Efficient TLB-Based Detection of Private Pages in Chip Multiprocessors

Albert Esteve, Alberto Ros, María E. Gómez, Antonio Robles, *Member, IEEE,* and José Duato, *Member, IEEE*

Abstract—Most of the data referenced by sequential and parallel applications running in current chip multiprocessors are referenced by a single thread, i.e., private. Recent proposals leverage this observation to improve many aspects of chip multiprocessors, such as reducing coherence overhead or the access latency to distributed caches. The effectiveness of those proposals depends to a large extent on the amount of detected private data. However, the mechanisms proposed so far do not consider neither thread migration nor the private use of data within different application phases. As a result, a considerable amount of private data is not detected. In order to increase the detection of private data, we propose a TLB-based mechanism that is able to account for both thread migration and application phases. Simulation results show that the average number of pages detected as private significantly increases from 43% in previous proposals up to 79% in ours while keeping a reasonable TLB miss rate. Furthermore, when our proposal is used to deactivate the coherence for private data in a directory protocol, it improves execution time by 13.5%, on average, with respect to previous techniques.

Index Terms—Multiprocessor, cache coherence, directory cache, coherence deactivation, TLB decay

1 Introduction and Motivation

During the last few years, most high-performance processors have been built using chip-multiprocessors (CMPs). Core count on these CMPs is rapidly growing according to Moore's Law stipulation, which, along with the implementation of a shared-memory programming model, leads to the requirement of efficient coherence maintenance among data in private caches. These circumstances bring new challenges to make coherence protocols scalable and to provide increasing performance.

On small core-count CMPs a snoop protocol can be used, as the bandwidth requirements for these approaches grows dramatically with the number of cores. There are many different approaches whose main goal is to reduce the traffic generated by broadcast messages when snooping [1], [2], but scaling to larger systems is still troublesome.

On the other hand, directory approaches require less network bandwidth, and are more scalable, but they introduce a directory structure whose on-chip area and leakage power grows exponentially with the number of cores. Due to the limited directory capacity or associativity, this causes a larger amount of directory-induced invalidations of cached blocks as a results of directory replacements. Therefore, conventional directory protocols are not suited for large core-count CMPs and some alternate proposals addressing the scalability problem could be used instead [3], [4], [5].

Recently, a different approach relying on classify accessed data into a private-shared scheme has been used to improve coherence management. Data is classified as private or shared based on whether it is accessed by just one thread, or by two or more threads respectively. This approach benefits from the observation that data referenced by sequential and parallel applications during their execution time is mostly private, thus it can be handled more efficiently in terms of performance and scalability.

Past proposals used data classification to address aforementioned shortcomings on CMPs. Specifically, Kim *et al.* [6] avoid issuing broadcast messages when accessing private blocks, thus leading to network traffic reductions in broadcast-based protocols. Alternatively, Cuesta *et al.* [7] propose to deactivate coherence for data not requiring it, which prevents directory caches from tracking private blocks, thus reducing both directory occupancy and access latency.

Additionally, the organization of the last-level cache (LLC) on a CMP can also be improved through data classification. Different papers effectively address the access latency growth caused by the use of a *NUCA* (Non-uniform Cache Architecture) organization as core count increases [8]. Alternatively, Hardavellas *et al.* [9] and Li *et al.* [10], [11] keep private blocks in the NUCA bank of the requesting processor to reduce the access latency to NUCA caches.

This paper, which is an extension of the approach provided by Ros *et al.* [12], proposes a mechanism that is capable of classifying data into a private-shared scheme while avoiding some of the major problems encountered on previous outlined techniques. Firstly, our mechanism relies on the TLB to keep the sharing status information, thus reducing storage requirements as it does not demand extra information in a directory-like structure [13], [14], [15] or in the page table [6], [7], [9], [16], [17].

Moreover, it exploits proximity of cores in a CMP in

- *A. Esteve, M.E. Gómez, A. Robles and J. Duato are with Department of Computer Engineering, Universitat Politècnica de València. E-mail: alesgar@gap.upv.es, {megomez,arobles,jduato}@disca.upv.es*
- *A. Ros is with Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia. E-mail: aros@ditec.um.es*

two different ways. Upon a TLB miss, our mechanism scrutinizes the information related to the use of pages from the TLBs of other cores, which allows them to get the sharing information along with the address translation, which accelerates the page translation process.

Lastly, our mechanism accounts for temporarily private pages (e.g., as a consequence of thread migration) by predicting for each core whether it will use the page in a near future or not. Since data access patterns change throughout different phases of the application lifetime [18], [19], we claim in this work that a temporal-aware detection mechanism is fundamental to achieve a good accuracy when detecting private pages. Note that the more private data detected, the more benefits can be obtained when using a classification scheme.

This paper extends [12] by proposing a novel access prediction mechanism to avoid premature page invalidations of TLB entries and cache blocks when applying decay techniques. In [12], low decay values cause so many invalidations that makes the high detection of private blocks unattractive. However, in this paper, we force pages migrating frequently from a core to another to become shared. Although this slightly decreases the proportion of private pages detected, the reduction in TLB and cache misses mitigates the performance degradation, thus making the use of a low decay value appealing for the first time. On the other hand, the replacement algorithm on TLBs has been improved, employing a modified LRU policy where invalidated lines are prioritized when evicting a page.

A large variety of scientific and commercial workloads have been run over a 16-core CMP cycle-accurate simulator, showing how our proposal effectively increases the average number of pages classified as private from 43% (with previous OS-based mechanisms) up to 79%. Additionally, we show how the proposed prediction mechanism is able to reduce the drawbacks of premature page invalidations, reducing the TLB misses due to the use of *Decay* techniques up to 4.73 times and forced L1 cache misses from up to 23.1% to just 4% on average, dramatically diminishing the execution time loss with low decay values. Furthermore, applying it to improve the efficiency of directory caches, we effectively scale down directory cache storage requirements up to 50%, while still reducing the execution time 7% compared to previous proposals.

The rest of the paper is organized as follows. In Section 2 we review the private-shared classification mechanisms. Section 3 shows the potential of an accurate classification mechanism motivating the techniques developed in this paper, through the analysis of some key aspects on TLB entries behavior. Section 4 describes the TLB-to-TLB technique employed in this paper to detect private pages and accelerate TLB misses. The proposed temporal-aware mechanism is presented in Section 5, and some details of its usage are given. Section 6 introduces our simulation methodology and Section 7 shows the performance results. Section 8 discusses other system configurations. Finally, Section 9 reviews the related work, and Section 10 draws some conclusions.

2 Background

Among the different mechanisms used to detect private accesses, we compare ours against those aided by the OS [6], [7], [9], [17]. Unlike hardware-based approaches [13], [14], OS-based mechanisms do not require additional hardware support because they take advantage of existing OS structures (i.e., page table and TLBs). On the other hand, compiler-assisted approaches [10], [11] face the difficulty of knowing at compile time (1) whether a variable is going to be shared or not and (2) in which core the processes and threads will be scheduled and rescheduled. The OS-based detection avoids these difficulties and provides a more accurate run-time mechanism.

An OS-based classification considers pages as private the first time they are accessed after a TLB miss, annotating the requesting core in the page table (keeper [7] or FAC -first accessing core- [17] field). When a page table entry is accessed, the keeper field is compared with the current requestor. If the keeper field does not match on a private page table entry, then it is re-classified as shared. To this end, each page table entry adds a P/S bit that indicates the page state (private or shared) and the aforementioned field containing the identifier of the first core that accessed the page. The P/S bit is also included in the TLB entries to allow a fast access to the page state for those cores that have the page entry in the TLB.

When a page changes from private to shared, the core having the page as private must be notified in order to update its TLB accordingly. As we explain in the following sections, depending on the use of the classification technique it may be necessary to perform other actions such as invalidating every block belonging to the page cached at the core accessing the page privately. Otherwise, coherence problems might arise.

2.1 Improving directory cache effectiveness

Recent directory-based protocols only keep directory information for a small fraction of the memory pages (those having at least one block cached) in small directory caches with the aim of reducing the memory overhead [20]. But, due to the lack of a backup directory, the eviction of an entry from the directory cache entails the invalidation of every cached copy of the block tracked by the entry. Since the size of directory caches is quite limited, they can suffer frequent evictions and, consequently, data caches may exhibit high miss rates due to these evictions, which results in serious performance degradation.

A private-shared classification mechanism can improve the effectiveness of directory caches by not tracking private blocks since they do not require coherence maintenance, as proposed in the *Coherence Deactivation* approach [7]. The proposal improves the availability of directory entries for the blocks that really need coherence (i.e., shared blocks) and exploits more efficiently the limited directory capacity.

This mechanism requires restoring the coherence state when a page transitions from private to shared since blocks in that page, which are cached by a single core,

are not tracked by the directory. In particular, all caches blocks within the page becoming shared must be evicted.

2.2 Reducing NUCA access latency

The organization of the LLC in a many-core CMP can be either private or shared. A private organization achieves low access-latency while a shared organization offers large storage capacity. Although a shared LLC organization (or NUCA cache) is more common, its average access latency increases with the number of cores in the system.

Again, a private-shared classification can reduce the NUCA access latency, as described in the *Reactive NUCA* proposal [9]. Private blocks are placed into the local NUCA bank of the requesting core, enabling low-latency accesses for such blocks, while shared blocks are placed across all tiles at the corresponding address-interleaved locations.

When a page changes from private to shared, every block belonging to that page that is cached in the core accessing the page, either in the L1 cache or in the local LLC bank, is evicted to avoid duplicated and incoherent data in the CMP.

2.3 Reducing traffic in broadcast-based protocols

Broadcast-based protocols offer low-cost and simple coherence for small-scale systems. However, the required broadcast traffic not only consumes an important amount of power but also prevents such protocols from being used in large-scale systems.

The scalability of snooping protocols can also be improved with a private-shared classification, as proposed by the *Subspace Snooping* approach for token-based protocols [6]. Since there are not copies of blocks belonging to private pages, cache-misses due to accesses to blocks within private pages can be resolved without broadcasting requests to all the nodes in the system, thus reducing unnecessary snoops.

In this case, when a page changes from private to shared, no action is required. Just updating the page as shared in the page table and in the TLB makes that successive cache-misses will be broadcast, discovering in a natural way the existence of other cached copies.

2.4 Building complexity-effective protocols

Cache coherence protocols require a number of base states to keep coherence of the blocks in cache. In order to improve efficiency, more states are added, which cause an explosion in the number of transient states to transition from a base state to another, and consequently, in complexity.

The private-shared classification can be used to build simple while efficient protocols as proposed in VIPS [16]. VIPS employs an efficient write-back policy for private blocks, which are critical for application performance, and a simple write-through policy with just two base states (VI) for shared blocks, thus reducing the number of race conditions that can occur.

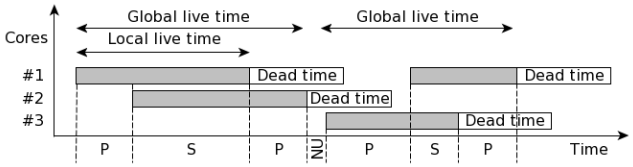


Fig. 1: Generation time idealization

Upon a private-to-shared page change, every block within the page has to be written-back to the shared cache.

3 Potential Exploitation of the Temporal Classification

A page-based data classification mechanism is intended to classify pages either as *Private* or as *Shared*. With the aim of performing a temporal-aware page classification, in this paper we rely on looking up the core's TLBs to check the presence of a certain page. TLB looking-up allows us to dynamically classify pages along time as private or shared depending on whether the page translation resides in a single TLB or, on the contrary, it is stored in two or more TLBs at a particular time instant. As a result, a temporal-aware page classification mechanism i.e., the usage in time of the memory pages, is provided. In order to design this temporal-aware classification mechanism we first analyze page sharing patterns along time. For doing this, we define the concept of *page generation time* in a core as the time spent since a page is first accessed (and missed) on a core's TLB until the page is finally evicted from that TLB. A page may have several generation times on the same core or in different cores. Notice also that the page generation times of different cores may overlap or not along time, which will decide whether the page at a certain point in time is private (do not overlap) or shared. So, a page could be classified initially as private, later as shared, then back to private and so on, depending on the overlapping of its generation times at different cores. We thus, define the *global generation time* as the elapsed time from a page is first cached on a TLB to the moment it is evicted from the last TLB in the system. This means that the global generation time expands along the overlapped local or individual generation times in cores. For private page generations, local and global generation times are equivalent.

However, this definition used to classify pages is just an approximation to the ideal. The sharing condition of a page in the system should be settled by the simultaneity of accesses to that page. If a page in a core TLB will not be longer accessed again, it will count as present in order to determine its sharing status, and therefore it will result in an inaccurate classification. As we are interested in analyzing the potentiality of an ideal page-based classification mechanism, page *dead time* (time from the last access to a page until its eviction in a core TLB) is excluded, therefore accounting only for the page *live time* (elapsed time since the page is fetched until its last access before being replaced in a core TLB) [21], i.e.

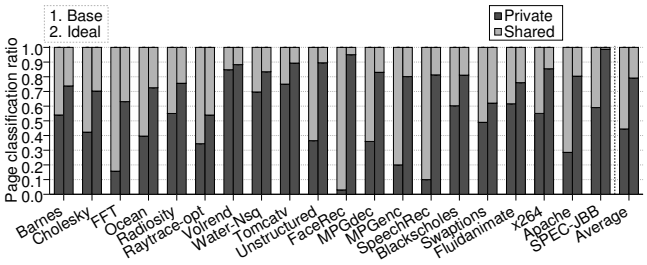


Fig. 2: Idealized page classification comparison

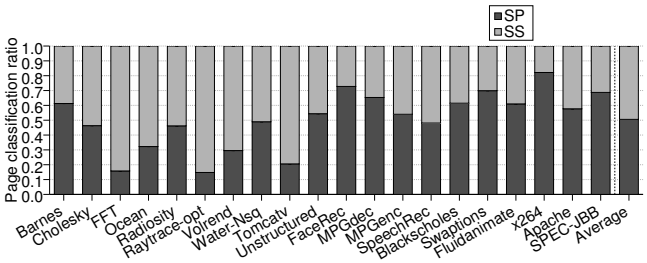


Fig. 3: Shared lives classification

the gray bar on Figure 1. This will allow us to determine to what extend this classification imprecision could affect to the effectiveness of the proposed page classification mechanism.

3.1 Classification Analysis

We compare the page classification provided by the baseline system, where the OS is used to detect private and shared pages, against the classification obtained from the ideal mechanism defined above. As the status of a page varies along time, to simplify the analysis we display a page as shared if at some point in time it has been classified as shared. In its turn, a page displayed as private has been always classified as private along time. That means that its live times on different cores never overlap. In other words, a page could live in different cores and remain private, as long as there is a strict exclusion on their live times. Notice that a particular case would be one in which the different lives of the page always elapse on the same core. On the contrary, the page will be classified as shared when its live times overlap at some point in time, despite the fact that its status may be private for most of the execution time.

Figure 2 shows the aforementioned comparison for both classification mechanisms: the *Base* bar represents the static classification mechanism performed by the OS, and the *Ideal* bar represents the page classification obtained when idealizing the generation time of a page, thus excluding the dead time. These results are obtained from the simulation environment presented in Section 6. As can be seen, the *Ideal* mechanism is able to classify as private, on average, 35% more pages than the *Base* classification for a total amount of nearly 80% of pages.

In order to analyze the potential of a dynamic and temporal-aware classification approach in which pages could be reclassified as private once after they have been classified as shared, next we quantify the number of lives

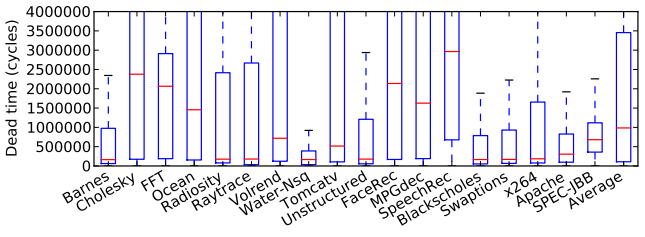


Fig. 4: Average time (cycles) from last access to a page in a core to its eviction in the TLB

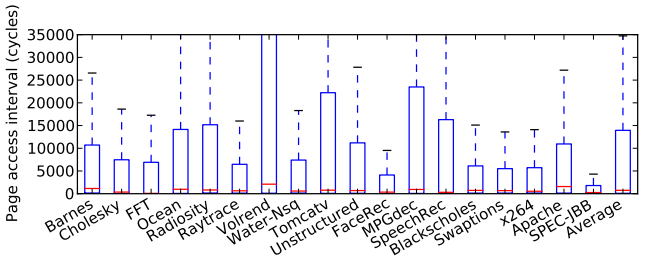


Fig. 5: Average time (cycles) between TLB accesses

of a page (percentage with respect to the total number of lives of the page) in which the page remains private. Figure 3 shows this percentage of private lives on shared pages (SP), and the percentage of shared lives on shared pages (SS), evidencing that one out of two shared page lives on average could be recognized as being private.

3.2 Approaching to the ideal scheme

Once analyzed the potential of temporal-aware mechanisms, we are interested in the design of feasible mechanisms based on the presence of the page in the TLB, trying to approach them to the *Ideal*. Using the TLBs instead of the page table to detect private pages allows to perform temporal-aware classification. Moreover, in order to approach the page generation time in a core to the ideal, rather than waiting for the TLB entry eviction, we use a mechanism to predict the last page access and invalidate it. This early invalidation mechanism needs to be small enough in time to tightly approximate the ideal, but sufficiently long as to avoid false evictions of pages that are going to be accessed again soon, which could cause severe damage to the system performance.

In this sense, Figure 4 represents the average page dead time (i.e. the time in cycles spent from the last access to the page TLB entry to its eviction). As this time metric is extremely variable through the samples of the different considered applications, the data is displayed as a box-and-whiskers plot. Here the focus is put on the median, which on average is located close to 1 million cycles. This means that an early eviction done 1 million cycles after the last access to a page will do successfully evict all the greater samples, which on average symbolizes half of the page lives. Yet this represents at the most an upper limit to the timer used to an early eviction. In some applications(e.g. Barnes, Swaptions, etc.) this value will act in barely a 25% of the page lives, evidencing that lower values could perform better.

On the other hand, the lower limit of a timer employed to early evict TLB entries should never be lower than the

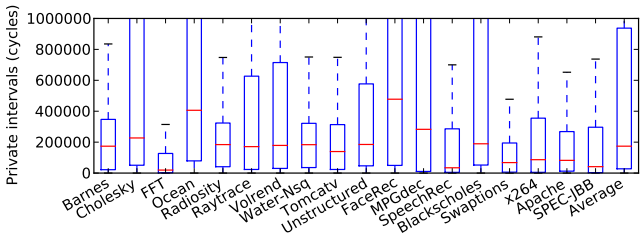


Fig. 6: Cycles spent as private on a global live

inter-access time of those entries to ensure no harming the system performance. Figure 5 shows the average number of cycles between accesses to the same page TLB entry in the same core. The number of cycles to evict the page should at least be approximate to the limit of the third quartile to avoid affecting negatively the system (with at least 75% of the samples below), meaning on average 10,000 cycles approximately. Although in some applications the limit is close to 25,000 cycles, turning out to be a safer lower value.

Lastly, as previously seen on Figure 1, there are intervals of a global page live where it becomes private, either to finally be evicted or to become shared again after some cycles. In these cases a notification mechanism able to track and update the sharing status of the page in the TLB may provide some benefit if those time periods are sufficiently long. Figure 6 shows how the samples representing these intervals are distributed. Note that a shared global page live will always start as private, but this period is negligible and is not represented on the graph. The longer the samples the more efficient and justifiable could be a notification mechanism to discover these private periods and update the page status accordingly. The median, on average, is placed around 200,000 cycles, although it evidences a high variability among the different applications.

So, we can conclude that a reasonable choice for a timeout value detecting the start of a dead time of a page should never be lower than 10,000 cycles and up to a threshold of 1 million cycles. This choice is endorsed by the exhibited results in Section 6. On the other hand, the high variability of the private intervals of page generation times discourage the use of a notification mechanism as it could be detrimental to the system performance.

4 TLB Miss Resolution through TLB-to-TLB Transfers

This section describes a simple mechanism that allows cores to retrieve information about the privacy of page and address translations from another core's TLB instead of from the page table. This mechanism, although simpler, has some similarities with the *Synergistic TLBs* mechanism proposed in [22]. Upon a TLB miss, getting the page information from a remote TLB is faster than from the page table, since "walking" the page table, often broken down into several levels, may imply several memory references (e.g., four memory references for the current 48-bit x86-64 virtual address space [23], or up to fifteen memory accesses for recent 32-bit ARMv7 virtual address space [24]).

Our mechanism works as follows. On a TLB miss, a page table walk process is started in order to get the address translation from the page table. However, simultaneously, the core snoops the other TLBs in the CMP by issuing a *page_info* request. When a core receives the page_info request, it checks its TLB and, in case of finding the page entry, the translation is sent to the requester by means of a short response message. When the first positive response is received, the page address translation is stored on the TLB and the memory request can proceed, therefore canceling the page table walk. Although upon every TLB miss the described mechanism snoops other TLBs, TLB misses are not frequent, thus keeping traffic overhead and energy consumption low and not jeopardizing its scalability, as shown in Section 7.3.

5 Temporal-Aware Private-Shared Classification

This paper deals with the fact that data can be requested by multiple cores and stored in their private caches during the application run time although being actually private (due to thread migration) or not shared at the same time (because of the different phases of applications). The detection mechanism that we propose (i) is aware of the temporality in memory references, (ii) can employ techniques to solve the TLB misses from the other CMP cores (as the one described in the previous section), and (iii) does not require extra hardware structures.

Although our detection mechanism can be applied to caches with any indexing and tagging technique, for the sake of clarity, in the next sections we assume the common virtually indexed, physically tagged (VIPT) L1 caches. A discussion about the application to other cache schemes is given in Section 8.4.

5.1 Basic idea

Our detection mechanism stores the sharing information along with the address translation in the TLB entries. Hence, each TLB entry has a Private (P) bit that indicates whether the page is private (bit set) or shared (bit clear). The P bit for a page is set when there is just one core TLB caching the translation. This indicates that it is the only core that can request such page blocks without causing a TLB miss. When several TLBs hold simultaneously an entry for that page, their P bits are clear. Additionally, a Process Identifier (PID) field is included to adequately distinguish them when sharing translation through the TLB-to-TLB translation resolution.

On memory references, before accessing the L1 cache, a TLB lookup is performed to get the physical address of the requested block. If a TLB miss takes place, a page_info request is sent to the other TLBs. The goal of this request is to get both the address translation and the information about the use of the page by the other cores. If any of the receiving cores is presumably going to access the page, then the TLB entry is marked as shared. Otherwise, the page will be classified as private.

TABLE 1: Response messages for TLB requests

State (in TLB or MSHR)	Address translation	Access prediction	Message type	Next state
Not Present	NO	Not used	Not_Present	Not Present
Requested (- or S)	NO	Used	Requested	Requested (S)
Decayed (P or S)	YES	Not used	Decayed	Not Present
Present (P or S)	YES	Used	Present	Present (S)

5.2 Page access prediction

As noted on Section 3.2, ideally the sharing condition of a page is settled by the simultaneity of the accesses to that page, and therefore each processor has to predict if it is going to access a page in a near future in order to accurately determine the use of data on pages within different phases of applications. To make predictions independent from the TLB size, we introduce a TLB *decay* technique, similar to the one proposed by Kaxiras *et al.* to save power in data caches [25]. The prediction works as follows. First, if the page entry is not present in the TLB, the core assumes that it will not access it in the near future. This situation can happen because the page has been never referenced by the core or because the entry has been evicted from the TLB since it has not been referenced for some time (TLBs employ a least recently used –LRU– policy). Second, if the page entry is present in the TLB, a 2-bit saturated counter is kept. This counter will be increased periodically according to a certain timeout and will be reset when any block within the page is accessed by the core (i.e., on a TLB hit). If the counter for a given entry saturates, the entry becomes *decayed*. Cores will consider decayed entries as not going to be accessed in a near future. Third, if the page entry is present and not decayed, the core predict that it will be accessed again. Table 1 shows the possible TLB entry states, the response messages provided by the core's TLBs to page_info requests, and the information included in the *responses*.

5.3 Coherent classification

The information about the private pages must be kept coherent in all the core TLBs. To keep this information coherent, we use the transition diagram shown in Figure 7 (excluding doted lines and *forced requests*, which will be explored later). Pages can be in three situations: (i) the page translation is not paged in any TLB; (ii) only one TLB holds the entry as private (either present or decayed); (iii) one or several TLBs hold the entry, all of them as shared.

Figure 7 shows the TLB state transitions depending on the incoming requests and responses to guarantee TLB coherence. When a TLB entry is *Not Present* in a given core and a local TLB request is issued by that core, the entry transitions to the *Requested* state. On the reception of remote page_info requests in this state, the TLB predicts to the others as accessing the page and the page transitions then to the *Requested S* state. This way, when two or more TLBs send page_info requests at the same time, they will answer to each other as accessing the page and TLB coherence is guaranteed since every TLB will see the page as shared. The *Requested S* transitions to *Present S* once all responses have been

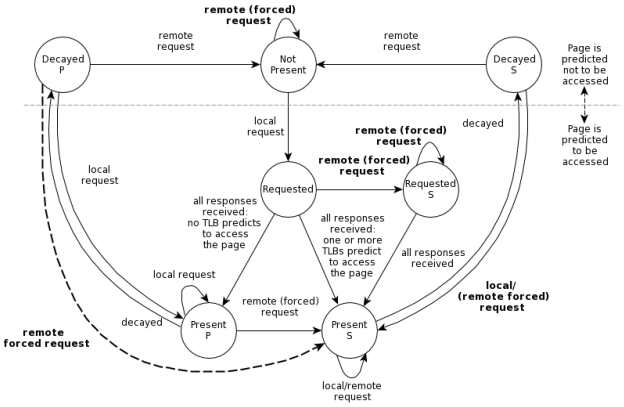


Fig. 7: TLB state transition diagram

received, regardless of their content. On the other hand, the *Requested* state transitions to *Present S* or *Present P* depending on the responses received from the other TLBs. It transitions to *Present S* if at least one TLB predicts to use the page, and to *Present P* otherwise.

When the entry becomes decayed, *Present P* and *Present S* states transition to *Decayed P* and *Decayed S*, respectively. On the other hand, from the *Decayed* states, if the core accesses the page, the decay counter is reset, and the TLB entry holding the page translation goes back to the *Present* state (P or S depending on the P bit). Note that these transitions only imply to saturate or reset the decay counter.

Finally, from the *Decayed* states, upon the reception of a remote TLB request, the TLB will answer with no intentions of accessing the page. At this point, this TLB loses the permission to access the page and the page transitions to the *Not Present* state. A subsequent access to that page will incur in a TLB miss. This can increase the number of TLB misses if the decay timeout is not chosen appropriately. However, this extra misses will probably find the entry in the other TLB and the miss will be resolved with short latency by means of a TLB-to-TLB transfer. TLB evictions (not shown for the sake of clarity) cause transitions to the *Not Present* state, but remaining TLBs are not notified about the evictions.

5.4 TLB-L1 cache inclusion policy

Since TLBs maintain per-page sharing information that finally will affect the coherence management of memory blocks, our proposal prevents data blocks from being stored in the core's L1 cache if the translation of their page is not stored by the TLB, as private pages are not aware of other cache copies. Hence, no copies of the blocks belonging to a private page can exist in other L1 cache. Specifically, a decayed TLB entry becomes *Not Present* (i.e. looses its permission to access the decayed page) on the reception of a TLB request. Furthermore, when a TLB entry transitions to *Not Present*, the cached blocks belonging to the corresponding page are evicted from L1 cache (and written back to the LLC when dirty). This will presumably hardly affect performance because of two reasons. First, a TLB entry transitions to *Not Present* when the page has not been accessed for some time, so most page blocks may have been already evicted

from the cache. Second, it is likely that no block within this page will be accessed in the near future. However with a low Decay value (more aggressive approach) it becomes more likely to evict an entry still alive, clearly hurting on execution time and network traffic.

When evicting the blocks from the cache, the access to the corresponding TLB entry is blocked. To do that, each TLB entry includes a *Lock* (L) bit. Note that when the state transitions from *Decayed* to *Not Present*, the corresponding response message is not sent until the page flushing of all the blocks belonging to the page is completed to ensure coherence.

5.5 Decay override on premature invalidations

As it will be exposed later, the *Decay* technique is quite effective to carry out a greedy classification of pages as private as a result of the capability to precisely predict the live time of a page. However, due to the high variability on the number of cycles of a page live time or the access interval itself, the *decay* could, in some cases, aggressively invalidate an entry that will presumably be accessed again soon, specially with low decay values. Furthermore, the cache inclusion policy between the TLB and the L1 cache will cause these premature invalidations to be specially harmful to the overall system performance, as it will evict L1 cache lines still being exploited, therefore dramatically increasing L1 cache miss rate.

To mitigate this drawback of *decay* employment it may be preferred in some cases to ignore it and allow the page to remain and become shared. To this end, it is required to have some kind of evidence of the possible occurrence of premature invalidations, allowing us to override the *decay* mechanism. In this context, when a TLB miss occurs (TLB entry is invalid or *Not Present*) but the requested TLB entry is still allocated in the local TLB, we might suspect that a premature invalidation occurred, and therefore, it may be advisable to cancel the decay invalidation on other core's TLBs for the current TLB miss resolution.

To carry out the *decay* override, we propose the use of a special request, which will be referred to as *forced* request, as it indeed forces the classification of a page as shared that otherwise would have been classified as private. A forced request is sent on a TLB miss whenever the TLB entry, despite being in invalid state (*Not Present*), continues to be allocated in the TLB. On the arrival of a *forced* request to a TLB, if the page is found *decayed*, it will be ignored, so the page will be automatically classified as shared and its invalidation will be canceled in anticipation of being accessed in the near future. Figure 7 shows how the TLB behaves when receiving a *remote forced* request applying the *Forced Sharing* approach. Basically, as soon as an entry is either on *Decayed P* or *Decayed S* states, when receiving a *remote forced* request it transitions to *Present S*, forcing it to become shared and preventing it from invalidate the TLB and L1 cache entries pertaining to the same page.

TABLE 2: Actions due to TLB-L1 inclusion and recovery

Application scenario	TLB-L1 inclusion	Recovery (P→S)
Coherence Deactivation	L1 flushing	L1 flushing
Reactive NUCA	L1 and LLC flushing	LLC flushing
Subspace Snooping	L1 flushing	No action
VIPS	L1 flushing	L1 flushing

5.6 Thread migration

Although the OS is not aware of phase changes of applications, it is aware of thread migrations. This section proposes a simple technique to help the page classification considering thread migrations. When a migration happens, all the TLB entries corresponding to pages accessed by the process of the migrated thread are marked as decayed. If the thread is then scheduled to run in a new core, all TLB pages will be found as decayed in the previous core's TLB, without going to main memory, and therefore, will be classified as private.

On the other hand, assuming different threads pertaining to the same process which are sharing virtual address space, if in the interim some pages are accessed by another thread in the previous core after migrating, the decay counter of the accessed pages will be reset and the pages will be considered as shared when requested by other core.

5.7 Actions required upon classification changes

Our temporal-aware page classification mechanism can be applied to perform different protocol optimizations. Depending on the optimization, when a page classified as private becomes shared, it may be needed to trigger a recovery procedure to restore the coherence state of the blocks belonging to the page as mentioned in Section 2. This recovery procedure depends on the optimization for which the private-shared classification is being applied to. Also, the TLB-L1 inclusion policy may vary depending on the purpose of the private-shared classification. Basically, it is necessary to perform the L1 flushing and the action required to restore the "normal" status of the lines, which usually matches with the recovery action. This actions are summarized in Table 2.

6 Simulation Environment

We evaluate our proposal with full-system simulation using Virtutech Simics [26] along with the Wisconsin GEMS toolset [27], which enables detailed simulation of multiprocessor systems. The interconnection network has been modeled using the GARNET simulator [28]. Interconnect power consumption estimations are done through ORION [29] power model. We simulate a 16-tile CMP architecture implementing directory-based cache coherence and with the parameters shown in Table 3. TLB miss latency considers four memory references to walk the page table, as in the 48-bit x86-64 virtual address space. Cache latencies have been calculated using the CACTI tool [30] assuming a 32nm process technology.

We use in our simulations a wide variety of parallel workloads from several benchmarks suites, covering different sharing patterns and sharing degrees. *Barnes*

TABLE 3: Base system parameters

Memory Parameters	
Processor frequency	2.8GHz
Cache hierarchy	Non-inclusive
Cache block size	64 bytes
Split instr & data L1 caches	64KB, 4-way (256 sets)
L1 cache hit time	1 (tag) and 2 (tag+data) cycles
Shared unified L2 cache	1MB/tile, 8-way (2048 sets)
L2 cache hit time	2 (tag) and 6 (tag+data) cycles
Directory cache	256 sets, 4 ways (same as L1)
Directory cache hit time	1 cycle
Memory access time	160 cycles
Split instr & data TLBs	128 sets, 4 ways
TLB hit time	1 cycle
Page size	4KB (64 blocks)
Network Parameters	
Topology	2-dimensional mesh (4x4)
Routing technique	Deterministic X-Y
Flit size	16 bytes
Data and control message size	5 flits and 1 flit
Routing, switch, and link time	2, 2, and 2 cycles

(8192 bodies, 4 time steps), *Cholesky* (tk15.O), *FFT* (64K complex doubles), *Ocean* (258 × 258 ocean), *Radiosity* (room, -ae 5000.0 -en 0.050 -bf 0.10), *Raytrace* (teapot), *Volrend* (head), and *Water-NSQ* (512 molecules, 4 time steps) are from the SPLASH-2 benchmark suite [31]. *Tomcatv* (256 points, 5 time steps) and *Unstructured* (Mesh.2K, 5 time steps) are two scientific benchmarks. *FaceRec* (script), *MPGdec* (525 tens 040.m2v), *MPGenc* (output of MPGdec), and *SpeechRec* (script) belong to the ALPBenchs suite [32]. *Blackscholes* (simmedium), *Swaptions* (simmedium), *Fluidanimate* (simsmall), and *x264* (simsmall) come from PARSEC [33]. Finally, *Apache* (1000 HTTP transactions), and *SPEC-JBB* (1600 transactions) are two commercial workloads [34]. Experimental results correspond to the parallel phase of the benchmarks.

7 Evaluation Results

We first analyze how the TLB-to-TLB transfers improve performance with the aim of knowing its contribution to the improvements obtained by our classification. Then, we also study the influence of the decay technique for TLBs, considering both the *Base Decay* and *Forced Sharing* approaches, on performance and classification accuracy. Finally, we study the benefits entailed by our proposals when applied to the *coherence deactivation* technique.

7.1 TLB-to-TLB Miss Resolution

This section analyzes the benefits and overheads of the TLB-to-TLB miss resolution mechanism by means of a sensitivity study of TLB sizes ranging from 256 sets to 64 sets (all of them are 4-way associative).[1]

The TLB-to-TLB miss resolution saves accesses to the page table, and therefore, reduces the TLB miss latency. This reduction translates into the improvements in execution time shown in Figure 8. Each bar shows the reduction in the number of cycles when compared to a configuration with the same TLB size but that does not implement TLB-to-TLB transfers. As we can observe, smaller TLBs implies more misses, and consequently, larger improvements in execution time when using the TLB-to-TLB transfer mechanism. On average, execution time is reduced by 6.5%, 11.9%, and 26.4% for 256-, 128-, and 64-set TLBs, respectively.

1. A deeper study can be found in [12].

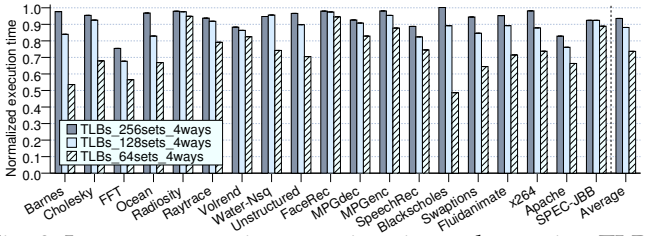


Fig. 8: Improvements in execution time when using TLB-to-TLB transfers

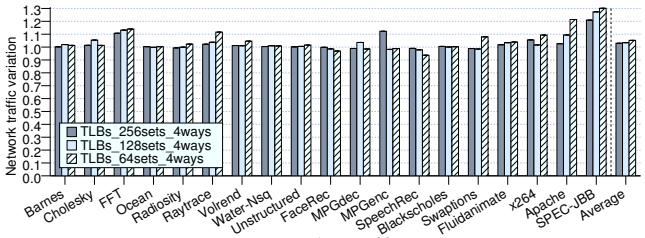


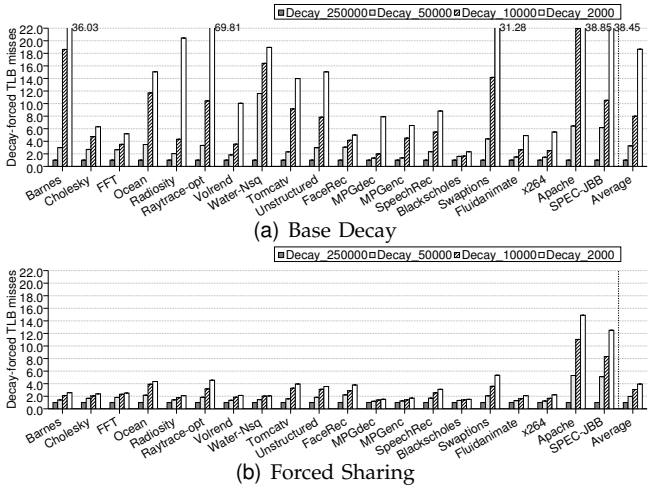
Fig. 9: Variations in network traffic when using TLB-to-TLB transfers

On the other hand, the network traffic can increase due to the extra page_info requests issued. Figure 9 shows the normalized network traffic. Since TLB misses are not frequent, the increase in traffic is low. Only *Spec-JBB* and *Apache* (and for small TLB sizes) reach an overhead of 20%. This is because commercial applications have a high TLB-versus-cache miss rate. Overall, the overhead in traffic of the TLB resolution mechanism is just 2.9%, 3.4%, and 5.2% for 256-, 128-, and 64-set TLBs, respectively. We believe that this low traffic overhead can be reasonably paid since the reduction in execution time is significant, as shown in Section 7.3.

7.2 Detection of private pages and decay value

This section shows the trade-off between the number of private pages detected by our mechanism and the number of TLB misses depending on the decay value and the approach chosen: *Base Decay* or *Forced Sharing*. A high value for the decay timeout results into few page entries decayed, and consequently few extra TLB misses. On the other hand, more private pages can be detected by our mechanism when a low timeout value is employed. This section analyses decay values of 250,000, 50,000, 10,000, and 2,000 cycles and considers 128-set TLBs.

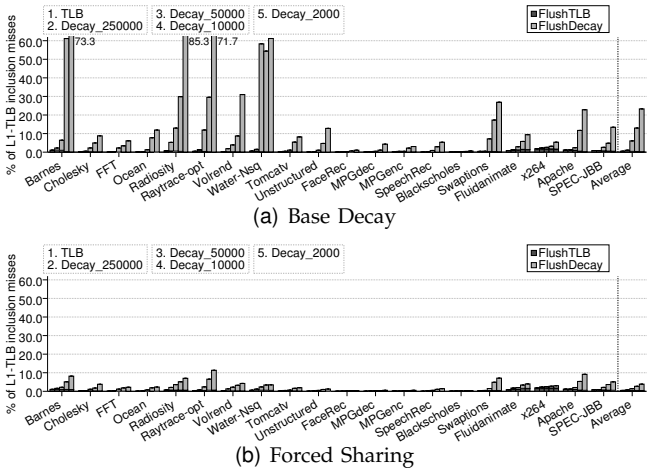
Figure 10 shows the number of TLB misses directly derived from the use of *Decay* technique (which come as a consequence of invalidating decayed TLB entries) for configurations with different decay timeouts, all of them normalized with respect to the maximum decay value considered. As can be seen, in the *Base Decay* approach, lower decay timeouts lead to increase the number of TLB misses, to a maximum of 69.81 times the total amount of misses obtained compared to the higher decay value considered, which will definitely impact negatively performance. Nonetheless, when using the



(a) Base Decay

(b) Forced Sharing

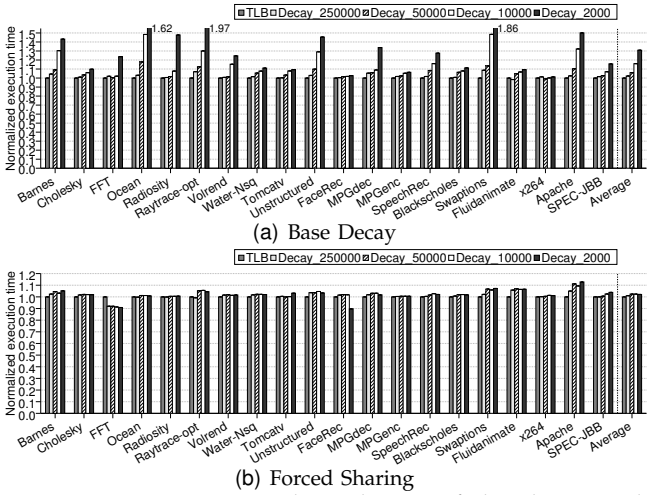
Fig. 10: Normalized TLB misses



(a) Base Decay

(b) Forced Sharing

Fig. 11: Percentage of L1 misses due to the TLB-L1 inclusion policy



(a) Base Decay

(b) Forced Sharing

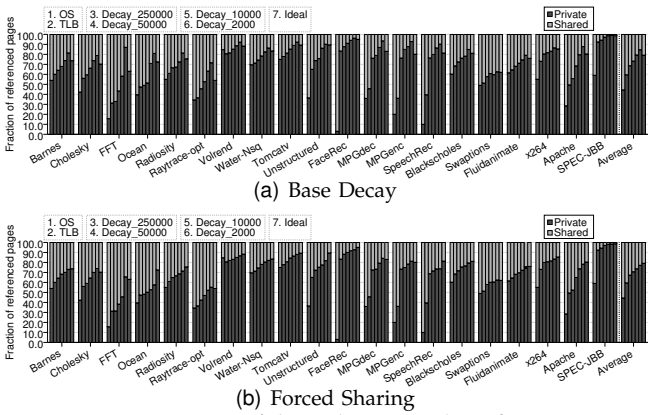
Fig. 12: Execution time degradation of the decay technique

Forced Sharing approach, this trend is much less harmful, with 4.73 times less TLB misses on average using the lower decay value. It is important to notice that the negative impact of the decay technique will be partly mitigated because those extra misses will probably find the address translation in the TLB of the core that caused the invalidation of the decayed entry.

Due to the TLB-L1 inclusion policy described in Section 5.4, when a TLB entry is evicted (*FlushTLB*) or invalidated (*FlushDecay*), all blocks pertaining to the corresponding page are evicted from the L1 cache. If either the decay technique or the LRU mechanism invalidates or evicts, respectively, a TLB entry that is on his dead time, there will not feasibly be many blocks present on the L1 cache, and they will not be likely revisited. But if the TLB entry is prematurely invalidated, the L1 cache will miss and thus, this will affect negatively to the system performance, both in terms of time and energy consumption. Figure 11 shows that for the *Base Decay* (Figure 11(a)), on average, nearly a 23.1% of the L1 misses are due to the inclusion policy when using the 2,000 decay, though in some benchmarks, as Barnes or Raytrace, it grows over 70%, and up to 85.3% in the case of Radiosity. On the contrary, using the *Forced Sharing* approach (Figure 11(b)), on average, a maximum of 3.87% of L1 misses are attributable to the inclusion policy.

Keeping an accurate classification entails some performance degradation due to the extra TLB and L1 misses, as shown in Figure 12, which is normalized to a system using TLB-to-TLB transfers, but without decay. The *Base Decay* approach degrades performance by 2.3% for 250,000 cycles decay, and up to a 30% for 2,000 cycles decay (Figure 12(a)). On the other hand, the *Forced Sharing* (Figure 12(b)) technique is able to nearly eliminate the negative impact of the low decay values. The degradation in execution time is less than 2.41% regardless of the employed decay value.

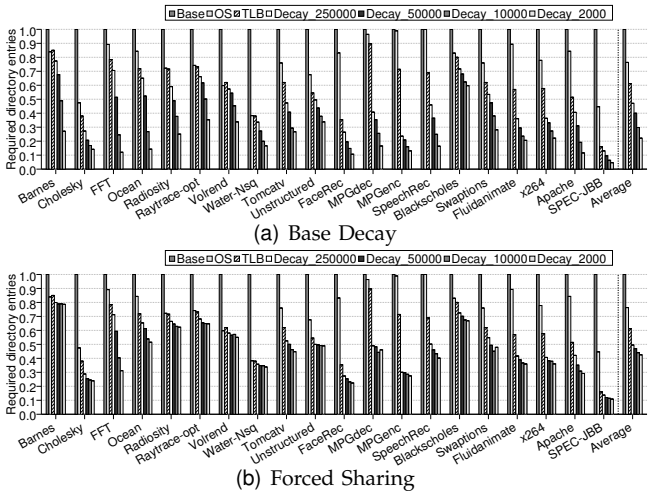
Although a low decay timeout leads to extra TLB and L1 misses, it also helps to detect more private pages. Figure 13 shows the pages considered as private and shared according to each mechanism. Particularly, we show numbers for the OS-based detection (*OS*), the idealized page live time introduced in Section 3 (*Ideal*), our base proposal without employing decay techniques (*TLB*), and a more elaborate detection using the decay technique. If a page has been considered as shared at least once during the execution of the application, it will be plotted as shared in the graph. Note that this is unfair for our approach since, unlike the OS-based proposal, we are able to re-classify pages from shared to private. We can see that the TLB mechanism classifies 15% more pages as private than OS. Additionally, employing the *Base Decay* approach we can improve this detection up to 40.2% (for a short decay timeout of 2,000 cycles), reaching a total of 84.69% (5% more private pages than the *Ideal*), but at the cost of dramatically increasing TLB misses, as previously shown. On the other hand, when using the *Forced Sharing* (Figure 13(b)) approach a 77.15% is obtained for the same decay timeout ($< 1\%$ deviation compared to the *Ideal*). It is important to notice that thread migration is rare in the evaluated applications due to its short execution time. As explained in Section 5.6, thread migration would have a dramatic impact for the OS classification, but not for our proposal.



(a) Base Decay

(b) Forced Sharing

Fig. 13: Private/shared pages classification



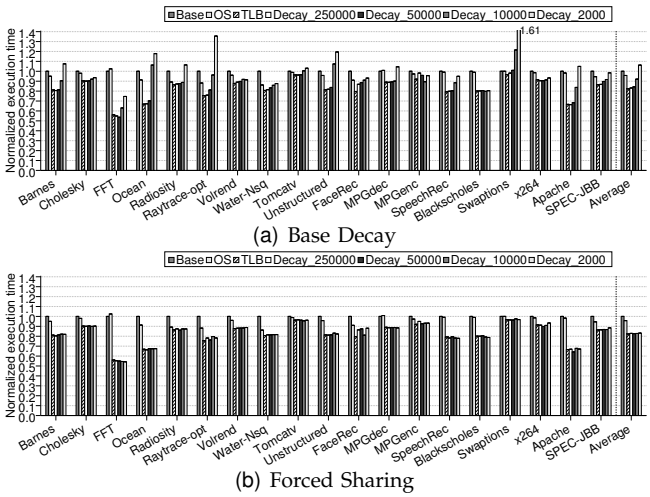
(a) Base Decay

(b) Forced Sharing

Fig. 14: Average directory entries required

7.3 A case of study: coherence deactivation

Previous sections analyze the impact on performance of the temporal-aware classification mechanism without taking advantage of possible optimizations. This section evaluates its impact when applied to the coherence deactivation scheme proposed for directory caches [7]. Although the percentage of detected private pages is a good general metric to show the goodness of our classification mechanism, each optimization has different requirements (see Section 5.7), and therefore, we need other metrics. For the coherence deactivation proposal we are interested in the number of required directory entries, which is shown in Figure 14. The OS-based classification can avoid the storage of 24.4% entries in the directory cache. However, when accounting for temporality, the number of entries required in the directory falls dramatically. Particularly, the *Base Decay* (Figure 14(a)) requires up to 78.3% less directory storage regarding the traditional TLB miss resolution that accesses the page table (*Base*). On the other hand, as the *Forced Sharing* (Figure 14(b)) detects less private pages, it requires more average directory entries. Specifically, it avoids the storage of up to 57.5% entries, i.e. 20.4% more storage is required when compared to the same timeout value for *Base Decay*. However, it must be taken into account that *Base Decay* virtually reduces the required



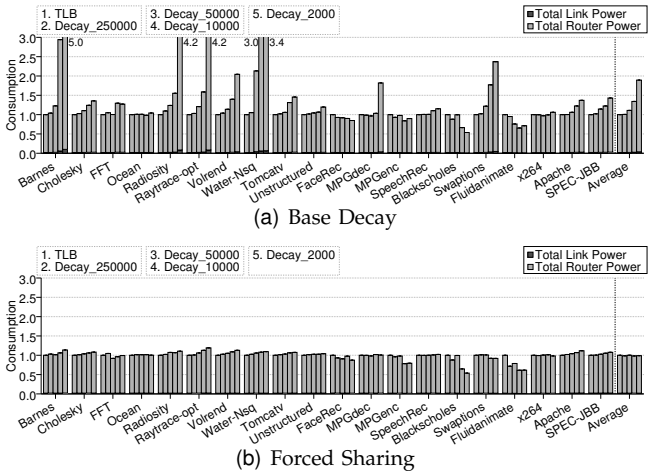
(a) Base Decay

(b) Forced Sharing

Fig. 15: Execution time improvements for coherence deactivation

number of directory entries at the expense of execution time degradation compared to *TLB*, in inverse proportion to the decay timeout value. In particular, the 2,000 cycles decay timeout for the *Base Decay* is prohibitively time consuming as previously noted (30% degradation as seen Figure 12(a)), while on the other hand, the same timeout using the *Forced Sharing* approach could turn to be a reasonable choice.

This large improvement of directory usage results in less directory evictions and consequently less invalidations in the private caches. Directory invalidations cause extra cache misses (named as *coverage* misses). Therefore, coherence deactivation reduces the number of coverage misses, and the execution time can be improved as shown in Figure 15. The figure is normalized with respect to the baseline configuration that does not detect private pages, plotting results for both *Base Decay* and *Forced Sharing* configurations. The OS-based classification can just reduce 4.3% the execution time. When compared to the base configuration, *TLB* reduces the execution time by 13.5% due to both the reduction in the required directory entries when applying temporal classification and the TLB miss resolution mechanism. For most applications the introduction of the decay technique either hurts performance (case of *Base Decay*) or does not provide additional improvements in execution time (case of *Forced Sharing*) with respect to *TLB*. *FFT* is the exception and only on higher decay timeout values using *Base Decay*. *Forced Sharing* approach (Figure 15(b)) is still capable to reduce the execution time a 7% using a 2,000 cycles decay. The benefits derived from the larger number of detected private blocks do not offset the overhead introduced. These results are mainly due to the prematurely invalidated TLB entries, but it is also altered by the "large" size of the directory (256 sets, 4-way). Since *TLB* removes 38.3% of entries, the current directory size is not a bottleneck any more. Nevertheless, in scenarios with smaller directories, or with frequent thread migration or with larger TLBs, the decay mechanism may introduce more improvements in performance, as shown below.



(a) Base Decay

(b) Forced Sharing

Fig. 16: Normalized Network consumption

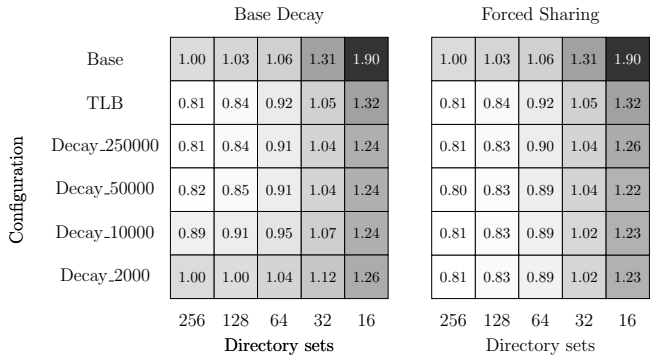


Fig. 17: Normalized execution time depending on the directory size and the decay timeout employed

As previously noted, coherence deactivation reduces coverage misses on L1 cache, while, on the other hand, decay technique application increases the overall number of misses. Cache misses come with a great impact on the network traffic, and therefore, on the network consumption. Figure 16 shows the energy consumption of the interconnect network for *TLB* and *Decay* techniques. As can be observed, the energy consumption is more relevant on lesser decay values. With *Base Decay* (Figure 16(a)) the network energy consumption increase is inversely proportional to the decay value, consuming up to 90% more energy in the interconnect than the *TLB* approach. However, *Forced Sharing* (Figure 16(b)) implementation prevents the network consumption growing over configurations, consuming up to 2.73% less power when compared to the *TLB*.

We may be interested in reducing the directory cache size to make it more scalable and fast. Reducing the directory size will increase execution time for all the configurations, but the decay technique allows to mitigate this increase. Figure 17 shows different configurations in the y axis and different cache sizes in the x axis. The value shown in each cell corresponds to the average execution time of all the applications normalized to the base configuration. When moving to smaller directory caches we can see that the use of the decay technique becomes necessary for a good performance. Particularly, using a 16-set directory, the *Forced Sharing* (right table) approach can outperform the *TLB* approach by up to 10% of its execution time (68% regarding *Base*). It also exceeds the *Base Decay* (left table) approach, mainly for lower decay values, by up to 19%. As we can see, the smaller the directory is, the smaller decay timeout is recommended, and hence the more suitable the *Forced Sharing* approach is when compared to *Base Decay*.

As a conclusion, we can say that the election of the decay value will depend on the benefits of performing a more accurate detection of private pages for the different protocol optimizations. When the improvements obtained thanks to the private-shared optimizations are higher, a smaller decay timeout can translate into more benefits. However, low decay values introduce overhead due to the induced early TLB invalidations. *Forced Sharing* approach, which slightly modifies the *Base Decay* state machine in order to enforce page sharing when an early TLB invalidation is detected, constrains these early TLB invalidations. By doing so, it mitigates the overhead introduced when using low decay values, avoiding many L1 cache misses produced by the inclusion policy, thus improving both traffic overhead (and energy consumption related to this traffic) and execution time. On the other hand, *Forced Sharing* deteriorates to some extent the amount of private pages detected when compared to the *Base Decay*, resulting on lower directory storage requirements reduction. Nevertheless, whereas the lower decay timeouts using *Base Decay* are reasonably impractical due to the performance loss, their usage is plausible under *Forced Sharing*. It has been evidenced that using a 16-set directory, *Forced Sharing* approach suffices to outperform *TLB* execution time to up to 10%.

8 Discussion

8.1 Scalability

The proposed TLB-based private-shared classification works fairly well for small- or medium-scale systems due to the reduced number of TLB misses (only up to 1% of the TLB accesses are misses). Large-scale systems may have excessive traffic when using this classification mechanism, even considering that TLB misses are much less frequent than cache misses. Although this paper focuses on small- or medium-scale systems, for larger systems our proposal could be adapted to reduce the traffic generated, for instance by having centralized sharing TLB information [35] (shared second level TLB structure, with or without a directory-like organization) to avoid broadcasts, or by employing a Token-like protocol [36] to avoid most TLB responses.

8.2 Large or multilevel private caches

Due to the TLB-L1 inclusion policy, after every TLB eviction, the blocks pertaining to the page are flushed from the private cache. The number of lines visited within the cache is not related to the cache size but to the page size (64 cache lookups for 4KB pages). However, the number of blocks effectively invalidated

upon a flush operation may increase with the cache size since more blocks belonging to an evicted page may be stored in a larger private cache, ultimately increasing cache misses. In this scenario, the TLB should scale accordingly to the cache size or use a multi-level TLB approach in order to dramatically reduce the total flush operations performed. Also, if applicable, decay timeout values would have to be modified accordingly to the live times of blocks in the private cache to avoid premature invalidations, which could increase inclusion-induced L1 cache misses.

Additionally, it is also applicable to configurations with two or more levels of private caches. In this case, performing the page flushing requires the invalidation of the corresponding page blocks at every private cache in the hierarchy. This action can be performed in parallel.

8.3 Large pages

Our proposal can also work in systems implementing large pages (with 2MB being the most common alternative on x86 architectures). However, the eviction of the entries for large pages from the TLB will require a more expensive cache flushing. In order to overcome the latency overhead, diverse approaches could be employed. For example, a simple counter can be added to the TLB entry indicating the number of *live* or cached blocks for the evicted TLB entry. Alternatively, a presence vector tracking large memory regions could be used, similar to the one used in [37], which would set the region limits to be flushed. Both approaches aim on reducing the amount of required lookups when flushing. They could even be combined, provided that the TLB entry size increase could be afforded.

Alternately, based on the observation that the majority of pages accessed per core on systems with superpage support are the smallest pages [38], we could avoid classifying superpages and consider them as coherent without significantly damaging system performance, while completely avoiding large pages overhead. Current systems supporting multiple page sizes implement multiple TLB structures to do so, thus the TLB actually storing large pages could classify them as shared without extra hardware support.

8.4 Virtual caches

Although this work assumes the common case of virtually indexed, physically tagged (VIPT) L1 caches. Our proposal is directly applicable to any other cache where the access to the TLB is required.

On the other hand, virtually indexed, virtually tagged (VIVT) caches, a.k.a. virtual caches, do not require TLB accesses for cache hits, which can result in faster lookups and less power consumption than the physical caches. Fortunately, the address translation is anyway performed on every cache miss since coherence is kept for physical addresses. Also, L1 hits do not issue coherence actions. These two characteristics allow our proposal to be completely applicable to virtual caches. We can still maintain TLB-L1 inclusion and classify every cache miss into private or shared.

9 Related Work

Some works rely on the compiler and/or memory allocator to classify memory pages in order to either remove coherence for private pages [39] or improve data placement [10], [11]. In [10], a data ownership analysis of memory regions is performed at compilation time. This information is transferred to the page table by modifying the behavior of the memory allocator by means of hooks. This proposal is further improved in [11] by considering a new class of data, named as practically-private, which is mapped to the NUCA cache according to a first-touch policy. In [39], private data is not stored at the LLC with the aim of avoiding cache thrashing for private blocks. Unlike our approach, these works statically mark data as private either by the memory allocator or at compile time, when privacy of some data cannot be guaranteed.

SWEL [13] is a novel hardware-based coherence protocol that uses a private-shared block classification at the directory to allocate shared read-write blocks only at the shared LLC, thus avoiding coherence maintenance for them. The main drawback of that proposal is the latency penalization of accessing shared read-write blocks, which must be served by the LLC cache. POPS [14] decouples data and coherence information in the shared LLC to reduce access latency to this information and to improve the aggregate NUCA capacity. It also employs a directory private-shared classification (this time with the help of a predictor table). Spatiotemporal Coherence Tracking [15] also classifies private and shared data at the directory, accounting for temporal private data. It tries to find large private regions to merge them in the directory to save directory space. Differently, our approach is not only aiming directory size reduction, but it has a larger scope, as described in Section 2. In general, private-shared classification at the directory has the drawback of adding extra area requirements. Additionally, it is not suitable for simple request-response protocols (such as VIPS [16], [40]), because of the requests sent by the directory upon private-to-shared changes.

Our proposal discovers the private-shared page status by benefiting from the resolution of TLB misses through TLB-to-TLB transfers. Concerning fast resolution of TLB misses, Synergistic TLBs [22] employs the snooping of other TLBs to propose a distributed-shared TLB organization. Also, UNITD Coherence [41] employs the TLB snooping to integrate the existing cache coherence protocol with the TLB coherence maintenance. Differently, we employ the TLB snooping to account for the temporality of private accesses and, so, improve the private-shared classification. In [42], neighbor TLBs are snooped with the aim of detecting shared pages. However, that proposal requires important modifications in the TLB, such as making it both physically and virtually indexed or the addition of a non-scalable full-sharing vector. Differently, we just add 4 bits to the TLB regardless of system size. Also we employ a decay technique for TLBs that allows more accurate access predictions.

Bhattacharjee *et al.* use inter-core cooperative TLB prefetchers to reduce the number of TLB misses by exploiting commonality and predictability in TLB miss

patterns across cores in CMPs [43]. After a core resolves a TLB miss, it can either push the address translation into the TLBs of the potential predicted sharers or search itself in advanced predictable future translations. In both cases, predictable translations are placed into a prefetch buffer. Alternatively, the authors propose to use a last-level TLB shared by all cores to achieve the same goal [35]. Both can be use in conjunction with our proposal.

A similar approach is made by Y. Li *et al.*, based on inter-core translation sharing for shared translations in order to reduce TLB misses through the use of a partial sharing buffer (PSB) [17]. It relies on the page table to capture the sharing state of the page and only when it becomes shared, the translation is stored on the PSB thus reducing TLB miss penalty with minimal additional hardware resources. Though, it does not count for temporality as TLB-to-TLB transfers do.

10 Conclusions

This paper studies the potential benefits of accurately predicting the access patterns of a page at the TLB in order to fully exploit a private/shared classification mechanism. To do so, we propose a new temporal-aware mechanism that improves the previously proposed by Ros *et al.* [12] for private pages detection, so we can limit the damage caused by miss predictions. Our mechanism classifies pages accessed by several cores at different time instants (e.g., thread migration or program phase changes) as private. This leads to a bidirectional page re-classification, from private to shared, and vice versa, resulting in a significant increase in the number of pages considered as private compared to an OS-based classification (from 43% to 79%). Furthermore, the new classification mechanism makes more appealing the use of a decay technique, reducing the TLB misses forced by the mechanism itself up to 4.73 times on average, and dramatically reducing power consumption and execution time while still providing directory storage requirements reduction (up to 50% less directory entries, with 7% execution time cutback compared to OS-classification).

Acknowledgments

This work has been jointly supported by the MINECO and European Commission (FEDER funds) under the project TIN2012-38341-C04-01/03 and the *Fundación Seneca-Agencia de Ciencia y Tecnología de la Región de Murcia* under the project *Jóvenes Líderes en Investigación* 18956/JLI/13.

References

[1] J. F. Cantin, J. E. Smith, M. H. Lipasti, A. Moshovos, and B. Falsafi, "Coarse-grain coherence tracking: RegionScout and region coherence arrays," *IEEE Micro*, vol. 26, no. 1, pp. 70–79, Jan. 2006.

[2] N. Agarwal, L.-S. Peh, and N. K. Jha, "In-Network Snoop Ordering (INSO): Snoopy coherence on unordered interconnects," in *15th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2009, pp. 67–78.

[3] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *42nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2009, pp. 423–434.

[4] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *17th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2011, pp. 169–180.

[5] D. Sanchez and C. Kozyrakis, "SCD: A scalable coherence directory with flexible sharer set encoding," in *18th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2012.

[6] D. Kim, J. A. J. Kim, and J. Huh, "Subspace snooping: Filtering snoops with operating system support," in *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2010, pp. 111–122.

[7] 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," in *38th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2011, pp. 93–103.

[8] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-level page allocation," in *39th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2006, pp. 455–465.

[9] 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)*, Jun. 2009, pp. 184–195.

[10] Y. Li, A. Abousamra, R. Melhem, and A. K. Jones, "Compiler-assisted data distribution for chip multiprocessors," in *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2010, pp. 501–512.

[11] Y. Li, R. G. Melhem, and A. K. Jones, "Practically private: Enabling high performance cmps through compiler-assisted data classification," in *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 231–240.

[12] A. Ros, B. Cuesta, M. E. Gómez, A. Robles, and J. Duato, "Temporal-aware mechanism to detect private data in chip multiprocessor," in *42th Int'l Conf. on Parallel Processing (ICPP)*, Oct. 2013.

[13] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian, "SWEL: Hardware cache coherence protocols to map shared data onto shared caches," in *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2010, pp. 465–476.

[14] H. Hossain, S. Dwarkadas, and M. C. Huang, "POPS: Coherence protocol optimization for both private and shared data," in *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2011, pp. 45–55.

[15] M. Alisafaee, "Spatiotemporal coherence tracking," in *45th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2012, pp. 341–350.

[16] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 241–252.

[17] Y. Li, R. G. Melhem, and A. K. Jones, "PS-TLB: Leveraging page classification information for fast, scalable and efficient translation for future cmps," in *TACO*, 2013, pp. 28–28.

[18] M. Kambadur, K. Tang, and M. A. Kim, "Harmony: Collection and analysis of parallel block vectors," in *39th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2012, pp. 452–463.

[19] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *10th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2001, pp. 3–14.

[20] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Blade computing with the AMD Opteron™ processor ("Magny Cours")," in *21st HotChips Symp.*, Aug. 2009.

[21] D. A. Wood, M. D. Hill, and R. Kessler, "A model for estimating trace-sample miss ratios," in *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1991, pp. 79–89.

[22] S. Srikantaiah and M. Kandemir, "Synergistic TLBs for high performance address translation in chip multiprocessors," in *43rd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2010, pp. 313–324.

[23] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: Skip, don't walk (the page table)," in *37th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2010, pp. 48–59.

[24] *ARM Architecture Reference Manual ARMv7-A and ARMv7-R*, 2012.

[25] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *28th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2001, pp. 240–251.

[26] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner,

"Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.
[27] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.
[28] 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)*, Apr. 2009, pp. 33–42.
[29] A. Kahng, B. Li, L.-S. Peh, and K. Samadi, "ORION 2.0: A Power-Area simulator for interconnection networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 1, pp. 191–196, 2012.
[30] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," HP Labs, Tech. Rep. HPL-2008-20, Apr. 2008.
[31] 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)*, Jun. 1995, pp. 24–36.
[32] 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*, Oct. 2005, pp. 34–45.
[33] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 72–81.
[34] 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)*, Feb. 2002, pp. 30–38.
[35] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level TLBs for chip multiprocessors," in *17th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2011, pp. 62–73.
[36] M. M. Martin, M. D. Hill, and D. A. Wood, "Token coherence: Decoupling performance and correctness," in *30th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2003, pp. 182–193.
[37] J. Zebchuk, B. Falsafi, and A. Moshovos, "Multi-graincoherence directory," in *46th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, 2013.
[38] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly tlb designs," in *21th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2015.
[39] J. Meng and K. Skadron, "Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling," in *Int'l Conf. on Computer Design (ICCD)*, Oct. 2009, pp. 282–288.
[40] S. Kaxiras and A. Ros, "A new perspective for efficient virtual-cache coherence," in *40th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2013, pp. 535–547.
[41] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy, "UNified instruction/translation/data (UNITD) coherence: One protocol to rule them all," in *16th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2010, pp. 1–12.
[42] 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)*, Aug. 2002, pp. 243–246.
[43] A. Bhattacharjee and M. Martonosi, "Inter-core cooperative TLB for chip multiprocessors," in *15th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2010, pp. 359–370.

Albert Esteve received the MS degree in computer science from the Universitat Politècnica de València, Spain, in 2012. He is currently a PhD student at the Parallel Architecture Group (GAP) of the Universitat Politècnica de València with a fellowship from the Spanish Government. His research interests include cache coherence protocols, and chip multiprocessor architectures.

Alberto Ros received the MS and PhD degree in computer science from the University of 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. He has been working as a postdoctoral researcher at the Universitat Politècnica de València and at Uppsala University. Currently, he is Associate Professor at the University of Murcia. His research interests include cache coherence protocols and memory hierarchy designs for manycore architectures.

María E. Gómez obtained her 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, network-on-chips and cache coherence protocols.

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 referred 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 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*.