Increasing the Effectiveness of Directory Caches by Avoiding the Tracking of Noncoherent Memory Blocks

Blas Cuesta, Alberto Ros, *Member*, *IEEE*, María E. Gómez, *Member*, *IEEE Computer Society*, Antonio Robles, *Member*, *IEEE Computer Society*, and José Duato

Abstract—A key aspect in 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 means directory caches do not have to keep track of noncoherent blocks, which reduces directory cache occupancy and increases its effectiveness. Since the detection of noncoherent 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 percent (on average) of the blocks accessed by a wide range of applications, thereby improving the efficiency of directory caches. This contributes either to shortening the runtime of parallel applications by 15 percent (on average) while keeping directory cache size or to maintaining performance while using directory caches 16 times smaller.

Index Terms—Multiprocessor, cache coherence, directory cache, operating system, coherence deactivation, noncoherent block

1 Introduction and Motivation

Nowadays, larger and more powerful shared-memory multiprocessors [10], [19], [28] are increasingly demanded. The efficiency of high-performance shared-memory multiprocessors 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 enables 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 [25] and commodity systems, such as the current AMD Magny-Cours processor [10], only keep track of cached memory blocks. In this case, the directory information is only kept in small directory caches [27], [17]. Due to the lack of a full directory, the eviction of directory entries entails the invalidation of the cached copies of the corresponding block. Since the size of directory caches is limited and systems incorporate an increasing number of processors and cores, directory caches may suffer frequent evictions and, consequently, they may exhibit high miss rates (up to 70 percent as reported in some recent studies [25], [14]). As a result, the miss rate of processor caches may become excessively high, which can lead to 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, i.e., they are either only accessed by one processor (private blocks) or not modified by any processor (read-only blocks). As Fig. 1 shows, these blocks account for 82 percent (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 space for directory caches. However, 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) will increase spectacularly and their capacity could be better exploited. This way, the number of cache misses

• *The authors are with the Department of Computer Engineering (DISCA), School of Computer Science, Universitat Politecnica de Valencia, Camino de Vera, s/n, Valencia 46021, Spain.*
E-mail: {blacuesa, aros, megomez, arobles, jduato}@gap.upv.es.

Manuscript received 12 May 2011; revised 1 Nov. 2011; accepted 25 Nov. 2011; published online 4 Dec. 2011.
Recommended for acceptance by R. Melhem.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2011-05-0318.
Digital Object Identifier no. 10.1109/TC.2011.241.

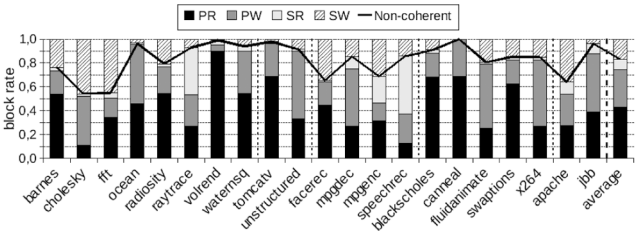


Fig. 1. Block classification on a per block basis. *PR* stands for **P**rivate **R**ead-only, *PW* for **P**rivate read-**W**rite, *SR* for **S**hared **R**ead-only, and *SW* for **S**hared read-**W**rite. *Noncoherent* marks the rate of blocks that do not require coherence.

caused by the blocks evicted from directory caches can be reduced while maintaining the size of directory caches, thereby improving a system performance. Alternatively, it may be preferable to reduce the size of directory caches while still maintaining a system performance. This option could be especially intended for environments with severe silicon area constraints, such as embedded systems or systems on chip (SoCs).

To improve the use of directory caches, in [11] we propose to deactivate coherence just for private blocks. Although most of the blocks are private (74 percent 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 percent, respectively). Furthermore, scenarios with thread migration could make private read-only blocks become shared read-only. Therefore, to improve the potential of that earlier approach and to address some of its weaknesses, in this paper we extend the mechanism proposed in [11] to additionally detect and deactivate 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 noncoherent memory blocks (i.e., both private and read-only blocks) at page granularity, 2) deactivates coherence for the accesses to such blocks, and 3) triggers a coherence recovery mechanism when a block that has initially been identified as noncoherent becomes coherent.

This proposal only requires minor modifications in the OS and 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) and page tables (PT).

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 percent (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 percent. This results in reductions in the miss rate of processor caches (about 40 percent), which is translated into performance improvements of 15 percent. Additionally, processors can maintain performance while using smaller directory caches. Results show that a system that implements our proposal achieves similar performance to 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 [11], our proposal achieves similar performance when it employs a directory cache half the size. Finally, dynamic energy consumption can be also reduced by about 40 percent on average mainly due to the elimination of accesses to both directory caches (5 percent) and the memory controller (18 percent), and the reduction in coherence traffic (18 percent).

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 presents our proposal. In Section 4, we discuss the contributions of deactivating coherence for shared read-only blocks. 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 do not require coherence maintenance. We take advantage of this to propose a mechanism that avoids tracking of these noncoherent blocks. In this section, we comment on works that are some way related to our proposal.

Like us, some authors use the OS to detect private and read-only blocks. Hardavellas et al. [18] use this detection to propose an efficient data placement policy for distributed shared caches (NUCA). Although the mechanism for classifying the pages is similar to ours, they only employ it for data placement in NUCA caches, while, differently, we focus on increasing directory effectiveness. Kim et al. [20] employ OS detection to reduce the fraction of snoops in a token-based protocol by detecting the sharing degree of blocks. Unfortunately, the proposed technique requires large TLBs and important hardware/OS modifications. Furthermore, Kim et al. [20] do not detect shared read-only data pages. Differently, our mechanism is much simpler and does not require complex hardware/OS modifications. In addition, our mechanism takes advantage of the great quantity of private blocks because, although most of the cache misses are for shared blocks, most of the referred blocks are private.

Our proposal can be used to reduce directory size, in particular the number of entries, while maintaining a system performance. Some proposals achieve similar reductions by combining several directory entries into a single one as proposed in [29]. Other approaches use a compressed representation of the sharing information, allowing more compact directory entry formats and reducing storage area requirements, such as the use of a limited number of pointers [1], segment directories [9], chained pointers [8], and coarse vectors [17]. Tagless [32] even removes the need for a conventional directory structure and replaces it with a grid of Bloom filters that provide imprecise sharing information and allow the storage requirements of a conventional directory structure to be to roughly halved. Recently, Cuckoo directory [16] has improved directory efficiency by avoiding most set conflicts without significantly overprovisioning directory capacity. It provides roughly the equivalent of a fully associative directory at the cost of a more complex insertion procedure. However, all these proposals are orthogonal to ours and can be used simultaneously to reduce directory size even more.

Some works remove the unnecessary traffic of broadcast-based protocols by performing coarse-grain tracking of blocks at the expense of increasing storage requirements. Moshovos [26] and Cantin et al. [7] proposed RegionScout filters and Region Coherence Arrays, respectively, which provide different tradeoffs between accuracy and implementation costs. In turn, RegionTracker [31] 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 ours the idea of deactivating the coherence mechanism when it is not indispensable. Nevertheless, 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 to reduce broadcast traffic, but to avoid allocating 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. [12] 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 used to prevent processors not sharing the page from carrying out a tag-lookup in their caches. Enright-Jerger et al. [13] extend the region tracking structure proposed by Zebchuk et al. [31] to keep track of the current set of sharers of a region. Unfortunately, these techniques increase storage requirements and entail large-scale hardware modifications, which make them difficult to be implemented in real systems. Furthermore, our technique does not intend to keep track of the page sharers, but only maintains information about whether the page is shared or not.

Other works also support cache coherence by means of a combination of software and hardware. Zeffer et al. [33] 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.

Finally, Fensch and Cintra [15] propose a coherence protocol that does not require hardware support to enforce cache coherence. Instead, it avoids the possibility of incoherence by not allowing multiple writable shared copies of pages. 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 act indiscriminately on all the referred memory blocks, a significant number of them cannot suffer from inconsistencies. In particular, both the blocks accessed by just one processor (i.e., private blocks) and those that are not written (i.e., read-only blocks) cannot suffer from inconsistencies. The unnecessary use of the directory cache for maintaining the tracking of those blocks increases the overload and makes coherence protocols less effective.

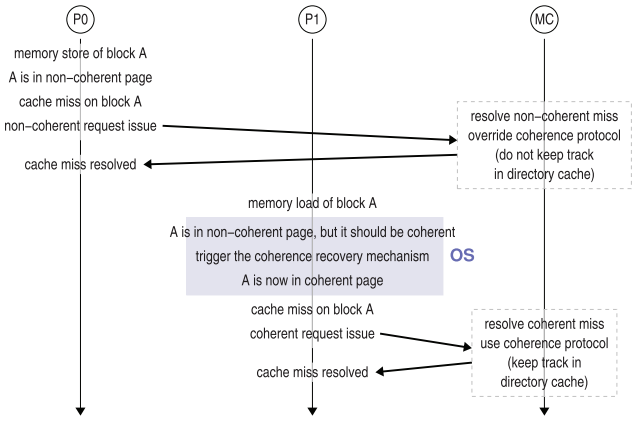


Fig. 2. Overview of the proposed mechanism. *P0* and *P1* are processors and *MC* is the memory controller. The shaded background indicates that the OS is in charge at that moment.

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 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 noncoherent (private or read-only). The cache misses for the blocks belonging to noncoherent pages are resolved without taking into account the coherence protocol. As a result, directory caches do not track the accesses to noncoherent blocks. As the OS detects subsequent memory accesses, 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 page involved. After the recovery process, the page is considered coherent and the memory accesses to its blocks will be tracked by the directory caches.

Fig. 2 outlines our proposal. First, *P0* issues a write operation on memory block A, which causes a cache miss. Assuming that A belongs to a noncoherent page, *P0* issues a noncoherent 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 occurs. While the OS is handling the TLB miss, it realizes that the page should be coherent instead of noncoherent. Consequently, it triggers the coherence recovery mechanism. When it finishes, the page becomes coherent and 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.

The following sections explain our proposal in detail, walking through different key aspects such as page classification (Section 3.1), the behavior of noncoherent requests

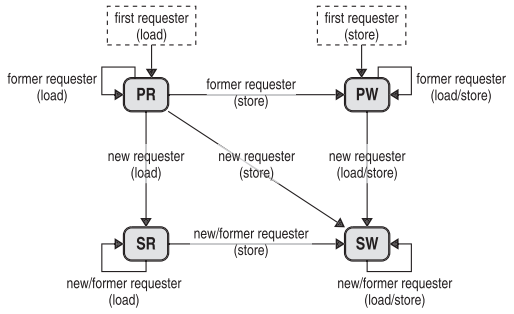


Fig. 3. State transition diagram in new operations view.

(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 into four types:

- **P***rivate* **R***ead-only (PR) page:* Only one processor accesses its blocks. All the accesses are loads.
- **P***rivate read-***W***rite (PW) page:* Only one processor accesses its blocks. At least one of the accesses is a store.
- **S***hared* **R***ead-only (SR) page:* At least two processors access its blocks. All the accesses are loads.
- **S***hared read-***W***rite (SW) 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 may 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 Fig. 3.

3.2 Noncoherent Requests

On memory references, processors first access their TLB to translate virtual addresses into physical addresses. As shown in Fig. 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 with the translation. Since the TLB entry data field often contains some reserved bits that are not used [5], we take advantage of three of them to include two new fields: the *state* field (2 bits), which indicates the page state

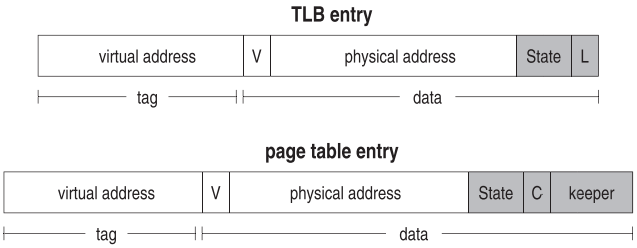


Fig. 4. TLB and page table entry format. Shaded fields are additional fields required by our proposal. V is the valid bit, L is the locked bit, and C is the cached-in-TLB bit.

TABLE 1
Updating the Page Table and TLBs and Use of the TLB-Updating and Coherence Recovery Mechanism

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

r/k/o-TLB stands for the requester/keeper/others' TLB, respectively.

(PR, SR, PW, 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 *noncoherent* request is issued. Otherwise a *coherent* request is sent out. Noncoherent requests override the coherence protocol and are always served by main memory. In addition, directory caches do not track them. This behavior 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 power consumption. Second, directory caches are less occupied and therefore they make better use of their capacity to track blocks that really need coherence. Notice that to instruct memory controllers to understand noncoherent 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 Fig. 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, i.e., whether the page has been cached in any TLB. Notice that these extra fields do not require dedicated hardware, only extra OS storage requirements, which are very small. 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 eight processors, like the AMD Magny-Cours, only six 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 need to be updated as indicated in Table 1. Let us analyze 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

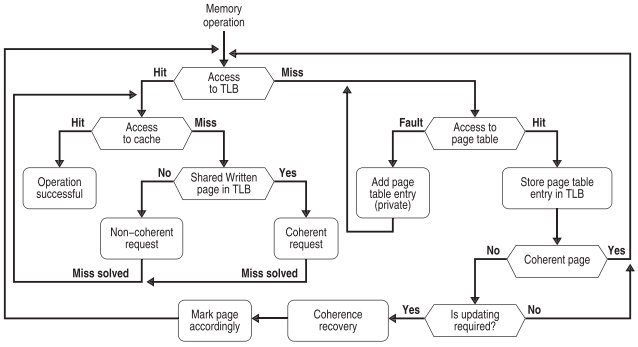


Fig. 5. Block diagram of the general working scheme.

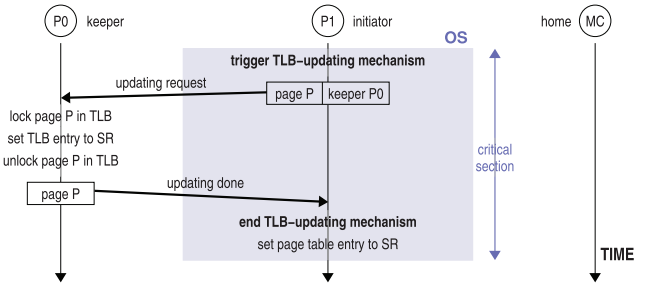


Fig. 6. TLB-updating mechanism. *P0* and *P1* are processors and *MC* is the home node.

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 labeled as PR, upon a load/store from a processor other than the keeper, a TLB 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. If the page is going to transition to SR, although it remains in a noncoherent 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 inconsistence between page table and keeper's TLB has no effect as long as 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 page involved needs to be restored. To this end the coherence recovery mechanism is used. Since nobody keeps the list of sharers, the recovery mechanism will have to perform a broadcast to evict all cached copies of blocks within the corresponding page and update all the sharers' TLBs. This process is also carried out when 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 occur, a special exception (marked^*) is forced to update all the TLBs and the page table. This forced exception will incur 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 not cause any 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.

Fig. 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 noncoherent, the page blocks do not need to be evicted to recover coherence. Fig. 6 shows a detailed example of this mechanism. The mechanism is executed inside a critical section, which guarantees an atomic access to the page table. The initiator (the 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 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 noncoherent 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). After recovery, since the page will be marked as coherent, the directory cache will be able to keep correct track of each of the page 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 store a block within an SR page (as mentioned in Section 3.3), is executed inside a critical section and works as follows:

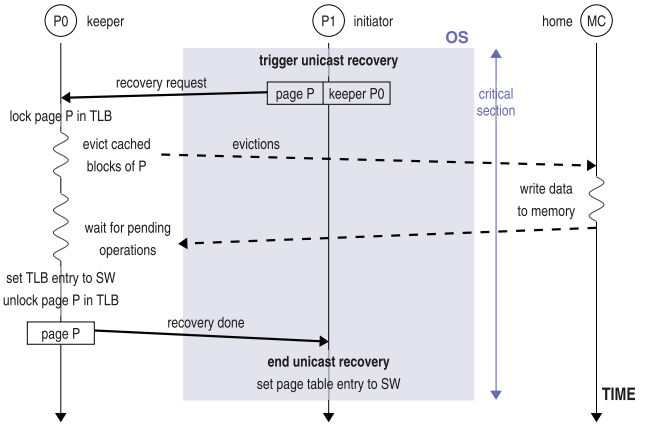


Fig. 7. Coherence recovery mechanism for a private page. *P0* and *P1* are processors and *MC* is the home node.

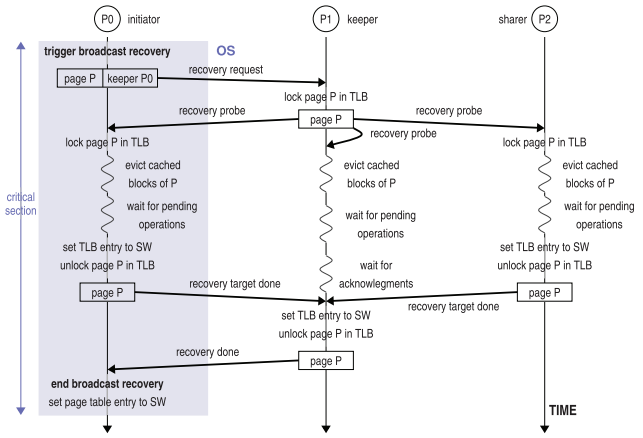


Fig. 8. Coherence recovery mechanism for a shared page. *P0*, *P1*, and *P2* are processors.

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

Second, on arrival of the recovery request, 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 because 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. However, 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 page involved. When finishing they check their Miss Status Holding Registers (MSHRs), 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 page involved are resolved, the corresponding TLB entry is set to SW. After this, the nonkeeper 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 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 page table entry in question and no other processor can access it, so race conditions cannot take place.

Figs. 7 and 8 illustrate the main differences between recoveries for private pages (unicast-based mechanism) and 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 coherent, the directory cache will be able to keep proper track of it.

3.6 Discussion on Coherence Recovery

In this section, we discuss the latency of the coherence recovery mechanism and its 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 observe that the recovery and TLB-updating mechanisms together are triggered less than 5 times per 1,000 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 paper 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 throughout this document. The only difference is that responsibility for 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 SR Blocks

The main difference between the proposal made in this paper and that in [11] is the deactivation of the coherence

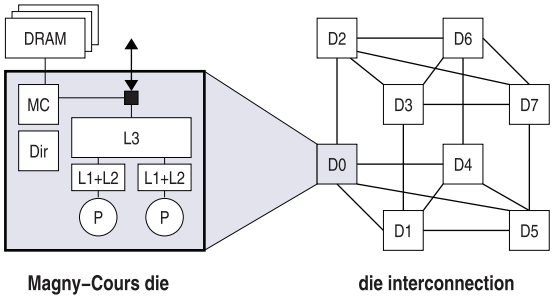


Fig. 9. Overview of the assumed system. *Dir* refers to the directory cache, *MC* to the memory controller, *L3/L2/L1* to the cache of level 3/2/1, and P to processing core.

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 three drawbacks. First, it requires additional resources and a more sophisticated recovery mechanism. In particular, TLBs and page tables need one additional bit to code four states (PR, SR, PW, and SW). Besides, a TLB-updating mechanism and a coherence recovery mechanism based on broadcast are required. They increase the complexity, but they do not have a significant effect on latency or traffic. Second, it can cause additional OS exceptions. These exceptions are used to initiate the coherence recovery mechanism in some cases. Although they are rarely required, they incur considerable delay. And third, it may increase the latency of some misses. Since SR blocks are treated as coherent, cache misses for them will probably be served by the owner processor. However, when we consider them as noncoherent, they will be served from main memory. As a result, their latency may increase. Nevertheless, as we see in Section 6, the advantages of avoiding the tracking of SR blocks outweighs this possible drawback.

On the other hand, the detection and coherence deactivation of SR pages provides many additional advantages that clearly offset its drawbacks (as later analyzed in Section 6). First, 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 greater scalability. Second, the classification of memory pages in PR, SR, PW, 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. And third, it partially addresses the problem that the proposal in [11] 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 coherence cannot be deactivated for them. However, in this proposal the private read-only blocks (more than 40 percent on average according to data in Fig. 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 noncoherent. Notice, though, that this proposal does not tackle the problem of thread migration for PW blocks.

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

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

5 Evaluation Methodology

We evaluate our proposal with full-system simulation using Virtutech Simics [23] running Solaris 10 and extended with the Wisconsin GEMS toolset [24], which enables detailed simulation of multiprocessor systems. For modeling the interconnection network, we have used GARNET [2], a detailed network simulator included in GEMS. Finally, we have also used the McPAT tool [22], assuming a 45 nm process technology, to measure the savings in terms of energy consumption that our proposal can entail.

For the evaluation of our proposal, we have first modeled a cache coherent HyperTransport system optimized with directory caches (PFs) similar to those of the AMD Magny-Cours. As shown in Fig. 9, we simulate eight dies, which constitutes the maximum number of nodes supported by the Magny-Cours protocol. Although each Magny-Cours die actually has six cores, we are only able to simulate two of them due to time constraints. Moreover, since this paper does not focus on the intradie broadcast-based coherence protocol and taking into account that such a protocol would considerably increase the simulation time, we do not model it. 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 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 deactivation of private/SR blocks (*deact P/SR*). The proposal described in [11] is referred to as deactivation of private blocks (*deact P*).

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 three suites (SPLASH-2 [30], ALP-Benchs [21] and PARSEC [6]), two scientific benchmarks, and two commercial workloads [3], 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 [7], [14], [15], 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. In particular, our caches are four times smaller than the ones used by the original Magny-Cours processor. 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 the benchmarks. We account for the variability in multithreaded workloads [4] by doing multiple simulation runs for each benchmark and injecting small random perturbations in the timing of the memory system for each run.

6 Performance Evaluation

In this section, we show how our proposal is able to reduce the number 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 directory cache size. Finally, we analyze the contributions that the coherence deactivation offers when, besides private blocks, it acts on shared read-only memory blocks.

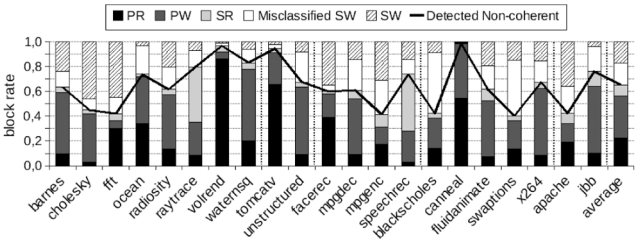


Fig. 10. Block classification on a per memory page basis.

6.1 Noncoherent Blocks

Fig. 10 illustrates the block classification rate using a coarse grain detection based on memory pages. This means that PW pages may contain PR blocks, but they will be considered as PW; SR pages may contain PR blocks, but they will be considered as SR; and SW pages may contain PR/PW/SR blocks, but they will be considered as SW. As the figure shows, about 84 percent (on average) of the referred memory blocks are noncoherent (i.e., PR, SR, or PW) 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 noncoherent blocks. In particular, it detects that 66 percent (on average) of the referred blocks do not require coherence. The remaining 34 percent are classified as coherent blocks and therefore they require coherence. According to these data, the use of a coarse-grain approach causes 18 percent of the referred blocks to be misclassified, which provides a good tradeoff between required resources and accuracy of the mechanism.

6.2 Processor Cache Misses

Since directory caches do not track cached blocks detected as noncoherent, 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 38 percent (on average), as Fig. 11 shows. In this figure, cache misses are classified into 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 misses, which are significantly reduced from about 50 percent of the total misses in the base system to 12 percent (on average). The reduction of

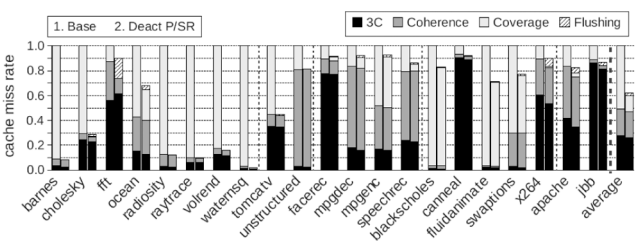


Fig. 11. Normalized cache miss rate.

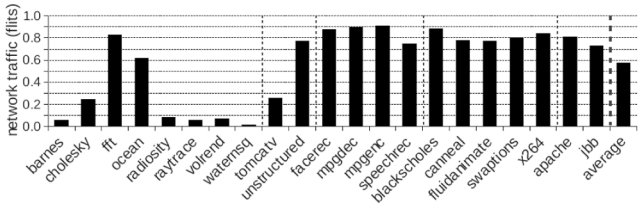


Fig. 12. Normalized network traffic.

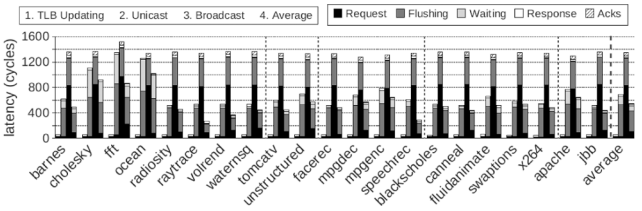


Fig. 14. Average latency of the TLB-updating and recovery mechanisms.

coverage misses depends to a large extent on the accuracy of the detection mechanism. Thus, in applications like barnes, cholesky, and radiosity (among others), few blocks are misclassified as SW. As a result, most of the noncoherent blocks are classified as such and directory caches omit their tracking. This allows directory caches to be less congested, which leads to high coverage miss reductions. This is important because, as reported in other studies [25], [14], 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 becoming increasingly larger.

The blocks initially misclassified as noncoherent are handled by the recovery mechanism. To recover the coherence state of those blocks, the mechanism invalidates them from caches, which may lead to additional misses referred to as flushing misses. Notice that, although the recovery mechanism causes 2 percent (on average) of flushing misses, the reduction in coverage misses is so significant that it largely offsets that increment.

The reduction in both directory evictions and cache misses has a meaningful impact on network traffic, as depicted in Fig. 12. Bars plot the total number of flits transmitted through the interconnection network when the coherence is deactivated and normalized to the network traffic generated by the base protocol. Those data include the traffic due to the coherence recovery mechanism. However, this is not shown separately because the traffic due to the recovery mechanism is really insignificant (lower than 0.5 percent on average). As shown in the figure, the coherence deactivation causes a reduction in network traffic of about 42 percent on average.

Our proposal is able to reduce not only the amount of cache misses but also their average latency, as Fig. 13 depicts. In this figure the latency of cache misses is split into four 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); and finally *response* is the latency from either the sending of the

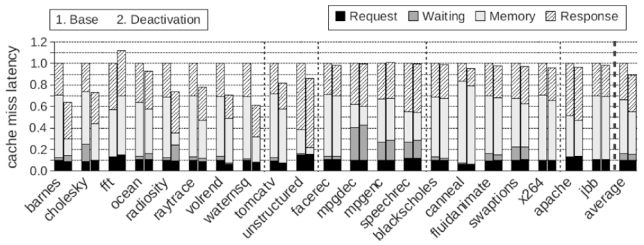


Fig. 13. Normalized cache miss latency.

memory response or the forwarding of the request to another processor until the completion of the miss. Since requests for noncoherent blocks do not need a directory cache lookup, their memory latency is smaller, which lowers the average latency of cache misses by about 10 percent on average. Notice that, for some applications like fft, facerec, 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 noncoherent. Since they are considered noncoherent, they must always be served by main memory, whose latency is high. On the other hand, in the base system SR blocks are considered coherent. Therefore, in some cases cache misses for SR blocks may be served by caches instead of by memory, which is faster. Despite this, the benefits of avoiding the tracking of SR blocks outweighs the increase in their average latency as illustrated in the following sections.

6.3 Coherence Recovery Mechanism

Fig. 14 shows the average latency of both the 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 acknowledgments from home); *response* is the latency of informing 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 can be seen, 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 unicast recoveries mainly due to the need to broadcast the recovery probes and collect the recovery target done messages (acks). The last bar of each group shows the average latency of all these mechanisms taking into account the number of times each 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.

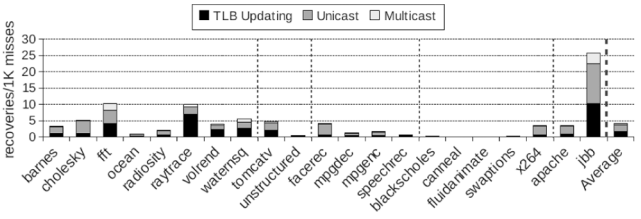


Fig. 15. Number of recoveries and TLB-updating per 1,000 misses.

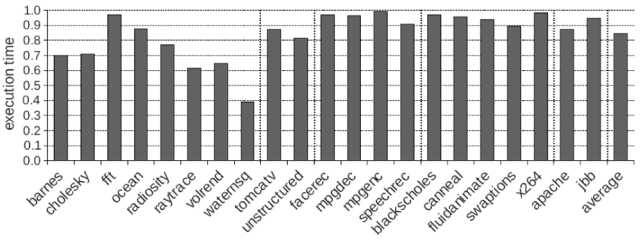


Fig. 16. Normalized runtime of applications.

Despite the fact that the latency of the coherence recovery mechanism can be considerable (mainly in the 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 shown in Fig. 15, on average the mechanisms are only triggered fewer than 5 times per 1,000 cache misses (up to 26 for the jbb application). As a result, their impact on system performance is almost unnoticeable (less than 1 percent on average) compared to the impact that cache misses have on it.

6.4 Execution Time

Mainly due to the reduction in the number of cache misses (and, in some cases, the additional reduction in the miss latency), the runtime of applications can be significantly lower, as depicted in Fig. 16. According to this figure, our proposal improves application runtime by 15 percent on average. For applications where both cache miss rate and latency are significantly reduced (barnes, cholesky, and watersnq among others), the system performance improves considerably. However, for applications where the reduction in cache misses is not so significant, the improvements in performance are more moderate.

6.5 Impact of Coherence Deactivation for SR Blocks

In this section, we evaluate the implications that the deactivation of private/SR blocks (*deact P/SR*) have in comparison to the deactivation of just private blocks (*deact P*).

Fig. 17 illustrates the potential provided by *deact P/SR* with respect to that of *deact P*. The total value of bars indicates the rate of actual noncoherent blocks. Each bar is divided into detected noncoherent blocks and undetected noncoherent blocks (i.e., blocks misclassified as SW). As can be seen, in applications like raytrace, mpgenc or speechrec, *deact P/SR* substantially increases the number of blocks that can actually be detected as noncoherent from 53, 48, and 39 percent to 91, 70, and 85 percent, respectively. Hence, the detection of SR blocks allows the mechanism to increase the number of noncoherent blocks and, therefore, to reduce the number of tracked blocks. On average *deact P/SR* detects

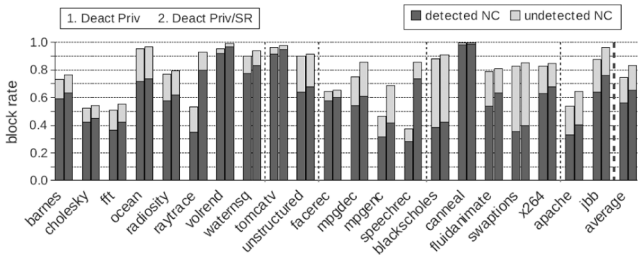


Fig. 17. Potential and precision of *deact P* and *deact P/SR*.

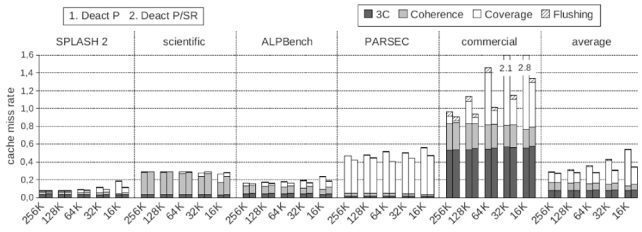


Fig. 18. Cache miss rate of *deact P* and *deact P/SR* in systems with 256K, 128K, 64K, 32K, and 16K directory caches.

about 9 percent more noncoherent blocks than *deact P*. Furthermore, the percentage of blocks that are misclassified as SW (undetected NC) is slightly reduced from 33 percent in *deact P* to 27 percent in *deact P/SR*, which indicates that *deact P/SR* improves the accuracy of the detection mechanism.

Since *deact P* is effective enough in removing almost all the coverage misses of most of the scenarios evaluated (see Fig. 11), the application of *deact P/SR* on those scenarios only leads to small improvements. However, as the size of directory caches is smaller, the effectiveness of *deact P* decreases and the benefits of *deact P/SR* are more visible. Hence, the systems where storage requirements are critical may need to use smaller directory caches and, consequently, in those systems the deactivation of SR blocks may be vital. To illustrate this, in this section we compare *deact P* against *deact P/SR* in systems using directory caches with sizes ranging from 256K to 32K. For the sake of clarity, the following graphs only show the average latency of the applications grouped in application suites and the average value for all the simulated applications.

Fig. 18 shows the cache miss rate of *deact P* and *deact P/SR*. As shown, as the directory cache size gets smaller, 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 P*, the cache miss rate grows much more quickly than when *deact P/SR* is used. This happens because, since the potential of *deact P/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 in the directory cache size affects less coherent blocks, which leads to slower growth in the cache miss rate.

Fig. 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 P* is slightly smaller than that of *deact P/SR* mainly because *deact P* considers SR blocks as coherent and therefore misses for SR blocks may be served by caches instead of memory. As the directory cache size

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

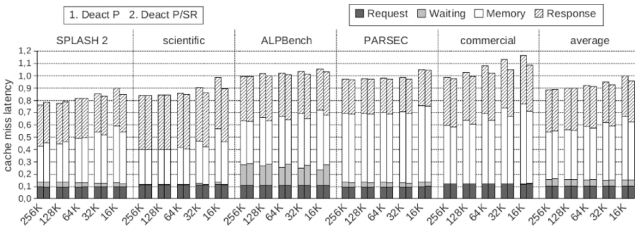


Fig. 19. Cache miss latency of *deact P* and *deact P/SR* in systems with 256K, 128K, 64K, 32K, and 16K directory caches.

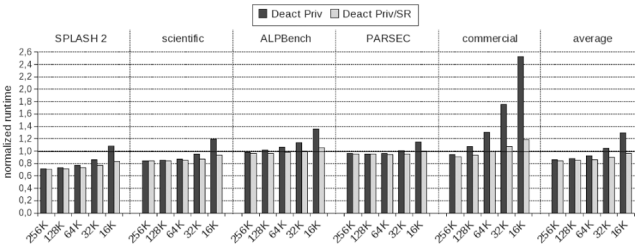


Fig. 20. Normalized runtime of applications for *deact P* and *deact P/SR* in systems with 256K, 128K, 64K, 32K and 16K directory caches.

decreases, the number of coverage misses increases due to the increase in evictions of cached blocks (mainly SR and SW blocks). Thus in *deact P*, 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 coherent, their memory latency includes access to directory caches, which makes the average latency of those cache misses increase. However, when using *deact P/SR*, SR blocks are not evicted from caches due to lack of space in the directory. Furthermore, to resolve the misses for those blocks, an access to the directory cache is not required, which makes the average miss latency increase more slowly. Hence, although in systems with 256K directory caches the average miss latency of *deact P* is smaller than that of *deact P/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 P/SR* becomes smaller than that of *deact P*.

Fig. 20 illustrates how the execution time of applications varies according to directory cache size. As observed, *deact P/SR* gets better results when the size of directory caches is really small compared to the working set of applications. Thus, in systems with 256K directory caches, *deact P/SR* only gets a slight improvement over *deact P* because in that specific scenario *deact P* is already able to avoid most of the coverage misses.[2] As directory caches are reduced, the differences between *deact P/SR* and *deact P* are more significant. In particular, *deact P/SR* achieves to reduce up to 16 times the directory cache size while maintaining application runtime below that of the base system. However, in the case of *deact P*, the directory cache can only be reduced eight times and, even then, the runtime slightly increases with respect to that of the base system. For a system with 16K directory caches, *deact P/SR* reduces the runtime by about 30 percent (on average) with respect to *deact P*. Hence, as directory caches are smaller, *deact P/SR* offers better performance than *deact P*, which indicates it provides higher scalability. Notice that directory caches consume precious on-chip area.

In order to emphasize the relevance of this result, let us point out the fact that recent proposals to reduce the directory size by using different approaches, such as Tagless [32], are only able to reduce the area requirements by 48 percent with respect to a conventional sparse directory while maintaining performance, whereas our proposal is able to reduce it by 87 percent (when using *DeactP*) or even by 94 percent (when using *Deact P/SR*).

2. Although not shown in the figure, *deact P/SR* outperforms *deact P* by a 4.5 percent on average for FFT, Ocean, SpeechRec and Apache.

6.6 Scalability

Deact P/SR scales with the core count, which is illustrated by Fig. 21. Due to the slowness of the simulation tools used, it is unfeasible to simulate benchmarks such as ALPBench or PARSEC. Therefore, to make an analysis with a higher core count possible, data in Fig. 21 are obtained by averaging the data for the SPLASH 2 benchmarks and the scientific applications. Fig. 21a illustrates that the accuracy of our proposal in detecting noncoherent blocks does not depend on core count, as the rate of detected noncoherent blocks is kept more or less constant. As a result, the influence on runtime also remains constant, as shown in Fig. 21b.

6.7 Impact of Memory Page Size

The block classification mechanism employed by our proposal is based on memory pages and therefore its accuracy depends on the page size. Fig. 22a shows the accuracy of the classification mechanism (in *deact PR/SR*) for different page sizes (from 4KB to 32KB). As expected, the larger page size is, the less accurate this mechanism is. On average, when pages are 4K, our mechanism detects that about 65 percent of the accessed blocks do not require coherence. However, when pages are 32K, it detects that about 50 percent of the accessed blocks do not require it. Despite this loss of accuracy, our mechanism is still able to detect a large number of the blocks that do not require coherence. As a result, most of the coverage misses are avoided, as Fig. 22b illustrates. Consequently the final impact of page size in overall performance is quite low, with performance being more or less constant, as can be seen in Fig. 22c.

6.8 Energy Consumption

Thanks to the reduction in cache misses and network traffic, our proposal is also able to reduce system energy consumption. Fig. 23 shows the dynamic energy consumption of directory caches, memory controllers, and the interconnection network. Since noncoherent requests do not need to

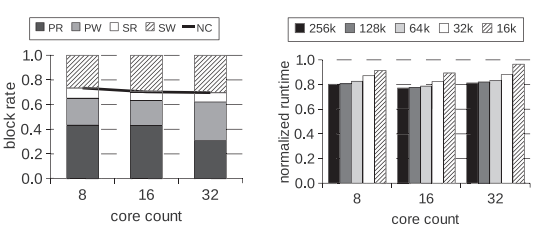


Fig. 21. Scalability analysis of *deact P/SR* for SPLASH 2 and scientific applications varying core count. (a) Block classification and (b) normalized runtime.

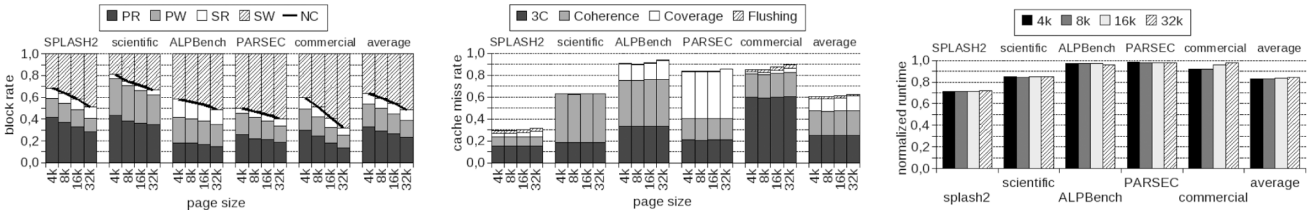


Fig. 22. Impact of page size on (a) block classification, (b) cache miss rate, and (c) runtime.

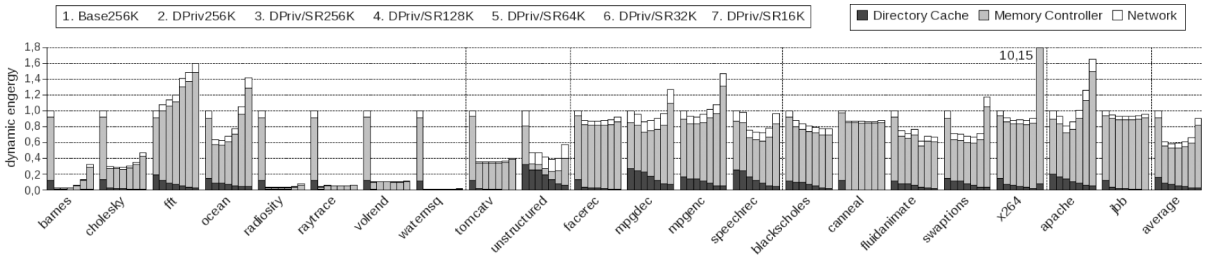


Fig. 23. Energy consumption.

access directory caches, their consumption is smaller. Furthermore, this reduction becomes more significant 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 this increase in accesses.

Although our proposal decreases the number of memory accesses (due to the cache miss reduction, as shown in Fig. 11), the recovery mechanism may increase it (due to the eviction of cached blocks). However, on average the reduction referred to offsets this increase. As a result the energy consumption of memory controllers is reduced by 45 percent 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. However, as directory caches become smaller, the number of cache misses increases and, consequently, more accesses to memory controllers will be required, thereby increasing 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 Fig. 12. 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 percent on average. As directory caches become smaller, energy consumption increases mainly due to the increase in accesses to memory controllers. Despite this, the dynamic energy consumption of a system using our proposal remains lower (5 percent on average) than that of the *base* system using directory caches 16 times larger.

Regarding static energy consumption (not shown in Fig. 23), this is closely linked 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, static power consumption is reduced by 48, 74, 86, and 92 percent, respectively.

7 Conclusions

In this paper, we propose a simple strategy 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 (i.e., private and read-only memory blocks). The OS is responsible for dynamically classifying the accessed blocks according to a coarse-grain strategy. Our proposal increases the number of available entries in directory caches. As a result the number of blocks invalidated due to the lack of entries in directory caches can be drastically reduced. This advantage can be used not only for increasing system performance (15 percent), but also for obtaining good performance with less storage requirements (directory caches 16 times smaller). The latter achievement is very useful for coping with the silicon area constraints arisen in the design of many-core chips.

Acknowledgments

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

References

[1] A. Agarwal, R. Simoni, J.L. Hennessy, and M.A. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence," *Proc. 15th Int'l Symp. Computer Architecture (ISCA)*, pp. 280-289, May 1988.

[2] N. Agarwal, T. Krishna, L.-S. Peh, and N.K. Jha, "GARNET: A Detailed On-Chip Network Model Inside a Full-System Simulator," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS)*, pp. 33-42, Apr. 2009.

[3] 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," *Proc. Fifth Workshop Computer Architecture Evaluation using Commercial Workloads (CAECW)*, pp. 30-38, Feb. 2002.

[4] A.R. Alameldeen and D.A. Wood, "Variability in Architectural Simulations of Multi-Threaded Workloads," *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA)*, pp. 7-18, Feb. 2003.

[5] "AMD64 Architecture Programmer's Manual Volume 2: System Programming," white paper, Advanced Micro Devices, 2010.

[6] C. Bienia, S. Kumar, J.P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *Proc. 17th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* pp. 72-81, Oct. 2008.

[7] J.F. Cantin, M.H. Lipasti, and J.E. Smith, "Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking," *Proc. Int'l Symp. Computer Architecture (ISCA),* pp. 246-257, June 2005.

[8] G. Chen, "Slid - A Cost-Effective and Scalable Limited-Directory Scheme for Cache Coherence," *Proc. Fifth Int'l PARLE Conf. Parallel Architectures and Languages Europe (PARLE '93),* 1993.

[9] J.H. Choi and K.H. Park, "Segment Directory Enhancing the Limited Directory Cache Coherence Schemes," *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS),* pp. 258-267, Apr. 1999.

[10] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *IEEE Micro,* vol. 30, no. 2, pp. 16-29, Apr. 2010.

[11] 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," *Proc. 38th Int'l Symp. Computer Architecture (ISCA),* June 2011.

[12] M. Ekman, F. Dahlgren, and P. Stenström, "TLB and Snoop Energy-Reduction Using Virtual Caches," *Proc. Int'l Symp. Low Power Electronics and Design (ISLPED),* pp. 243-246, Aug. 2002.

[13] N.D. Enright-Jerger, L.-S. Peh, and M.H. Lipasti, "Virtual Circuit Tree Multicasting: A Case for On-Chip Hardware Multicast Support," *Proc. 35th Int'l Symp. Computer Architecture (ISCA),* pp. 229-240, June 2008.

[14] 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," *Proc. IEEE/ACM 41th Int'l Symp. Microarchitecture (MICRO),* pp. 35-46, Nov. 2008.

[15] C. Fensch and M. Cintra, "An OS-Based Alternative to Full Hardware Coherence on Tiled CMPs," *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA),* pp. 355-366, Feb. 2008.

[16] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo Directory: A Scalable Directory for Many-Core Systems," *Proc. 17th Int'l Symp. High-Performance Computer Architecture (HPCA),* pp. 169-180, Feb. 2011.

[17] A. Gupta, W.-D. Weber, and T.C. Mowry, "Reducing Memory Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," *Proc. Int'l Conf. Parallel Processing (ICPP),* pp. 312-321, Aug. 1990.

[18] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," *Proc. 36th Int'l Symp. Computer Architecture (ISCA),* pp. 184-195, June 2009.

[19] R. Kalla, B. Sinharoy, W.J. Starke, and M. Floyd, "POWER7: IBM's Next-Generation Server Processor," *IEEE Micro,* vol. 30, no. 2, pp. 7-15, Apr. 2010.

[20] D. Kim, J. Ahn, J. Kim, and J. Huh, "Subspace Snooping: Filtering Snoops with Operating System Suport," *Proc. 19th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* pp. 111-122, Sept. 2010.

[21] M.-L. Li, R. Sasanka, S.V. Adve, Y.-K. Chen, and E. Debes, "The ALPBench Benchmark Suite for Complex Multimedia Applications," *Proc. Int'l Symp. Workload Characterization,* pp. 34-45, Oct. 2005.

[22] 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," *Proc. IEEE/ACM 42nd Int'l Symp. Microarchitecture (MICRO),* pp. 469-480, Dec. 2009.

[23] P.S. Magnusson, M. Christensson, and J. Eskilson et al., "Simics: A Full System Simulation Platform," *Computer,* vol. 35, no. 2, pp. 50-58, Feb. 2002.

[24] M.M. Martin, D.J. Sorin, and B.M. Beckmann et al., "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset," *Computer Architecture News,* vol. 33, no. 4, pp. 92-99, Sept. 2005.

[25] M.R. Marty and M.D. Hill, "Virtual Hierarchies to Support Server Consolidation," *Proc. 34th Int'l Symp. Computer Architecture (ISCA),* pp. 46-56, June 2007.

[26] A. Moshovos, "RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence," *Proc. 32nd Int'l Symp. Computer Architecture (ISCA),* pp. 234-245, June 2005.

[27] B.W. O'Krafka and A.R. Newton, "An Empirical Evaluation of Two Memory-Efficient Directory Methods," *Proc. 17th Int'l Symp. Computer Architecture (ISCA),* pp. 138-147, June 1990.

[28] M. Shah, J. Barreh, and J. Brooks et al., "UltraSPARC T2: A Highly-Threaded, Power-Efficient, SPARC SoC," *Proc. IEEE Asian Solid-State Circuits Conf.,* pp. 22-25, Nov. 2007.

[29] R. Simoni, "Cache Coherence Directories for Scalable Multiprocessors," PhD thesis, Stanford Univ., 1992.

[30] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA),* pp. 24-36, June 1995.

[31] J. Zebchuk, E. Safi, and A. Moshovos, "A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy," *Proc. IEEE/ACM 40th Int'l Symp. Microarchitecture (MICRO),* pp. 314-327, Dec. 2007.

[32] J. Zebchuk, V. Srinivasan, M.K. Qureshi, and A. Moshovos, "A Tagless Coherence Directory," *Proc. IEEE/ACM 42nd Int'l Symp. Microarchitecture (MICRO),* pp. 423-434, Dec. 2009.

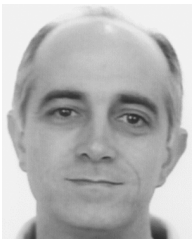
[33] H. Zeffer, Z. Radović, M. Karlsson, and E. Hagersten, "TMA: A Trap-Based Memory Architecture," *Proc. 20th Int'l Conf. Supercomputing (ICS),* pp. 259-268, June 2006.

Blas Cuesta received the MS and PhD degrees in computer science from the Universitat Politècnica de València, Spain, in 2002 and 2009, respectively. From 2004 to 2011, he has been with the Parallel Architecture Group (GAP) in the Department of Computer Engineering at the same university, working in the design and evaluation of scalable coherence protocols for shared-memory multiprocessors. Since 2011, he has been with the Intel Labs Barcelona. His research interests include cache coherence protocols, memory hierarchy designs, scalable cc-NUMA and chip multiprocessor architectures, and interconnection networks.

Alberto Ros received the MS and PhD degrees in computer science from the Universidad de Murcia, Spain, in 2004 and 2009, respectively. In 2005, he joined the Computer Engineering Department at the same university as a PhD student with a fellowship from the Spanish government. Since 2009, he has been working as a researcher at the Parallel Architecture Group (GAP) of the Universitat Politècnica de València. He is working on designing and evaluating scalable cache coherence protocols for shared-memory multiprocessors. His research interests include cache coherence protocols, memory hierarchy designs, and scalable multiprocessor architectures. He is a member of the IEEE.

María E. Gómez received the MS and PhD degrees in computer science from the Universitat Politècnica de València, Spain, in 1996 and 2000, respectively. She joined the Department of Computer Engineering (DISCA) at Universitat Politècnica de València in 1996 where she is currently an associate professor of computer architecture and technology. Her research interests are in the field of interconnection networks, networks-on-chips, and cache coherence protocols. She is a member of the IEEE Computer Society.

Antonio Robles received the MS degree in physics (electricity and electronics) from the Universitat de València, Spain, in 1984 and the PhD degree in computer engineering from the Universitat Politècnica de València in 1995. He is currently a full professor in the Department of Computer Engineering at the Universitat Politècnica de València. He has taught several courses on computer organization and architecture. His research interests include high-performance interconnection networks for multiprocessor systems and clusters and scalable cache coherence protocols for SMP and CMP. He has published more than 70 refereed conference and journal papers. He has served on program committees for several major conferences. He is a member of the IEEE Computer Society.



José Duato received the MS and PhD degrees in electrical engineering from the Universitat Politècnica de València, Spain, in 1981 and 1985, respectively. He is currently a professor in the Department of Computer Engineering at the Universitat Politècnica de València. He was an adjunct professor in the Department of Computer and Information Science at The Ohio State University, Columbus. His research interests include interconnection networks and multiprocessor architectures. He has published more than 380 refereed papers. He proposed a powerful theory of deadlock-free adaptive routing for wormhole networks. Versions of this have been used in the design of the routing algorithms for the MIT Reliable Router, the Cray T3E supercomputer, the internal router of the Alpha 21364 microprocessor, and the IBM BlueGene/L supercomputer. He is the first author of the *Interconnection Networks: An Engineering Approach* (Morgan Kaufmann, 2002). He was a member of the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, and *IEEE Computer Architecture Letters*. He was a cochair, member of the steering committee, vice chair, or member of the program committee in more than 55 conferences, including the most prestigious conferences in his area of interest: HPCA, ISCA, IPPS/SPDP, IPDPS, ICPP, ICDCS, EuroPar, and HiPC. He has been awarded with the National Research Prize *Julio Rey Pastor 2009*, in the area of Mathematics and Information and Communications Technology and the *Rei Jaume I Award on New Technologies 2006*.