TLB-Based Temporality-Aware Classification in CMPs with Multilevel TLBs

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

Abstract—Recent proposals are based on classifying memory accesses into private or shared in order to process private accesses more efficiently and reduce coherence overhead. The classification mechanisms previously proposed are either not able to adapt to the dynamic sharing behavior of the applications or require frequent broadcast messages. Additionally, most of these classification approaches assume single-level translation lookaside buffers (TLBs). However, deeper and more efficient TLB hierarchies, such as the ones implemented in current commodity processors, have not been appropriately explored.
This paper analyzes accurate classification mechanisms in multilevel TLB hierarchies. In particular, we propose an efficient data classification strategy for systems with distributed shared last-level TLBs. Our approach classifies data accounting for temporal private accesses and constrains TLB-related traffic by issuing unicast messages on first-level TLB misses. When our classification is employed to deactivate coherence for private data in directory-based protocols, it improves the directory efficiency and, consequently, reduces coherence traffic to merely 53.0%, on average. Additionally, it avoids some of the overheads of previous classification approaches for purely private TLBs, improving average execution time by nearly 9% for large-scale systems.

Index Terms—Distributed Shared TLB, Data Classification, TLB Usage Predictor, Coherence Deactivation

1 INTRODUCTION

OVER the last years high performance processors have evolved by doubling the core count every 18 months. This, coupled with the fact that most chip multiprocessors (CMPs) provide programmers with a shared-memory model, turns coherence maintenance in multilevel cache hierarchies into an increasingly critical issue. To meet these new scalability challenges, many proposals focus on directory-based coherence solutions as they tend to use less network bandwidth than their snooping-based counterparts, thus representing the most scalable alternative. However, as core count grows, directory-based protocols demand larger amounts of directory storage and energy. Due to physical and technological constraints, the storage area dedicated to the directory in the die must be limited. This fact definitively causes a larger amount of directory-induced invalidations as a result of replacements in the directory, which may severely degrade performance [1].

An effective way to improve directory efficiency and reduce replacements is based on classifying data as private or shared in order to handle blocks more efficiently according to their sharing status. Considering such a classification, Cuesta *et al.* propose deactivating coherence maintenance for private [1], or more generally non-coherent [2], blocks. The deactivation reduces the coherence overhead, ultimately enabling smaller or more efficient directory designs as it eludes the tracking of these blocks. Alternatively, Demetriades and Cho [3] avoid the invalidation of private blocks stored in the cache when evicting directory entries by delegating block-discovering responsibilities to the last-level cache (LLC) organization, and therefore reducing the number of required directory entries while barely affecting performance.

The aim of a classification mechanism is to detect as much private data as possible without degrading the overall system performance, minimizing its overheads. However, data classification usually requires a large amount of extra storage in order to track the data sharing status. Some mechanisms cannot dynamically reclassify data from shared to private, thus missing the detection of temporarily-private data and limiting the potential performance benefits of data classification. Recently, Ros *et al.* [4] propose a mechanism that deals with these drawbacks by (i) storing the sharing information in the system translation lookaside buffers (TLBs) and (ii) discovering the current sharers when a TLB miss takes place by broadcasting requests to other cores' TLBs and reclassifying the page if necessary. The evaluation of this proposal only assumes *private*, *single-level* TLBs.

Nevertheless, current multicore systems use multilevel TLB structures which help to address the increasing application memory footprints and constrain the performance loss that comes with them. Furthermore, shared last-level TLBs have been already explored by Bhattacharjee *et al.* [5], and Lustig *et al.* [6], exhibiting a reduction in TLB misses for parallel workloads when compared to private last-level TLBs.

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

Our proposal. In this paper we propose and evaluate techniques to perform a TLB-based data classification for multilevel TLBs that employ either a private or a shared last-level TLB, and compare them with previous single-level TLB classification approaches. The main contributions of this work are:

- We propose the first adaptive TLB-based classification mechanism for shared TLB structures.
- We show the importance of usage prediction techniques for multilevel TLBs, which make the classification insensitive to TLB size.
- We propose a usage predictor for shared TLB structures (SUP), which naturally avoids overheads over previous predictors for private TLB structures.
- We perform an extensive evaluation considering single and multilevel private TLBs, and multilevel TLBs with a shared last-level TLB.
- We show how classification schemes improve system performance in contemporary architectures when coherence maintenance is deactivated for private data.

Results. Through full-system simulations, we show how execution time improves by 6.8% over a baseline configuration with private L2 TLBs classification scheme, and up to 8.2% with a distributed shared L2 TLB classification scheme. Furthermore, through precise traffic analysis, we reveal how TLB-based classification mechanisms benefit from shared TLB structures, reducing TLB and cache traffic to only 53.0%. Finally, *SUP* avoids traffic overhead for a 32-core system, ultimately reducing the traffic issued by 30.9%, compared to the baseline.

The rest of the paper is structured as follows. Section 2 reviews current classification mechanisms and TLB organizations in the literature. Section 3 describes the temporal-aware classification mechanism for multilevel TLBs with private last-level TLBs. Section 4 describes the proposed classification mechanism for multilevel TLBs with shared last-level TLBs, and discusses some key aspects for its design. Section 5 introduces the simulation environment and methodology used to obtain the results presented in Section 6. Finally, Section 7 draws some conclusions.

2 Background and Motivation

Data classification mechanisms are gaining importance as they allow treating blocks or accesses more efficiently depending on their sharing status. Specifically, Kim *et al.* [7] avoid broadcast messages in *snoopy* protocols when accessing private blocks, thus leading to a reduction of network traffic. Li *et al.* [8] introduce a small buffer structure close to the TLB, namely partial sharing buffer (PSB). When a page becomes shared it will feasibly be found on the PSB upon a TLB miss, obtaining the page translation with both lower latency and fewer storage resources. Hardavellas *et al.* [9] and Kim *et al.* [10], [11] keep private blocks on the local bank in distributed shared caches in order to reduce access latency. Ros and Kaxiras [12] propose an efficient and simple cache-coherence protocol by implementing a write-back policy for private blocks and a write-through policy for shared blocks. End-to-End SC [13] allows instruction reordering and out-of-order commits of private accesses from the write-buffers, since they do not affect the consistency model enforced by the system. Finally, Cuesta *et al.* [1] propose avoiding directory storage of private blocks, therefore deactivating coherence maintenance for those blocks and leading to smaller and faster directories. The evaluation of this work is built on top of this optimization, which is explained in section 2.3.

2.1 Classification techniques

All proposals described above use classification approaches that take advantage of existing OS structures (i.e. TLBs and page table) in order to perform the page classification scheme. An OS-based classification mechanism annotates the first requesting core in the corresponding page table entry (keeper [1] or FAC —first accessing core— [8] field) after the first TLB miss and classifies it as private. On subsequent page table accesses to the same page, the keeper field is compared to the current requester. If they do not match, then the entry is reclassified as shared. To this end, an extra-bit field is added to the page table and replicated in the TLB in order to fast access the sharing status. When a reclassification occurs, the sharing information in the keeper's TLB must be updated accordingly to avoid inconsistencies.

Other classification approaches have also been proposed. Directory-based mechanisms [14], [15], [16], [17], [18] suffer the important drawback that most of the data-optimization techniques for classification schemes are not applicable due to a late (post cache miss) discovery of the classification. Compiler-assisted approaches [10], [11] deal with the difficulty of knowing at compile time (i) whether a variable is going to be accessed or not, and (ii) in which cores the data will be scheduled and rescheduled. Finally, approaches based on the properties of programming languages [19], [20], despite being very accurate, are not applicable to most existing codes. Conversely, run-time approaches perform accurate classification for any code, thus avoiding these difficulties.

2.2 TLB-based classification

The sharing status of a page may evolve through different application phases from private to shared and back to private. An OS-based mechanism performs a non-adaptive classification, i.e. when a page transitions from private to shared it remains in that state for the rest of the execution time (or until it is evicted from the main memory). In applications that run for a long time, most pages may be considered shared at some point, thus neglecting the advantages of the classification. Allowing reclassification to private may significantly increase the number of private data detected when compared to an OS-based mechanism, and also alleviate miss-classification due to thread migration.

Temporality-aware classification. The goal of a *TLB-based classification mechanism* [4], [21], [22] is to achieve an adaptive classification that accounts for temporarily-private pages and tolerates thread migration. The TLB-based classification mechanism is based on inquiring the other cores' TLBs in the system (through TLB-to-TLB requests) on every TLB miss. The other TLBs reply to the requester indicating whether or not they are caching the page translation. This way, the TLB suffering the cache miss knows whether the page is shared or not.

Fast TLB-miss resolution. In TLB-based classification mechanisms, when a TLB miss takes place, a TLB-to-TLB request is broadcasted, in parallel with a page table walk. The TLBs that are caching the page translation include that information into the responses to that request as well. When the first page translation is received, it is stored in the requester TLB and the page walk is canceled. This way, the TLB miss latency can be considerably reduced [23], [24]. If no page translation is received from remote TLBs, the miss is completed when the page table walk process ends. A token-counting approach may be employed in order to avoid the requirement for all TLBs to reply upon every TLB-to-TLB request, which increases traffic requirements [22].

TLB-cache inclusion policy. TLB-based classification mechanisms rely on a strict inclusion policy between the TLB and the local caches to the processor (e.g. the L1 cache in this work). Hence, the absence of a valid page translation in the TLB ensures the absence of any valid block belonging to that page in the L1 cache. This is the reason why probing TLBs is a sufficient condition to guarantee that a page is private (i.e. avoids *false privates*). The TLB-cache inclusion policy does not necessarily hurt system performance, since evicted TLB entries have probably not been recently accessed (LRU-like policy is applied), and thus it is not common to find blocks for the evicted page in cache.

Usage predictor. TLB-based classification is determined by the presence of page entries in TLBs, despite the fact that they may have ceased to be accessed, thus making classification accuracy sensitive to the TLB size. A usage prediction mechanism for TLBs is essential to decouple classification from TLB size [4]. This predictor resembles the one employed in the Cache Decay approach [25] and determines whether or not a page is going to be accessed in the near future. In particular, the predictor uses one saturated counter per TLB entry that it is periodically increased according to an internal timeout and reset on every memory access to the page. When a TLB entry is predicted not to be used (its counter is saturated) and it is probed with a TLB-to-TLB request, the entry is invalidated. When a TLB entry is invalidated, all the blocks pertaining to the related page in the local cache hierarchy are invalidated, and the core is disqualified as a potential sharer of the page. This way, the usage predictor effectively increases the amount of private data detected for large or multilevel TLBs.

Forced sharing. When the usage predictor is very aggressive, shared pages can be eagerly invalidated from the TLBs. Premature invalidations induce more TLB misses, which in turn induce further invalidations. It can be seen as a positive feedback process. To prevent this scenario, a *forced-sharing* request can be issued when there is a TLB miss for an entry that is still present in the TLB but has been eagerly invalidated [21]. The forced-sharing request bypasses the usage predictor and avoids invalidating disused entries in remote TLBs.

2.3 Coherence deactivation

On current CMPs, the directory cache suffers from scalability issues. The directory area grows quadratically with the number of cores. Additionally, due to its limited associativity, it cannot simultaneously track all blocks stored in the processors' local caches. This causes directory entry evictions, which usually entail the invalidation of cached blocks. When a core accesses a block which has been invalidated due to directory coverage constraints, it causes a type of miss known as *Coverage* miss [26]. These misses may cause severe performance degradation.

Coherence Deactivation [1] is a technique that avoids the tracking of private (or non-coherent [2]) blocks by the directory. Consequently, directories exploit their limited storage capacity more efficiently, as long as the classification mechanism that detects private or non-coherent data is accurate.

Coherence recovery. Since coherence deactivation bypasses the coherence protocol for non-coherent accesses, a *recovery* operation is required when a page becomes coherent again, in order to avoid inconsistencies. To this end, the blocks of a non-coherent page that becomes coherent must be either evicted from the cache (flushing-based recovery) or updated in the directory cache (update-based recovery). Once the recovery mechanism has finished, the directory cache is in a coherent state according to the new page classification. Both recovery strategies show negligible performance impact, given that the recovery is triggered less than 5 times per 1000 cache misses, on average [2].

2.4 Multilevel TLBs

The organization of the TLB is becoming more and more crucial for performance in current CMPs. TLB misses are in the critical path of memory operations, and they may result in a long latency penalty. On a TLB miss, the page table is accessed in order to obtain the page address translation. "Walking" the page table often requires several memory accesses (e.g. up to twenty-four memory accesses on x86-64 virtual address space [27], or fifteen memory accesses for the recent 32-bit ARMv7 virtual address space [28], both supporting virtualized environments).

Since TLBs are usually accessed in parallel with L1 caches, the L1 TLB is commonly split into data and instruction TLBs. Moreover, the concept of multilevel cache hierarchies has been extended to TLBs, allowing private unified L2 TLBs to be present on many current architectures (e.g. Intel Xeon [29] or ARMv7 [28]). However, purely private TLB organizations exhibit some deficits, as they do not completely fulfill inter-core TLB sharing opportunities [30], leading to some avoidable and predictable TLB misses. Particularly on parallel applications, the same page translation is likely replicated on multiple TLBs when implementing private TLB structures, which wastes both storage resources and energy. Also, even on sequential applications, the system may experience TLB thrashing on some cores while other cores' TLBs exhibit small page footprints, provided that each individual TLB allocates a fixed set of resources.

To overcome this, Bhattacharjee *et al.* [31] use inter-core cooperative prefetchers to improve the TLB hit ratio, emulating a distributed shared TLB. They also propose a centralized shared L2 TLB [5], improving translation latency in a 4-core CMP. Both approaches have recently been explored further [6]. Nevertheless, centralized shared L2 TLBs are not relied upon when scaling to a large number of cores, since centralized organizations increase end-to-end latencies as the core count grows, which, as previously noted, will be added to the critical path. Furthermore, centralized shared TLBs may need a high-bandwidth interconnect to be able to connect to all the cores of a CMP.

TLB-based classification techniques have never been thoroughly evaluated for multilevel TLB hierarchies. Nonetheless, as previously noted, current architectures demand deeper and more efficient TLB structures. This work explores how TLB-based classification schemes perform comparatively under single and multilevel TLB hierarchies, including a comprehensive trade-off analysis. While assuming private L2 TLBs only requires straightforward extensions compared to single level TLBs, the use of a shared last-level TLB leads to a number of optimizations that decrease the overheads over previous classification mechanisms (see section 5).

3 TLB-based classification in systems with private L2 TLBs

The TLB hierarchy of contemporary architectures include split data and instruction private L1 TLBs and unified private L2 TLBs. Among the most common architectures adopting this TLB hierarchy we find AMD's K7, K8, and K10, Intel's i7 and Xeon, ARMv7 and ARMv8, or the HAL SPARC64-III [28], [32], [33], [34], [35]. TLB-based classification schemes, however, have been mostly analyzed for single-level TLBs. This section explains the implications of implementing a TLB-based classification mechanism [21] for a private, two-level TLB organization.

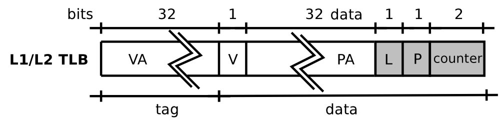


Fig. 1: L1/L2 TLB entry with the extra fields in gray.

3.1 Main considerations

Differently from a single-level TLB classification scheme, in a multilevel TLB hierarchy, a L1 TLB miss does not trigger a TLB-to-TLB request. Instead, the private L2 TLB is consulted. On an L2 TLB hit, the page translation is obtained and cached in the L1 TLB, thus resolving the L1 miss. On a L2 TLB miss, the TLB-to-TLB request is initiated along with the access to the page table.

TLB-to-TLB requests look up both the L1 and the L2 TLB looking for in-use TLB entries. If there is a hit in any of the TLBs, the response is positive, and the page translation is sent to the requester. This lookup can be performed in parallel. The usage predictor is implemented in both TLB levels. The 2-bit saturated counter is reset on a hit to its corresponding entry.

The forced-sharing optimization (see Section 2.2) also needs to be adapted for a multilevel TLB environment. In this case, the forced-sharing request takes place when accessing a present but invalid entry either in the L1 or the L2 TLB. In this case we consider that the TLB entry has been prematurely invalidated, as it has not been evicted yet from the TLB structure.

Figure 1 shows how TLB entries are extended for both TLB levels. The *Lock* (L) bit allows blocking memory accesses to the corresponding page while the sharing status is uncertain (e.g. while a TLB miss request is ongoing). The *Private* (P) bit tracks the page sharing status. The 2-bit saturated *counter* is used for the usage prediction mechanism, as explained in Section 2.2.

3.2 Implementation details

This section discusses the key implementation choices made for the proposed TLB-based classification scheme considering private two-level TLB hierarchies.

TLB inclusion policy. We employ an exclusive policy between the L1 and the L2 TLBs, since it maximizes the TLB capacity. Note that an increase in the number of TLB misses can dramatically degrade system performance. The downside of keeping exclusive TLBs is that, upon the reception of a TLB-to-TLB request, both TLB levels have to be accessed. We opt for performing this operation in parallel, thus incurring only the access latency of the L2 TLB, but at the cost of increasing the energy consumption in the case of a TLB hit. Since TLB-to-TLB requests are not frequent, we believe that this is the most appropriate design choice.

TLB consistency. The TLB hierarchy is shootdown-aware. As both TLB levels implement an exclusive policy, they can be checked in parallel. Furthermore, this property avoids incurring wrong (outdated) page classification, by flushing both the TLB and the cache.

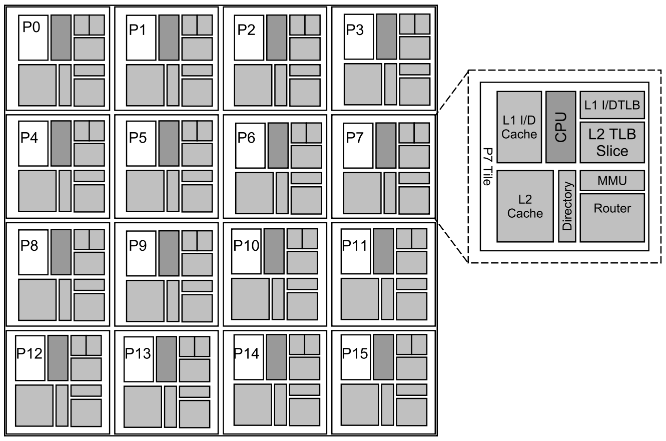
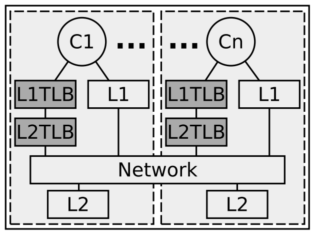


Fig. 2: Baseline architecture with a distributed shared L2 TLB organization.

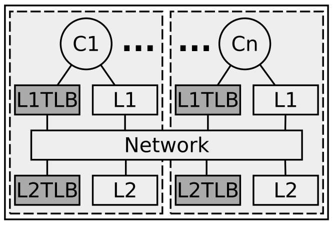
4 TLB-BASED CLASSIFICATION IN SYSTEMS WITH DISTRIBUTED SHARED L2 TLBS

Current application memory footprints demand deeper TLB hierarchies. Using a centralized shared last-level TLB, as an alternative to purely private TLB organization, provides high performance improvement mainly with parallel applications [5]. However, a centralized structure is not scalable.

In this work, we exploit the use of a distributed shared L2 TLB similar to the NUCA cache organization [36] (Figure 2), with the aim of supporting data classification while avoiding some of the overheads associated to TLB-based mechanisms. Such a distributed organization has been previously suggested [8], however it has not been extensively explored. The baseline CMP considered in this work includes per-core L1/L2 TLBs, memory management unit (MMU), L1/L2 caches, and directory cache.



(a) Private two-level TLB structure.



(b) Two-level TLB structure with a distributed shared second level TLB.

Fig. 3: System configurations in the evaluation.

Figure 3 shows how different TLB hierarchies are logically connected for both private (Figure 3a) and distributed shared (Figure 3b) last-level (L2) TLBs. The interconnect network considered is a mesh topology. By employing a shared L2 TLB, the classification is naturally obtained through unicast messages to the corresponding L2 TLB bank, issued upon every L1 TLB miss. The sharing status is stored in the TLB structure and accessible for all requesting cores through the network.

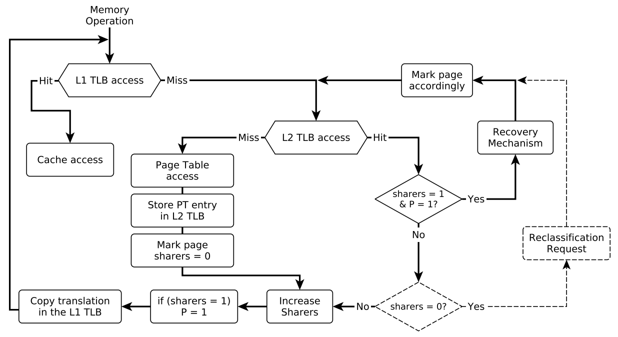


Fig. 4: Block diagram of the general working scheme under a memory operation for shared last-level TLBs. Dashed arrows and boxes operate only under the usage predictor.

4.1 Basic protocol

The main idea behind this work is to leverage the shared L2 TLB in order to track the sharing information, thus naturally classifying memory accesses into private or shared at page granularity. To this end, a counter is associated with each second-level entry, recording up-to-date information about potential page sharers.

Figure 4 outlines the actions required by the proposed classification scheme in order to resolve memory operations through the TLB hierarchy, including recovery and reclassification operations, explained in Section 4.2 and Section 4.3, respectively.

Upon a memory access on a given core, prior to accessing the cache hierarchy, a TLB lookup is performed to look for the virtual-to-physical address translation. If it hits the L1 TLB, the sharing status information for that page is retrieved. Otherwise, on an L1 TLB miss, the L2 TLB is consequently accessed. Requesting the translation to the L2 TLB increases its sharers count. Therefore, whether there is a miss in the L2 TLB (and thus the page table in main memory has to be accessed), or there is a hit and no other L1 TLB currently holds the page (i.e. there is no other potential sharer), the page ends with a single sharer, and is thus marked as private. Otherwise, if the page had one or more sharers upon the reception of an L1 TLB miss request, it ends with more than one sharer and is marked as shared. In either case, the sharing status is specified alongside the response message containing the virtual-to-physical page translation to the upper TLB hierarchy level. The translation is finally stored in both TLB levels.

When the L2 TLB suffers an eviction, the sharing status is lost. In this case, in order to avoid classification inconsistencies, all L1 TLB entries holding the related page must be evicted (invalidating the corresponding L1 cache blocks). An inclusive policy between L1 and L2 TLBs is therefore recommended when assuming a shared L2 TLB. This policy brings also the advantage of exploiting inter-core sharing patterns in parallel applications, hitting on subsequent accesses from different cores to the L2 TLB.

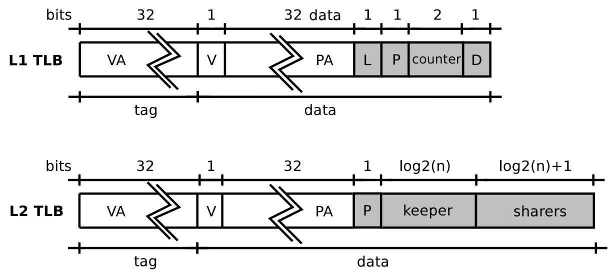


Fig. 5: L1 and L2 TLB entries, with the extra fields in gray for shared second level TLB structures.

As commented above, the L2 TLB keeps track of the sharing status for each page in the TLB hierarchy. Therefore, the L2 TLB entries require some extensions, as illustrated in Figure 5. A *Private* (P) bit specifies whether the page is private (bit set) or shared (bit clear). A *sharers* field counts the number of current sharers of a page, allowing shared-to-private page reclassification. The *sharers* field is updated after every miss or eviction on L1 TLBs. This implies that L1 TLB evictions must be notified to the L2 TLB. This is essential to accurately unveil reclassification opportunities when the page ceases to have sharers. The page sharing status is updated in the L2 TLB according to the sharers count. Finally, a *keeper* field contains the identity of the holder of a private TLB entry. The *keeper* field helps to avoid broadcasts when updating the sharing state in the private TLBs when a transition to shared occurs. The *keeper* is updated every time the L2 TLB receives a request for a page and the current number of sharers is zero. In this case, the requester core's TLB becomes the new *keeper*.

The total storage resources required by this information is $1 + log_2(N)$ bits for the *sharers* field (counting from 0 to N sharers, both inclusive), $log_2(N)$ bits for the *keeper* field, and one bit for P, where N is the number of cores in the system. This adds up to $2 + log_2(N) * 2$ total bits. Since the TLB entry data field often contains some unused bits [37] that are reserved, hardware overhead may be avoided by taking advantage of them. Anyhow, the shared second level TLB entry format requires only 10 bits assuming a 16-core CMP, or 22 bits for a 1024-core CMP. Therefore, for a TLB to support our proposed classification approach, the area overhead increases logarithmically with the system size, representing $\sim$14% or $\sim$23% of the L2 TLB area for a 16-core or a 1024-core CMP respectively, according to CACTI [38].

4.2 Coherence recovery mechanism

This section reviews the coherence recovery mechanism for our proposed shared TLB classification scheme, which is required by the coherence deactivation technique that we employ as data optimization for our study case (see section 2.3).

If a non-coherent page (i.e. private) transitions to a coherent state (i.e. shared), coherence status needs to be recovered in order to avoid the presence of untracked blocks (not cached in the directory). In our case, when an L1 TLB miss request reaches a private L2 TLB page entry, the sharing status may evolve to shared and thus, a coherence recovery is initiated (see Figure 4). A special *recovery* request is issued to the current page *keeper*. If an L1 TLB receives a *recovery request*, all blocks pertaining to that page in the L1 cache are flushed to avoid inconsistencies. Then, after flushing all the page's blocks in the private cache, the sharing status in the L1 TLB is securely set to shared and a *recovery response* is sent to the corresponding L2 TLB bank. Upon the reception of the *recovery response*, page sharing status is updated in the L2 TLB, which becomes coherent (i.e. shared). Thus, directory cache can start tracking all blocks accessed for that page.

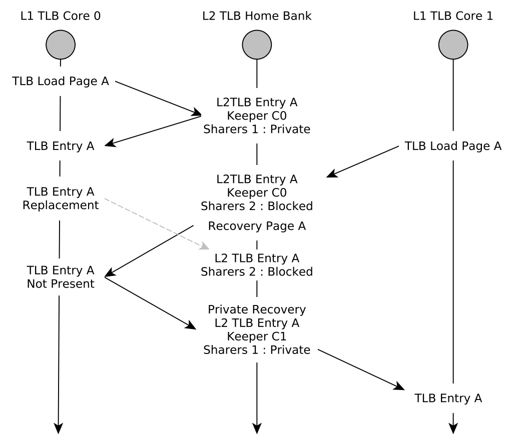


Fig. 6: Coherence recovery mechanism resolved to Private. Page A in the keeper (*C0*) is evicted prior to receiving the Recovery message and thus, the Recovery is resolved to Private and the keeper is updated.

During a recovery process, a race may occur if the keeper evicts its TLB entry due to a conflict. Therefore, if the recovery request misses on the keeper's L1 TLB, a special *recovery* response is sent back with no further actions required (Figure 6). Then, the requester becomes the new keeper and the page remains as private.

4.3 Shared TLB usage predictor

The described classification mechanism is so far dependent on the size or associativity of the L1 TLBs. In order to decouple TLB size from classification accuracy, a usage predictor is required. Unlike usage prediction for private TLB structures, which relies on broadcast TLB-to-TLB requests to discover the usage status of a TLB entry, shared TLB structures send a single request to the L2 TLB bank. As a consequence, the TLB usage predictor needs to be reworked and tuned for the new environment.

We propose a shared TLB usage predictor (SUP), employing a 2-bit saturated counter only for L1 TLB entries (see Figure 5). SUP also adds a new *Disused* (*D*) bit, which is set when the corresponding counter saturates for the first time. Altogether, 4 additional fields are required per L1 TLB entry, for a total of 5 bits which are insensible to system size. Thus, the area overhead is only ∼4% of the L1 TLB area.

Every time the D bit is set, a *disuse announcement* is sent to the L2 TLB, which decreases the *sharers* counter. Therefore, we operate under the assumption that the page is not going to be reaccessed soon from a core that has already fallen into disuse. If an access occurs, the counter is reset, but the D bit remains set. No more messages will be sent to the L2 TLB bank while it remains so, even if a disused TLB entry is evicted (i.e. disused translations evict silently), since the sharers count has already been appropriately decreased. Therefore, the *sharers* field tracks the number of pages currently in use, unveiling early reclassification opportunities. However, reclassification requires probing L1 TLBs as they might have been reaccessed from the time they fell into disuse.

Reclassification process. In particular, whereas an L1 TLB miss hits in the L2 TLB bank and the *sharers* count is 0, a reclassification process is triggered (see Figure 4). First, in case the P bit remains unset (i.e. the page has been shared), and a reclassification opportunity arises, it starts by sending a broadcast request, excluding the requester that initiated the reclassification. L1 TLBs reply according to the information of their predictor counters. On the one hand, if the counter is saturated (i.e. the page is currently not in use) the L1 TLB invalidates the page entry (flushing the blocks in the L1 cache) and responds with a NACK. On the other hand, a reaccessed L1 TLB unset its D bit and responds with an ACK. If not present, it still has to respond with a NACK. When all responses are collected, the *sharers* count is updated accordingly to the number of positive acknowledgments received. Finally, the miss that originated the reclassification is resolved (which increases the sharers count once more), setting P according to the resulting sharers count (i.e. private if there is 1 sharer and shared otherwise).

Conversely, when a reclassification process starts for a private page (the sharers count is 0 and the P bit is set), only the keeper needs to be probed with a unicast request. Therefore, a reclassification process might be used in order to keep the classification as private for longer. As in a broadcast reclassification, the *keeper* responds with a positive or negative acknowledgement depending on its usage prediction. Finally, the sharers count in the L2 TLB is either kept as 0 (and the page is brought as private for the requester, which becomes the new *keeper* and the only sharer that is accounted for), or restored to 1 (and the page subsequently transitions to shared).

In addition, the *forced-sharing* optimization can be also adapted in the context of a shared L2 TLB, although we expect that premature invalidations are not going to hurt system performance under this configuration. The reason is that *disused* entries are only invalidated when the L2 TLB considers that a reclassification probe should take place, naturally acting as a filter for premature invalidations. Nonetheless, L1 TLBs send a *forced-sharing* request to the L2 TLB when the page is accessed and their corresponding entry is found present but invalid. If the miss triggers a reclassification, the probes issued to L1 TLBs just unset the D bits rather than invalidating the translations (independently from the status of the saturated counter). Thus, the page is kept as shared and can be securely reaccessed without incurring extra L1 TLB misses due to predictor-induced invalidations.

4.4 Classification status

The sharing status of a page is managed by the L2 TLB through the P bit and the *sharers* count. This status is updated as a consequence of L1 TLB miss requests, evictions, or disuse announcements, and L2 TLB evictions. Figure 7 depicts the state-transition diagram.

Pages in the L2 TLB can be in four states: (i) not present; (ii) present but not in use in any L1 TLB ($sharers = 0$); (iii) present and private with only one L1 TLB holding the entry ($P = 1$ and $sharers = 1$); and (iv) present and shared with one or several L1 TLBs currently holding the entry ($P = 0$ and $sharers > 0$).

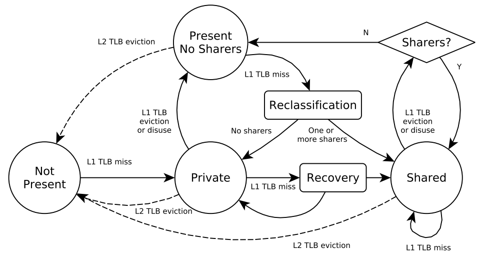


Fig. 7: L2 TLB classification state diagram with SUP.

If the page translation is *Not Present* in the L2 TLB and a miss request is received, the page transitions to *Private* after "walking" the page table. Then, if a different L1 TLB misses while the page is in *Private* state, a recovery mechanism is initiated and the page transitions to *Shared*. Note, thought, that classification might transition back to *Private* if a race condition occurs, as explained in Section 4.2. On the other hand, receiving an eviction or a disuse announcement for a *Private* page causes a transition to *Present No Sharers*. In this state, the L2 TLB acts as a victim TLB for the next missing L1 TLB. When the miss occurs, disused entries might have silently reaccessed the page translation and the classification coherence must be assured by means of a *Reclassification* request. If the reclassification ends up with no sharers, the miss is resolved to *Private*. Otherwise, the miss is resolved to *Shared* and the sharers count is updated. Finally, further L1 TLB miss requests for a *Shared* page leave the page in the same state, just increasing the sharers count. Receiving evictions or disuse announcements for a *Shared* page decreases the sharers count. Finally, the page transitions to *Present No Sharers* only if the count reaches 0.

5 Simulation Environment

We evaluate the classification schemes described in this work through full-system simulations using Virtutech

TABLE 1: System parameters for the baseline system.

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 L1 TLBs	32 sets, 16-way (512 entries)
L1 TLB hit time	1 cycle
Unified L2 TLB	128 sets, 16-way (2048 entries)
L2 TLB hit time	3 cycles
Prediction timeouts	250K, 50K, 10K, and 2K cycles
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

Simics [39], along with the Wisconsin GEMS toolset [40], which enables detailed simulation of multiprocessor systems. The interconnection network has been modeled using the GARNET simulator [41]. We simulate a 16-tile CMP architecture that implements directory-based cache coherence and employs the parameters shown in Table 1. The TLB sizes are the same whether it is a purely private two-levels TLB hierarchy or it implements a shared second-level TLB. The L2 TLB miss latency considers four memory references to walk the page table, as in the 48-bit x86-64 virtual address space. Cache and TLB latencies have been calculated using the CACTI tool [38] assuming a 32nm process technology. Predictor timeout values in the evaluation are based in the inter-access study in [21]. Every predictor value evaluated corresponds to the number of cycles required to increase the predictor counter field. Thus, four timeouts are required for the field to saturate and the translation entry to fall into disuse.

The evaluation is performed with a wide variety of parallel workloads from several benchmarks suites, covering different sharing patterns and sharing degrees. *Barnes* (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 [42]. *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 [43]. *Blackscholes* (simmedium), *Swaptions* (simmedium), and *x264* (simsmall) come from PARSEC [44]. Finally, *Apache* (1000 HTTP transactions), and *SPEC-JBB* (1600 transactions) are two commercial workloads [45]. All reported experimental results correspond to the parallel phase of the benchmarks.

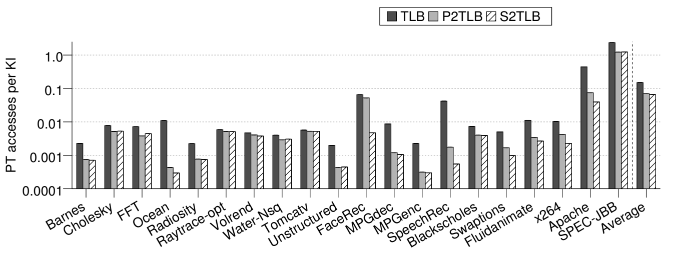


Fig. 8: TLB misses ending up as page table accesses per 1000 instructions. No classification.

6 Evaluation Results

In this section, we first analyze how the aforementioned TLB hierarchies behave prior to applying a classification mechanism. Next, we evaluate how TLB-based classification schemes behave on multilevel TLB hierarchies, assuming both a private L2 TLB, which is a common design nowadays, and a distributed shared L2 TLB of the same size. Specifically, we evaluate our shared TLB usage predictor (SUP) and compare it to previous approaches in order to highlight how the classification is improved, and most of the flaws of the private predictor design are avoided. As a consequence, the scalability of the classification scheme is substantially improved.

6.1 TLB architecture analysis

This section shows how different TLB configurations behave prior to the application of any classification mechanism and the coherence deactivation technique that benefits from it. Specifically, we compare: (i) a system with a single-level TLB with TLB-to-TLB transfers used to accelerate TLB misses but without classification purposes (*TLB*); (ii) a system with per-core private L2 TLBs and, again, TLB-to-TLB transfers to accelerate L2 TLB misses (*P2TLB*); and (iii) a system with private L1 TLBs and distributed shared L2 TLBs (*S2TLB*).

Figure 8 shows the number of accesses to the page table per kilo instructions, i.e. TLB misses that are resolved in the page table (TLB-MPKI). When the fast TLB-to-TLB transfer miss resolution mechanism is implemented, TLB misses are only resolved by accessing the page table if no other TLB is currently holding the translation. Notice that the y axis is plotted on a logarithmic scale in order to discern the different magnitudes of each application. As can be expected, including a second level TLB (*P2TLB* or *S2TLB*) reduces the average number of accesses to the page table, compared to a single-level TLB structure (*TLB*) with fast TLB miss resolution through broadcast TLB transfers. However, in some cases, *S2TLB* can be observed effectively exploiting the size of the L2 TLB over the private TLBs approach. For instance, SpeechRec or Apache, among others, reduce the number of accesses to the page table to a greater extent, showing how a shared TLB configuration helps preventing redundant page translation copies. Finally, the number of TLB-MPKI is remarkably low, less than 0.07 misses on av-

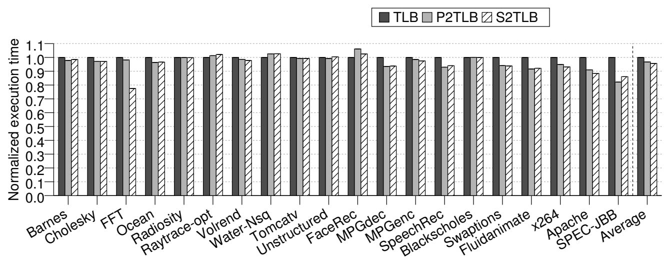


Fig. 9: Normalized execution time. No classification.

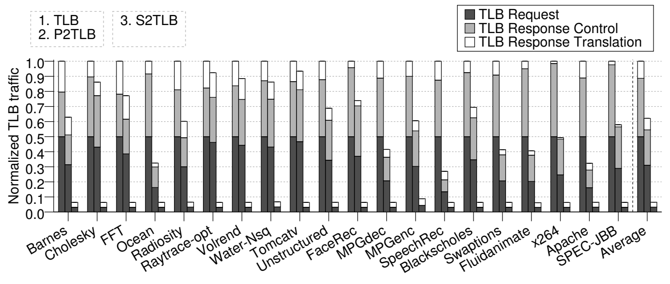


Fig. 10: Normalized traffic attributable to the TLB. No classification.

erage using a two-level TLB structure. Differently, using a single-level TLB retains TLB-MPKI slightly over 0.15 misses on average, despite implementing TLB-to-TLB transfers.

Accessing the page table in the main memory after a TLB miss implies performing an expensive page walk operation. Therefore, when the page table access is avoided, execution time is improved. Figure 9 shows how *S2TLB* slightly improves execution time by only 1% on average compared to *P2TLB* with the same total size, and up to 4.4% compared to the *TLB* scheme. Despite the fact that the shared L2 TLB reduces the number of accesses to the page table, the improvement is comparatively low as a result of the small absolute amount of TLB misses reported in Figure 8. In the case of SPEC-JBB or Apache, where the MPKI reported is greater, the improvement over the *TLB* scheme is more noticeable. Differently, in some benchmarks, as Barnes or SPEC-JBB, a private L2 TLB slightly outperforms a distributed shared L2 TLB scheme. The performance shrinkage occurs on account of the additional latency of accessing a shared L2 TLB, which needs to traverse the network in order to reach the home TLB slice. Therefore, on benchmarks with high L2 TLB hit ratio or accessing a great amount of private data, a greater access latency may hurt system performance, overmatching the potential benefits of inter-core sharing patterns exploitation. On the contrary, as FFT has more accesses to shared pages, *S2TLB* execution time improved.

Finally, TLB-to-TLB transfers significantly increase TLB traffic, since every TLB miss induces a broadcast message and many responses, potentially including many replicated translations. Figure 10 shows all

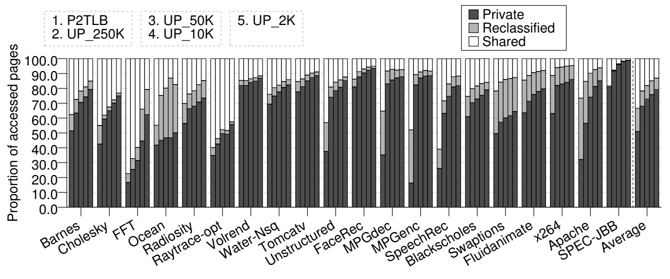


Fig. 11: Private/Shared page classification with private TLBs.

network flits (the flow control unit in which network packets are divided —see Table 1) transmitted across the network, normalized to *TLB*. Essentially, *P2TLB* reduces the TLB traffic by 48.0% compared to *TLB*, as it first relies in the L2 TLB to resolve L1 TLB misses. Therefore, the broadcast TLB miss resolution mechanism is only invoked after an L2 TLB miss. In contrast, L1 TLB misses with a shared L2 TLB are resolved through unicast messages. As a consequence, TLB network traffic with *S2TLB* is reduced to barely 6.5% compared to *TLB*.

6.2 TLB-based classification mechanisms

So far, different TLB hierarchies have been analyzed, without any classification scheme whatsoever. This section analyzes how temporal-aware TLB-based classification techniques work assuming systems with two-level TLBs, and evaluating their potential when applied to coherence deactivation for private data. First, using a private L2 TLB; then the classification scheme proposed in this paper for distributed shared L2 TLBs.

6.2.1 Private second level TLB

This section analyzes how a system with a purely private two-level TLB hierarchy behaves alongside the TLB-based classification scheme, as explained in Section 3. The quality of the classification scheme is tested by applying it to deactivate coherence maintenance.

Classification accuracy. A good, first, general metric to determine the effectiveness of TLB-based classification is the amount of private data detected, provided they do not allow *false private* classification (see Section 2.2). Figure 11 shows the amount of pages classified as private or shared, both with and without a TLB usage predictor (*UP*), including several predictor timeout values ranging from 250,000 to 2,000 cycles. The characterization is extended to discern the amount of shared pages that are reclassified to private at least once (*Reclassified*). By differentiating reclassified data we offer more insight into the potential benefits of a temporal-aware classification. Note that for a page to be considered private in the figure, it must remain so for the entire execution time.

Particularly, *P2TLB* classifies slightly above 50% of all accessed pages as *Private*, and 15.7% of shared pages are *Reclassified*. However, when a usage predictor is employed, classification accuracy depends on the page

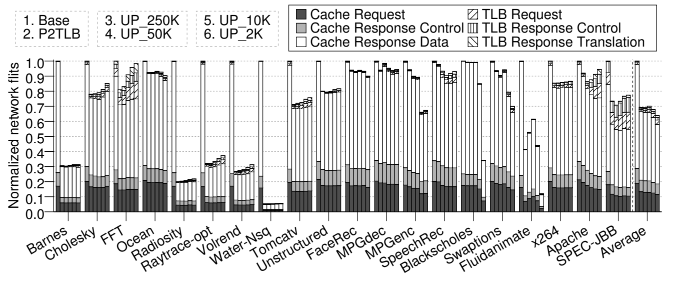


Fig. 12: Normalized network traffic under coherence deactivation. Classification with private TLBs.

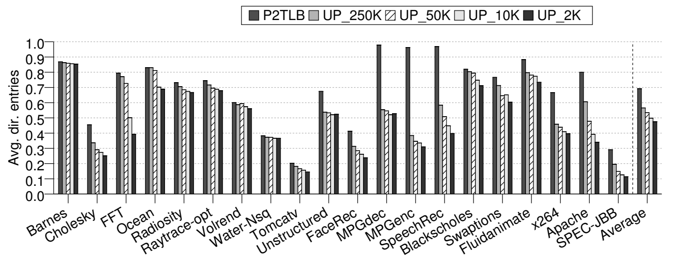


Fig. 13: Average amount of directory entries under coherence deactivation. Classification with private TLBs.

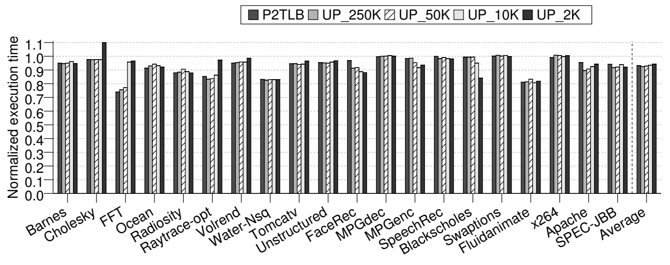


Fig. 14: Normalized execution time under coherence deactivation. Classification with private TLBs.

access patterns, decoupling it from the size and associativity of the TLB. As a consequence, the amount of *Private* pages is increased with *UP_2K* up to 76.6% on average. Private data detection with UP is improved even over *Private* and *Reclassified* pages combined in *P2TLB*. Moreover, *UP_2K* still reclassifies nearly 10% of pages.

Coherence deactivation. Figure 12 shows the total normalized amount of network flits transmitted through the interconnection network, classified into cache- and TLB-related traffic. Applying the classification to deactivate the coherence maintenance reduces cache traffic directly proportional to the amount of private data detected. The baseline system has the same overall configuration but without employing a classification scheme. Specifically, *P2TLB* reduces cache traffic to just 68.46%, on average, when compared to *Base*. Moreover, applying TLB usage predictor further follows this progression, requiring on average just 58.0% of *Base* cache traffic when employing the lowest predictor value. On the contrary, since TLB miss rate is increased due to some additional predictor-induced TLB invalidations, TLB traffic is increased as the predictor value decreases. Particularly, TLB traffic represents 10.4% of the total network traffic for *UP_2K*. Furthermore, TLB traffic overhead for TLB-based classification will not presumably scale horizontally with the system as broadcast cost greatly increases with the number of cores.

As the classification becomes more accurate, the pressure in the cache directory is alleviated, as blocks that pertain to private pages are not stored for non-coherent data (i.e. private) under coherence deactivation. Figure 13 shows how, as expected, *P2TLB* reduces the average number of directory entries required per cycle by 30.8% compared to the baseline system. Moreover, when employing a usage predictor for TLBs, directory usage is further reduced, by up to 52.5% on average for the lowest predictor timeout.

Coherence deactivation also entails an improvement in execution time, since reducing the directory storage requirements contributes to a significant reduction of coverage misses (induced by directory evictions) in the cache structure. Additionally, *P2TLB* effectively reduces L2 TLB miss penalty by means of TLB-to-TLB transfers. Figure 14 shows how *P2TLB* improves execution time by 6.8% compared to the baseline. TLB usage predictor does not contribute into a system performance reduction (even damaging it to some extent). This is due to the fact that directory ceases being a performance bottleneck (i.e. almost all coverage misses are prevented) with plain *P2TLB*, whereas prediction implies some extra overheads. Nonetheless, this factor is mostly on account of the optimization chosen to test the different classification approaches (i.e. coherence deactivation). Of course, alternative (or additional) data optimization could be applied, ultimately offsetting prediction overheads with the potential benefits of a more accurate classification.

Conclusion. To sum up, employing the classification mechanism for private two-level TLBs reveals how a TLB usage predictor (UP) is required to perform an accurate private classification. The TLB usage predictor timeouts considered in the analysis increase classification accuracy as its value decreases, and consequently, directory usage and cache traffic are improved when the classification is applied to coherence deactivation. However, TLB-based classification comes with some overheads, as it increases TLB traffic due to the TLB-to-TLB miss resolution mechanism, and incurs some performance degradation. Despite the fact that this does not completely discourage the use of a low predictor timeout over its potential benefits, it evidences the flaws of a TLB-based classification scheme for private TLB structures. Finally, TLB-to-TLB transfers are not supposed to scale with the number of cores in the system.

6.2.2 *Distributed shared second level TLB*

This section evaluates a TLB-based classification for distributed shared L2 TLBs using the shared TLB usage

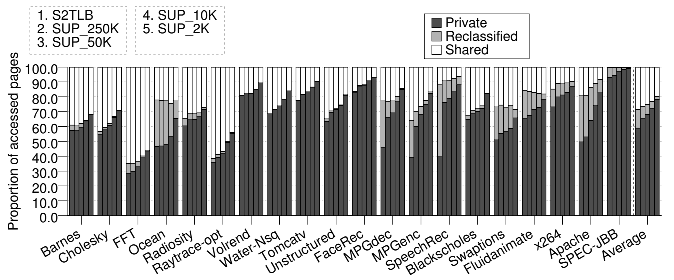


Fig. 15: Private/Shared page classification with distributed shared last-level TLB.

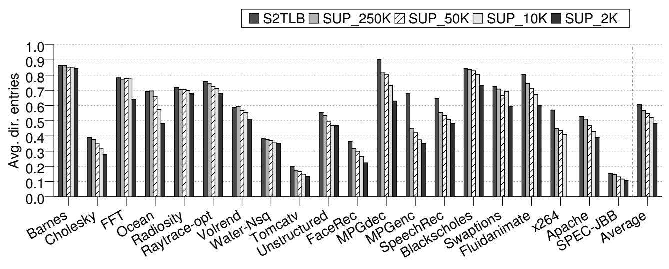


Fig. 17: Average directory usage under coherence deactivation. Classification with distributed shared L2 TLB.

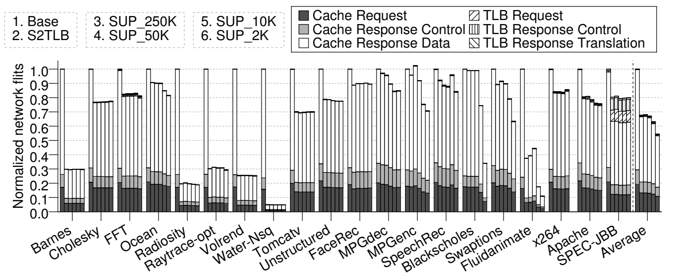


Fig. 16: Total flits injected under coherence deactivation. Classification with distributed shared L2 TLB.

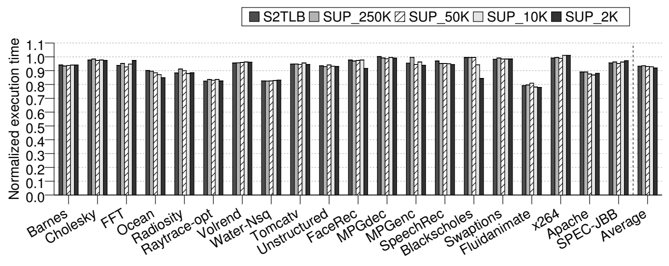


Fig. 18: Execution time under coherence deactivation. Classification with distributed shared L2 TLB.

predictor presented in section 4.3.

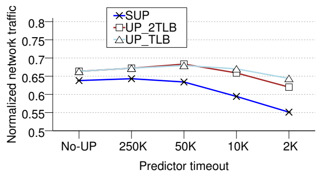
Classification accuracy. Figure 15 depicts how pages are classified into private and shared for different classification mechanisms in a scenario with a distributed shared L2 TLB. The evaluation is focused in our classification mechanism detailed in Section 4 (*S2TLB*), and different predictor timeouts for a shared TLB usage predictor (*SUP*). On the one hand, *S2TLB* classifies 59.4% of pages as *Private*, and 14.6% as *Reclassified*, showing how it performs a precise and adaptive classification at page level. On the other hand, *SUP* increases the proportion of *Private* pages up to a 78.1%, but reduces *Reclassified* pages to merely a 2% of all accessed pages for a 2,000 cycles timeout. However, *SUP* accuracy for private data detection beats even *UP* for private TLB structures, as can be seen comparing with Figure 11. Note that *SUP* relies on the shared L2 TLB as a filter for short-term reclassifications. Our proposal for shared TLB structures initiates a reclassification process only when the home L2 TLB tile is accessed and found in a *Present No Sharers* state (see Section 4.4). Even so, a reclassification process may still fail to transition to private again if a core reaccesses a disused page. Conversely, a TLB predictor for purely private TLB structures is more dynamic and forceful, which favors short-term reclassifications, albeit possibly hurting system performance.

Coherence deactivation. Figure 16 shows the total normalized network usage and its classification into cache or TLB messages when the classification is applied to coherence deactivation. *Base* is our baseline system with the same overall configuration, including a distributed shared second level TLB, but without data classification nor coherence deactivation. Classification mechanisms for shared TLB structures reduce the network traffic since they avoid the costly TLB-to-TLB broadcast transfers. *S2TLB* TLB traffic represents only 1.8% of the total. Furthermore, *SUP* prevents the TLB traffic increase attributed to lower predictor timeouts, keeping it to as much as 2.3% of the total traffic for the lowest considered timeout, while the cache traffic is halved. All in all, *SUP* improves classification accuracy while significantly reducing traffic overhead by avoiding unnecessary TLB invalidations. Therefore, fewer TLB misses are induced by the predictor, which represents a far more scalable approach in terms of traffic.

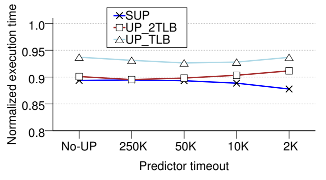
Figure 17 shows the average number of required directory entries, in this case normalized to the baseline system with a distributed shared last-level TLB. Particularly, *S2TLB* prevents the storage of 39.3% of directory entries per cycle on average. Furthermore, when applying our shared TLB usage predictor, directory storage requirements are reduced to merely 51.7% with a low predictor timeout value, obtaining similar figures compared to those of a purely private TLB structure, as seen in Figure 13 (roughly 1% difference).

Finally, Figure 18 shows the execution time of different applications employing our TLB-based classification for shared TLB structures under coherence deactivation. *S2TLB* reduces execution time by 6.8% compared to a baseline system without coherence deactivation. Additionally, *SUP* further contributes to better system performance, reducing execution time up to 8.2%, even using a 2,000 cycles predictor timeout.

Conclusion. The classification scheme introduced in this paper for shared TLB structures benefits system performance as expected when applied to coherence deactivation. Moreover, our shared TLB usage predictor (SUP) enhances private detection without performance

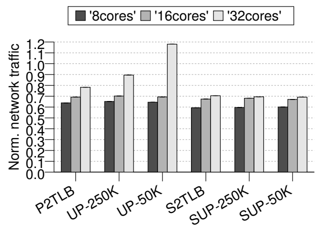


(a) Total flits injected.

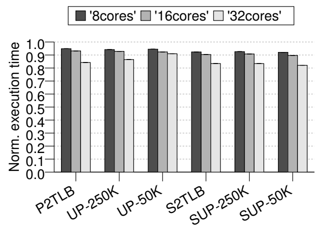


(b) Normalized runtime.

Fig. 19: Classification overhead analysis.



(a) Total flits injected.



(b) Normalized runtime.

Fig. 20: Scalability analysis.

degradation.

6.3 Private against shared TLB classification

This section offers a comparative analysis of how the classification mechanisms behave for both private and shared TLB structures when applied to coherence deactivation, focusing on the usage predictors herein detailed.

Overhead analysis. We show results for the TLB-based classification mechanisms with a single level and two private level TLB structures (*UP_TLB* and *UP_2TLB*, respectively), and the shared TLB classification mechanism (*SUP*). All results presented in Figure 19 are normalized to a baseline with a single TLB level without coherence deactivation, which employs broadcast TLB-to-TLB transfers for TLB miss resolution purposes.

Figure 19a shows the total traffic for the different TLB structures. We observe how, in this case, both purely private TLB structures behave similarly, reducing traffic over 30% compared to the baseline. As TLB usage predictor decouples the page lifetimes from TLB size or associativity, including a private second TLB level hardly contributes to reduce network traffic. If a TLB entry is invalidated, either in the first or the second TLB level, accessing that page would result in a miss in the TLB hierarchy, thus invoking the broadcast TLB-to-TLB resolution mechanism that is causing the traffic overhead. Conversely, *SUP* reduces both TLB and cache traffic over the approach for private TLB structures. *SUP* efficiently avoids premature cache flushes and reduces TLB invalidation frequency even with low predictor values, provided that the distributed shared second TLB level acts as a filter for premature invalidations. Moreover, TLB misses are resolved through unicast messages, which represent a far more scalable solution. Particularly, *SUP* reduces total traffic to just 55.1% with a 2,000 cycles predictor timeout.

Similarly, Figure 19b shows the average execution time evolution for the different predictor timeout values considered. We observe how applying the shared TLB usage predictor (*SUP*) reduces execution time as the predictor timeout decreases, a reduction of up to 12.23% over baseline with a 2,000 cycles timeout. Differently, even though *UP_2TLB* predictor also improves the classification accuracy, it moderately increases execution time compared to not employing a predictor, dropping its performance gainings to just 8.8% with a 2K predictor timeout. In other words, the performance divergence with *SUP* reaches 3.5%. *UP_TLB* follows a similar trend as *UP_2TLB*, but with lower performance gainings over baseline. This is due to the predictor induced invalidations, which result in higher TLB miss rates and, ultimately, into performance shrinkage.

Scalability analysis. This section shows how the different classification schemes scale when deactivating coherence maintenance. Due to the slow pace of the simulation tools, this study is only performed using SPLASH 2 benchmarks and scientific application. All the results in Figure 20 are normalized to a system of the same core count, with a purely private TLB structure that does not perform data classification; therefore, coherence maintenance is not deactivated. Lower predictor timeouts have not been included in this study to favor better figure readability, although they follow a similar trend.

The page classification mechanisms based on purely private TLB structures evaluated in this paper aim for small- or medium-scale systems. As can be seen in Figure 20a, broadcast messages issued after every TLB miss do not scale for larger systems. Particularly, when employing a usage predictor for TLBs, which induces extra TLB misses, up to 50.7% more traffic is issued compared to *P2TLB* for a 50,000 cycles predictor timeout, ultimately offsetting the benefits of deactivating coherence for a 32-core CMP. Conversely, our classification mechanism for shared TLB hierarchies completely avoids broadcast requests, relying solely on unicast messages after missing on the first TLB level. As a consequence, *SUP* not only avoids traffic overhead for larger systems, it even reduces traffic by 30.9% compared to the baseline for *SUP-50K*.

Consequently, leveraging a shared last-level TLB for page classification improves global system performance over its competitors, specially with *SUP*, which improves private data detection, avoids the overheads of the prediction scheme for private TLB structures, and exploits inter-core sharing patterns. Therefore, *SUP* contributes to reduce execution time by 17.8% for a 50,000 cycles timeout with a 32-core CMP (Figure 20b), nearly 9% better performance than *UP* with the same predictor timeout.

Conclusion. Employing SUP on a system with shared last-level TLBs has proved to squeeze coherence deactivation potential to its maximum, providing a better classification while avoiding the majority of its overheads, specially for low predictor timeout values, ultimately representing the best-suited classification scheme for large-scale CMPs.

7 Conclusions

In this paper we have evaluated different approaches to the classification of data into private or shared at page level in CMPs with different TLB hierarchies. We revisit TLB-based classification for purely private multi-level TLB structures, which get similar figures to previous works. Furthermore, we propose a TLB-based classification approach which leverages the use of a distributed shared last-level TLB. A shared last-level TLB structure allows to naturally discover the sharing access pattern through the L1 TLB misses received from different cores to the home L2 TLB tile. Additionally, we show the interest of employing a usage predictor for TLBs in order to achieve an accurate classification independent from TLB size, and propose a predictor for systems with distributed shared last-level TLBs (SUP).

Our proposal improves classification accuracy while avoiding some of its overheads. Specifically, SUP improves the classification of private pages up to 78.1% on average. As a consequence, execution time is improved by 12.2% and traffic is reduced to only 55.1% over a single level baseline system when SUP is applied to coherence deactivation. Conversely, a TLB-based classification for a private two-level TLB structure slightly hurts execution time as long as the predictor timeout value is reduced, progressively hurting system performance up to 3.5% over SUP.

Acknowledgments

This work has been jointly supported by the MINECO and European Commission (FEDER funds) under the project TIN2015-66972-C5-1-R and TIN2015-66972-C5-3-R 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] 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.

[2] ——, "Increasing the effectiveness of directory caches by avoiding the tracking of non-coherent memory blocks," *IEEE Transactions on Computers*, vol. 62, no. 3, pp. 482–494, mar 2013.

[3] S. Demetriades and S. Cho, "Stash directory: A scalable directory for many-core coherence," in *20th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2014.

[4] 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.

[5] 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.

[6] D. Lustig, A. Bhattacharjee, and M. Martonosi, "Tlb improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level tlbs," in *ACM Transactions on Architecture and Code Optimization (TACO)*, Jan. 2013.

[7] 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.

[8] Y. Li, R. Melhem, and A. Jones, "PS-TLB: Leveraging Page Classification Information for Fast, Scalable and Efficient Translation for Future CMPs," in *8th Int'l Conf. on High-Performance and Embedded Architectures and Compilers (HiPEAC)*, Jan. 2013.

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

[13] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi, "End-to-end sequential consistency," in *39th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2012, pp. 524–535.

[14] 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.

[15] 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.

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

[17] J. Zebchuck, B. Falsafi, and A. Moshovos, "Multi-Grain Coherence Directory," in *46th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2013.

[18] M. Davari, A. Ros, E. Hagersten, and S. Kaxiras, "An efficient, self-contained, on-chip, directory: DIR$_1$-SISD," in *24th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2015, pp. 317–330.

[19] A. Ros and A. Jimborean, "A dual-consistency cache coherence protocol," in *29th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, May 2015, pp. 1119–1128.

[20] ——, "A hybrid static-dynamic classification for dual-consistency cache coherence," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. PP, no. 99, Feb. 2016.

[21] A. Esteve, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Efficient tlb-based detection of private pages in chip multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, Mar. 2015.

[22] A. Esteve, A. Ros, A. Robles, M. E. Gómez, and J. Duato, "Tokentlb: A token-based page classification approach," in *Int'l Conf. on Supercomputing (ICS)*, Jun. 2016.

[23] 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.

[24] 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.

[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] A. Ros, B. Cuesta, R. Fernández-Pascual, M. E. Gómez, M. E. Acacio, A. Robles, J. M. García, and J. Duato, "Emc^2: Extending magny-cours coherence for large-scale servers," in *17th Int'l Conference on High Performance Computing (HiPC)*. Goa (India): IEEE computer society, Dec. 2010, pp. 1–10. [Online]. Available: http://ditec.um.es/ aros/papers/aros-hipc10.pdf

[27] T. W. Barr, A. L. Cox, and S. Rixner, "Spectlb: A mechanism for speculative address translation," in *38th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2011.

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

[29] "Technical Resources: Intel Xeon Processors," http://goo.gl/ZlM8I2, [Online; accessed Nov-2015].

[30] A. Bhattacharjee and M. Martonosi, "Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors," in *18th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, feb 2009, pp. 29–40.

[31] ——, "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.

[32] "Advanced Micro Devices," http://www.amd.com, [Online; accessed Nov-2015].

[33] "Intel Corporation," http://www.intel.com, [Online; accessed Nov-2015].

[34] *ARM Architecture Reference Manual ARMv8-A*, 2015.

[35] "Sun Microsistems," http://www.sun.com, [Online; accessed Nov-2015].

[36] C. Kim, D. Burger, and S. W. Keckler, "Nonuniform cache architectures forwire-delay dominated on-chip caches," in *23rd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2003, pp. 99–108.

[37] AMD, "AMD64 architecture programmer's manual volume 2: System programming," whitepaper, Jun. 2010.

[38] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," HP Labs, Tech. Rep. HPL-2008-20, Apr. 2008.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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 (IISWC)*, Oct. 2005, pp. 34–45.

[44] 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.

[45] 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.

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 memory hierarchy designs, and memory consistency 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. She has published more than 50 conference and journal papers. She has served on program committees for several major conferences. 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. 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*.