and it is referred to as *deact Priv/SR* (deactivation of private/SR blocks). The previous proposal on which this work is based [8] is referred to as *deact Priv* (deactivation of private blocks).

TABLE 3
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

We evaluate our proposal with a wide variety of parallel workloads (21) from 3 suites (SPLASH-2 [26], ALPBenchs [17], and PARSEC [5]), two scientific benchmarks, and two commercial workloads [2], which are shown in Table 3. Due to time requirements, we are not able to simulate these benchmarks with large working sets. Consequently, as done in most works [6], [11], [12], we simulate the applications assuming smaller data-sets. To avoid altering the results, we reduce the size of both processor caches and directory caches accordingly to application data-sets. Particularly, the simulated caches are four times smaller than those assumed by 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 the original Magny-Cours (2×).

All the reported experimental results correspond to the parallel phase of benchmarks. We account for 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

We organize the evaluation of our proposal in two parts. First, in Section 6.1, we compare our proposal with the base system and second, in Section 6.2, we evaluate the additional contributions that *deact Priv/SR* offers with respect to those offered by *deact Priv*. Besides, in that section, we also study the impact of both approaches when the size of directory caches is reduced.

6.1 Evaluating the Coherence Deactivation

This section illustrates how the mechanism proposed in this paper (*deact Priv/SR*) is able to considerable reduce

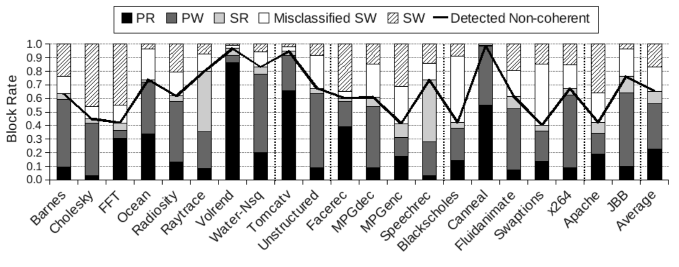

Fig. 9. Block classification.

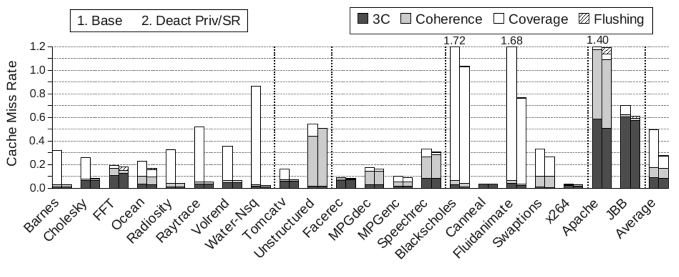

Fig. 10. Cache miss rate (in percentage).

the amount of blocks tracked by directory caches. This results in less processor cache misses, which leads to performance improvements. Additionally, it also saves energy consumption.

6.1.1 Non-Coherent Blocks

As Figure 9 shows, about 84% (on average) of the referred memory blocks are actually non-coherent (i.e., PR, PW, SR, or misclassified SW) and, consequently, they do not require coherence. Since our mechanism is based on a coarse-grain classification of blocks, it is not able to identify all the non-coherent blocks. In particular, it detects that 66% (on average) of the referred blocks do not require coherence (*detected non-coherent* line in the figure). The remaining 34% are classified as SW blocks (i.e., both actual SW and misclassified SW) and, therefore, they require coherence. According to these data, the use of a coarse-grain approach causes 18% of the referred blocks to be misclassified, which offers a good trade-off between required resources and detection accuracy.

6.1.2 Processor Cache Misses

Since directory caches do not track cached blocks detected as non-coherent, they are less congested. Therefore, they suffer less evictions and, consequently, less blocks are invalidated from processor caches. As a result, the processor cache miss rate is reduced by about 45% (on average), as Figure 10 shows. In this figure, cache misses are classified in four groups: *3C* misses are Cold, Capacity, and Conflict misses; *Coherence* misses refer to those caused by invalidations due to store operations issued by other processors; *Coverage* misses are those caused by the invalidations issued as a consequence of evictions in directory caches; and *Flushing* misses are due to invalidations performed by the recovery mechanism. Since our proposal improves the effectiveness of directory caches, it mainly acts on the coverage

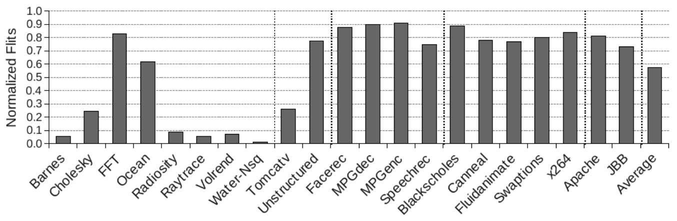

Fig. 11. Normalized network traffic.

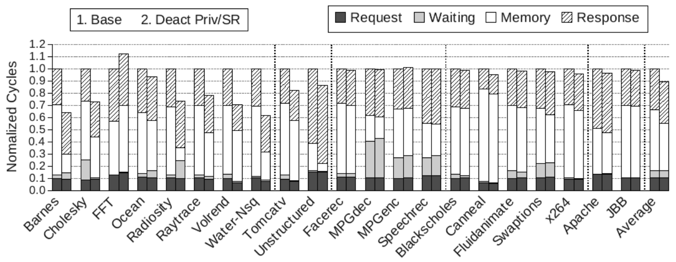

Fig. 12. Normalized cache miss latency.

misses, which are significantly reduced from about 70% in the base system to 30% when using our proposal. As shown, our proposal is not able to completely remove all the coverage misses. Indeed, the reduction of coverage misses partially depends on the accuracy of the detection mechanism. Thus, in applications like radiosity, volrend, or tomcat (among others), few blocks are misclassified as SW. As a result, most of the non-coherent blocks are classified as such and directory caches omit their tracking. This allows directory caches to be less congested, which leads to avoid all the coverage cache misses. This achievement is important because, as reported in other studies [21], [11], the number of coverage misses may be really important in some scenarios and it is reasonable to think that it will grow in future multiprocessor systems since they are increasingly larger. On the other hand, in applications like blackscholes, fluidanimate, or swaptions the number of misclassified blocks is considerable, thereby leading to moderate reductions in the number of coverage misses.

When a page transitions to SW, the recovery mechanism evicts from caches all its blocks, which leads to additional misses referred to as flushing misses. Thus, as shown in Figure 10, the recovery mechanism causes 2% (on average) of additional misses. Notice, though, that the reduction in coverage misses is so significant that it largely offsets that increment.

The reduction of both directory evictions and cache misses has a meaningful impact on network traffic, as depicted in Figure 11. Bars plot the total number of flits transmitted through the interconnection network when the coherence is deactivated normalized to the network traffic generated by the base protocol. Those data include the traffic due to the TLB-updating and the coherence recovery mechanisms. However, this traffic is not shown separately because it is really insignificant (lower than 0.5% on average). As shown in the figure, the coherence deactivation causes a reduction in network traffic of about 42% on average.

Our proposal is not only able to reduce the cache miss rate, but also their average latency, as Figure 12 depicts. In this figure, the latency of cache misses is split into 4 stages: *request* latency refers to the transmission latency of requests to the home node; *waiting* is the time that requests remain in the home waiting for the beginning of their service; *memory* is the latency of the memory controller, which also includes the latency of the directory cache (if accessed) and main memory; finally, *response* is the latency from either the issue of the memory response (if required or non-coherent request) or the forwarding of the request to another processor until the completion of the miss (if coherent request). Since requests for non-coherent blocks do not need a directory cache lookup, their memory latency is smaller, which lowers the average latency of cache misses by about 10% on average. Notice that, for some applications like fft, facerec, mpgdec, mpgenc, speechrec, blackscholes, and jbb, the memory latency is equal to or higher than that of the base system and, therefore, the average miss latency is not reduced. This happens due to the fact that SR blocks are considered as non-coherent. Since they are considered as non-coherent, they must always be served by main memory, whose response latency is highly. On the other hand, in the base system, SR blocks are considered as coherent. Therefore, in some cases, cache misses for SR blocks may be served by caches instead of by memory, which is faster. As a result, for those applications the latency reduction of cache misses for private blocks is balanced (or overcome) by the latency increase of cache misses for SR blocks. Despite this fact, the benefits of avoiding the tracking of SR blocks clearly outweighs the possible increase of miss latency as seen in next sections.

6.1.3 TLB-Updating/Coherence Recovery Mechanisms

Figure 13 shows the average latency of both TLB-updating and coherence recovery mechanisms according to the timing parameters shown in Table 2. The latency of these mechanisms is split into several components: *request* is the latency of transmitting updating/recovery requests and, if required, recovery probes; *flushing* is the latency of issuing the evictions of all the cached blocks within the page to flush; *waiting* is the latency of finishing the evictions (waiting, if required, for receiving the acknowledgements from home); *response* is the latency of informing to the initiator of the finalization of the page flushing; and *ack* is the latency of collecting the recovery target done packets, which are only used in case of broadcast. As shown, the latency of the TLB-updating mechanism (first bar of each group) is negligible because it only comprises the latencies of updating requests and updating done messages (responses). The latency of the unicast recoveries (second bar) is higher than that of the TLB-updating because they additionally include the latencies of waiting and flushing, which are quite important. The latency of the recoveries that require broadcast (third bar) is even higher than that of the

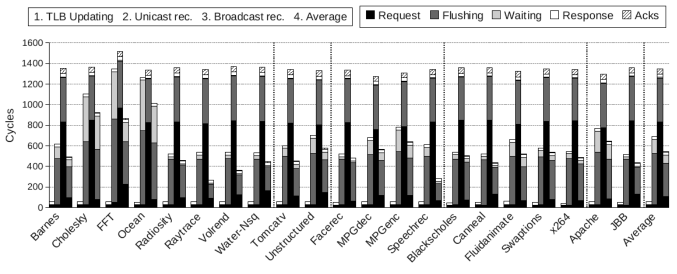

Fig. 13. Average latency of the TLB-updating and recovery mechanisms.

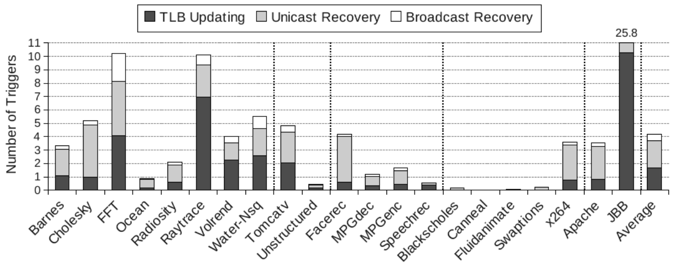

Fig. 14. Number of recoveries and TLB-updating per 1000 misses.

unicast recoveries mainly due to the need to broadcast the recovery probes and collect the recovery target done messages (acks). Last bar of each group shows the average latency of all these mechanisms taking into account the number of times that every one is triggered. Thus, since the TLB-updating and the unicast recovery are much more frequent than the broadcast recovery, the average latency is slightly lower than that of the unicast recovery mechanism.

Despite the fact that the latency of the coherence recovery mechanism can be considerable (mainly in case of broadcast), this mechanism is not frequently used. To illustrate this statement, we estimate the number of times that the TLB-updating and the coherence recovery mechanisms are triggered with respect to the total number of misses. As show in Figure 14, on average, the mechanisms are only triggered less than 5 times per 1000 cache misses (up to 26 for the jbb application). As a result, their impact on system performance is almost unnoticeable compared to the impact that cache misses have on it.

6.1.4 *Execution Time*

Mainly due to the reduction in the number of cache misses (and, in some cases, the additional reduction of the cache miss latency), the runtime of applications can significantly lower, as depicted in Figure 15. According to these data, our proposal improves application runtime by 15% on average. As shown, for applications where both cache miss rate and latency are significantly reduced (barnes, cholesky, and waternsq among others), the system performance considerably improves. However, for applications where the reduction of caches misses is not so significant, the improvements on performance are more moderate.

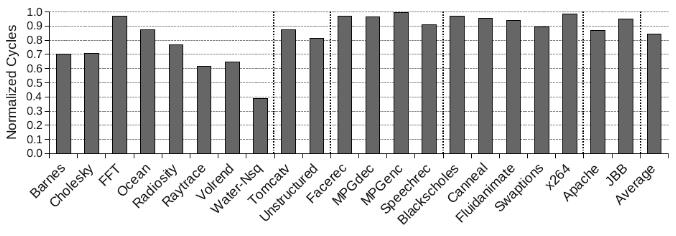

Fig. 15. Normalized runtime of applications.

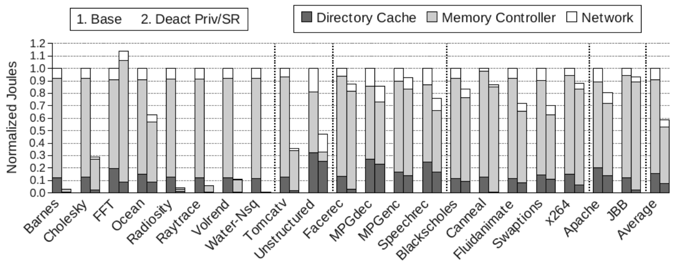

Fig. 16. Normalized energy consumption.

6.1.5 *Energy Consumption*

Thanks to the reduction in cache misses and network traffic, our proposal is also able to reduce system energy consumption. Figure 16 shows the dynamic energy consumption of directory caches, memory controllers, and the interconnection network.

Since non-coherent requests do not need to access directory caches, their consumption is reduced.

Although our proposal decreases the number of memory accesses (due to the cache miss reduction, as shown in Figure 10), the recovery mechanism may increase it (due to the eviction of cached blocks). However, on average, the referred reduction offsets this increase. As a result, the energy consumption of memory controllers is reduced by 45% on average. Notice that, for the FFT application, the number of flushed blocks is noticeable and, therefore, the energy consumption of memory controllers increases slightly.

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 11. Taking into account the overall consumption of directory caches, memory controllers, and the interconnection network, we can see that energy consumption is reduced by about 40% on average.

Regarding static energy consumption (not shown in Figure 16), 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.

6.2 Impact of Coherence Deactivation for SR Blocks

In this section we evaluate the contributions that the deactivation of private/SR blocks (*deact Priv/SR*) offers in comparison to deactivate the coherence just for private blocks (*deact Priv*). In particular, throughout this section we show how *deact Priv/SR* behaves better than *deact*

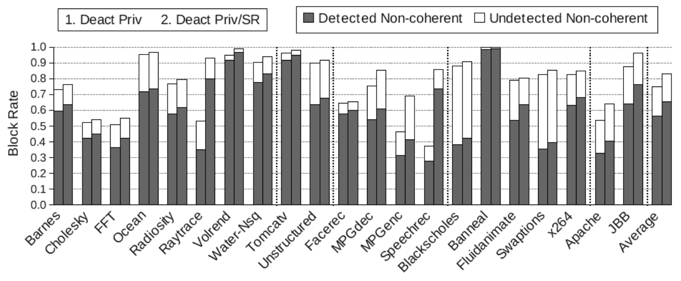

Fig. 17. Potential and accuracy of *deact Priv* and *deact Priv/SR*.

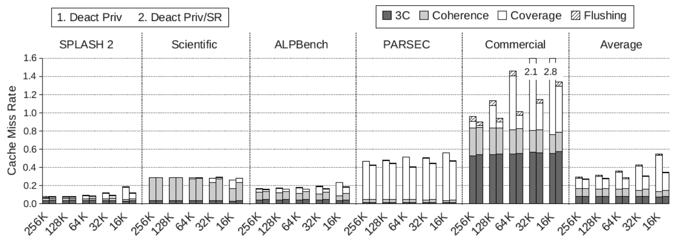

Fig. 18. Cache miss rate of *deact Priv* and *deact Priv/SR* in systems with 256K, 128K, 64K, 32K, and 16K directory caches.

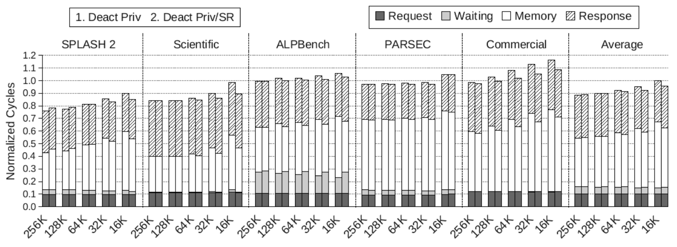

Fig. 19. Normalized cache miss latency of *deact Priv* and *deact Priv/SR* in systems with 256K, 128K, 64K, 32K, and 16K directory caches.

Priv in scenarios with directory caches of reduced size. We do this because, since *deact Priv* is effective enough for removing almost all the coverage misses of most of the evaluated scenarios (see Figure 10), the application of *deact Priv/SR* on those scenarios only leads to small improvements. However, as the size of directory caches is reduced, the number of coverage misses increases and *deact Priv* is not able to remove all them. In those systems with small directory caches, the benefits of *deact Priv/SR* are more visible because it improves the effectiveness of *deact Priv*. To illustrate this, we compare *deact Priv* against *deact Priv/SR* in systems using directory caches whose sizes range from 256K to 32K. For the sake of clarity, the graphs only show the average latency of the applications grouped in application suites and the average value for all the simulated applications.

6.2.1 Potential

Figure 17 illustrates the potential provided by *deact Priv/SR* with respect to that of *deact Priv*. The total value of bars indicates the rate of maximum number of non-coherent blocks (private blocks in *deact Priv* or private/read-only blocks in *deact Priv/SR*) that can be detected in each case. Each bar is divided into detected non-coherent blocks and undetected non-coherent blocks (i.e., blocks misclassified as shared in *deact Priv* or as SW in *deact Priv/SR*). The remaining represents the coherent blocks. As shown, in applications like raytrace, mpgenc, or speechrec, *deact Priv/SR* substantially increases the number of potential blocks that can be detected as non-coherent from 53%, 48%, and 39% to 91%, 70%, and 85%, respectively. Hence, the detection of SR blocks allows the mechanism to increase the number of detected non-coherent blocks. Thus, on average, *deact Priv/SR* detects about 9% more non-coherent blocks than *deact Priv*. Furthermore, the percentage of misclassified blocks (undetected NC) is slightly reduced from 33% in *deact Priv* to 27% in *deact Priv/SR* (with respect to the total number of non-coherent blocks), which indicates that *deact Priv/SR* improves the accuracy of the detection mechanism.

6.2.2 Processor Cache misses

Figure 18 shows the cache miss rate of *deact Priv* and *deact Priv/SR*. As the directory cache size reduces, the number of coverage misses increases because directory caches are not able to simultaneously track all the cached coherent blocks. However, notice that when using *deact Priv*, the cache miss rate grows much more quickly than when *deact Priv/SR* is used. This happens because, as the potential of *deact Priv/SR* is higher and it can act on more blocks, directory caches do not need to track so many blocks. As a result, the reduction of the directory cache size affects a less number of coherent blocks, which leads to a slower growth of the cache miss rate.

Figure 19 shows the cache miss latency normalized to that of the base system^1. For systems with 256K directory caches, the miss latency of *deact Priv* is slightly smaller than that of *deact Priv/SR* mainly because *deact Priv* considers SR blocks as coherent and, therefore, misses for SR blocks may be served by caches instead of memory. As the directory cache size decreases, the number of coverage misses increases due to the increase in evictions of cached blocks. Thus, in *deact Priv*, the SR and SW blocks that are evicted from caches will have to be served by memory again (when they are requested) and, since they are considered as coherent, their memory latency includes the access to directory caches, which makes the average latency of those cache misses increase. On the other hand, when using *deact Priv/SR*, the resolution of misses for SR blocks does not need to access the directory cache, which makes their resolution latency faster than in *deact Priv*. Hence, although in systems with 256K directory caches, the average miss latency of *deact Priv* is slightly smaller than that of *deact Priv/SR*, in systems with 128K directory caches their latencies become equal. In addition, as the directory cache size continues to diminish, the average miss latency of *deact*

1. Notice that the base system assumes 256K directory caches.

TABLE 4
Comparison between TLB updating/recovery mechanisms for *deact Priv* and *deact Priv/SR* with 256K directory caches. *Overhead* represents the overhead of the mechanisms in *deact Priv/SR* with respect to that in *deact Priv*. A positive value indicates that the overhead increases, whereas a negative value indicates a reduction.

	average latency (cycles)		number of triggers		maximum overhead of mechanisms in
	Deact Priv	Deact Priv/SR	Deact Priv	Deact Priv/SR	*deact Priv/SR* normalized to *deact Priv*
SPLASH 2	707	604	1929	2120	-6.11%
scientific	587	513	1999	2269	-0.8%
ALPBench	590	495	6351	6767	-10.61%
PARSEC	545	496	4974	5238	-4.16%
commercial	611	539	22183	25231	0.34%
average	626	543	5432	5963	-4.78%

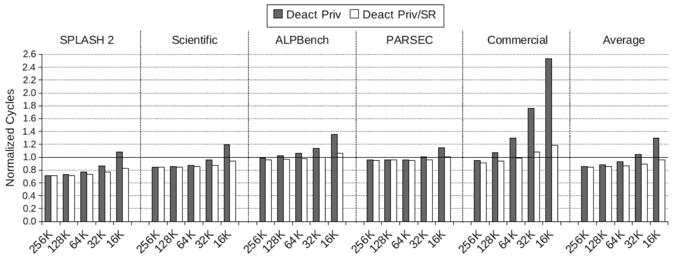

Fig. 20. Normalized runtime of applications for *deact P* and *deact P/SR* in systems with 256K, 128K, 64K, 32K, and 16K directory caches.

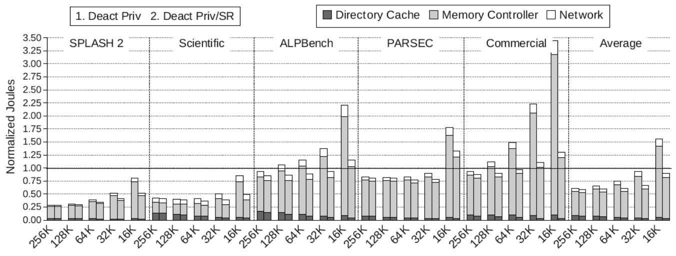

Fig. 21. Normalized dynamic energy consumption.

Priv/SR becomes smaller than that of *deact Priv*.

6.2.3 TLB-Updating/Recovery Mechanisms

Since the performance of these mechanisms does not depend on the directory cache size, we only show the comparison in systems with 256K directory caches. Second and third columns of Table 4 contain the average latency of the TLB/recovery mechanisms (considering the number of times that each mechanism is triggered). As shown, in *deact Priv/SR* the mechanisms are faster (on average) because, as commented in Section 6.1.3, the TLB-updating mechanism is quite fast in comparison to the recovery mechanisms, which reduces their overall average latency. Table 4 also shows the number of times that the mechanisms are triggered. Since *deact Priv/SR* requires the TLB-updating mechanism (which is not used in *deact Priv*), in *deact Priv/SR* the mechanisms are more frequently triggered. Last column of Table 4 illustrates the maximum overhead of *deact Priv/SR* respect to the overhead of *deact Priv*. As shown, the reduction of the average latency of recoveries in *deact Priv/SR* offsets the increase of their triggers. As a result, in *deact Priv/SR* the mechanisms have less impact on the overall performance. Notice that coherence recovery mechanism is triggered less times in *deact Priv/SR* because less pages need to be converted from non-coherent to coherent. Only in case of commercial applications, the overhead is similar for both proposals. This happens because the TLB-updating mechanism further increases the total number of triggers when considering SW pages as SR.

6.2.4 Execution Time

Figure 20 compares how the execution time of applications varies according to the directory cache size. As observed, *deact Priv/SR* gets better results as the size of directory caches decreases. In particular, *deact Priv/SR* achieves to reduce up to sixteen times the directory cache size while maintaining application runtime below that of the base system. However, in case of *deact Priv*, directory caches can only be reduced eight times and, even in that case, the runtime slightly increases with respect to that of the base system. Hence, as directory caches are smaller, *deact Priv/SR* offers better performance than *deact Priv*, which indicates it provides higher scalability.

6.2.5 Energy Consumption

Figure 21 illustrates the evolution of the dynamic energy consumption as the directory cache size is decreased. On the one hand, the consumption of directory caches is reduced as the directory cache size decreases because, despite the fact that smaller directory caches suffer more accesses (due to a larger number of entry evictions), their access latency is lower, which offsets such an increase of accesses.

Regarding memory controllers, their energy consumption increases as the directory cache size is reduced mainly because of the increase in the accesses to memory controllers. Despite this, the dynamic energy consumption of a system using our proposal remains lower (5% on average) than that of the *base* system using directory caches 16 times larger.

Regarding static energy consumption (not shown in Figure 21), 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. 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, 8, and 16 times smaller than that of the base system, the static power consumption is reduced by 48%, 74%, 86%, and 92% respectively.

7 Conclusions

In this paper we propose 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 blocks that do not require coherence maintenance. These blocks comprise not only private memory blocks, but also shared read-only blocks. The OS is responsible for dynamically classifying the accessed blocks according to a coarse granularity. Our proposal reduces the amount of information that has to be stored in directory caches. As a result, the number of blocks invalidated due to evictions in directory caches can be drastically reduced. This advantage can be used not only for increasing system performance (15%), but also for maintaining the same performance having directory caches 16 times smaller. The latter achievement is very useful to cope with the silicon area constrains arisen in the design of many-core chips.

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] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. *To appear in Int'l Symp. on Computer Architecture (ISCA)*, June 2011.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. POWER7: IBM's next-generation server processor. *IEEE Micro*, 30(2):7–15, Apr. 2010.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

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

[29] 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.