Temporal-Aware Mechanism to Detect Private Data in Chip Multiprocessors

Alberto Ros*, Blas Cuesta†, María E. Gómez‡, Antonio Robles‡, José Duato‡

*Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia
aros@ditec.um.es

†Intel Labs Barcelona
blasx.cuesta@intel.com

‡Department of Computer Engineering, Universitat Politècnica de València
{megomez,arobles,jduato}@disca.upv.es

***Abstract*—Most of the data referenced by sequential and parallel applications running in current chip multiprocessors are referenced by only one thread and can be considered as private data. A lot of 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 depend to a large extent on the amount of detected private data. However, the mechanisms proposed so far do not consider thread migration and the private use of data within different application phases. As a result, a considerable amount of data is not detected as private. In order to make this detection more accurate and reaching more significant improvements, we propose a mechanism that is able to account for both thread migration and private data within application phases. Simulation results for 16-core systems show that, thanks to our mechanism, the average number of pages detected as private significantly increases from 43% in previous proposals up to 74% in ours. Finally, when our detection mechanism is used to deactivate the coherence for private data in a directory protocol, our proposal improves execution time by 13% with respect to previous proposals.**

I. Introduction

Chip-multiprocessors (CMPs) usually implement the shared-memory programming model which requires coherence maintenance among data in private caches. In these systems coherence is ensured by means of cache coherence protocols. These protocols are one of the most important design aspects of CMPs because the have a huge impact on system performance and scalability. Thus, since the core count is continuously increasing, coherence protocols must scale to provide increasing performance.

The two typical approaches for maintaining coherence are snooping and directory. Snooping protocols are based on broadcasting messages and consequently the bandwidth requirements of the network grow dramatically with the core count. A lot of efforts have been made to reduce them [1], [2], [3]. An alternative solution to reach this goal is the use of directory protocols, which keep track of cached memory blocks and do not require the use of broadcast messages. Nevertheless, they require a directory structure whose size grows exponentially with the system size. Again, some works have addressed this problem [4], [5], [6].

Another important design aspect of CMPs is the organization of the last-level cache (LLC). Usually, it is organized as a *NUCA cache* (Non-Uniform Cache Architecture) to reduce the number of off-chip accesses [7]. Since the average access latency to NUCA caches increases with the number of cores some works have also addressed this inefficiency [8], [9], [10].

A common technique used to improve coherence protocols and the organization of the LLC is classifying data accessed by applications into private (i.e., accessed by only one thread) and shared (i.e., accessed by two or more threads) [3], [5], [10], [11], [12], [13], [14], [15], [16]. The key observation behind these works is that a significant fraction of the memory blocks referenced during the execution of sequential and parallel applications is private. Hence, they implement mechanisms to detect such private blocks in order to handle them in a more efficient and fast way. For instance, Cuesta *et al.* [5] propose to prevent directory information of private blocks from being included in the directory, which leads to more effective, smaller, and faster directories; Hardavellas *et al.* [10] and Li *et al.* [11], [14] keep private blocks in the NUCA bank of the requesting processor to reduce the access latency to NUCA caches; and Kim *et al.* [3] avoid issuing broadcast messages when accessing private blocks, thus leading to network traffic reductions.

The effectiveness and benefits of all these proposals depend to a large extent on the amount of private data detected. However, the mechanisms that they use to detect private data have some limitations drawbacks: (i) they have high storage requirements; (ii) they are unable to exploit core proximity of CMPs, which makes them slow; and (iii) they do not account for temporality in memory references, which significantly reduces the amount of detected private data, and consequently, the final impact on performance.

In this paper we propose a different mechanism to detect private blocks that deals with these three problems. First, unlike previous proposals, our mechanism only keeps the information about private data in the TLB, which reduces the storage requirements. Hence, it does not require extra information in a directory-like structure [12], [13], [16] or in the page table [3], [5], [10], [17].

Second, it exploits the proximity of cores in a CMP. Upon a TLB miss, our mechanism collects information about the use of pages from the TLB of other cores. This also allows us to get the missing address translation, which accelerates the TLB miss resolution. Differently, in the related proposals it is mandatory to retrieve the information about private pages from the page table, which prevents them from being employed along with an efficient TLB-to-TLB miss resolution mechanism [18], [19].

Third, our mechanism accounts for temporarily private pages (e.g., as a consequence of thread migration or temporarily private data within application phases) 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 [20], [21], we claim in this work that a temporal-aware detection mechanism is fundamental to achieve a good accuracy when detecting private pages.

Cycle-accurate simulations of a 16-core CMP running a large variety of scientific and commercial workloads show that our proposal is able to increase the number of pages classified as private from 43% (with previous OS-based mechanisms) up to 74%, on average. Additionally, our results show that the TLB-to-TLB miss resolution mechanism solves 60% (on average) of TLB misses without requiring access to the page table, which results in execution time improvements of 11.9% (on average). When applying our mechanism to the previously described coherence optimizations, it can achieve much more significant improvements. For example, when it is applied to improve the efficiency of directory caches, the average execution time is reduced by 17% over the baseline and by 13% over previous proposals.

The rest of the paper is organized as follows. In Section II we review the private-shared classification mechanisms and its applications. Section III describes the TLB-to-TLB transfer technique used in this paper to accelerate TLB misses. The proposed temporal-aware mechanism is presented in Section IV, and details of its application for different optimizations are given in Section V. Section VI introduces our simulation methodology and Section VII shows the performance results. Section VIII discusses other system configurations. Finally, Section IX reviews the related work, and Section X draws some conclusions.

II. Background

Among the different mechanisms used to detect private accesses or blocks, we compare ours against those aided by the OS [5], [10], [3]. Unlike hardware-based approaches [13], [12], 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 [11], [14] 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, annotating the requesting core in the page table. If another core different to the first requestor accesses a private page, then it is re-classified as shared. To this end, each page table entry adds a PS bit that indicates the page state (private or shared) and a field that stores the id of the first core that accessed the page. The PS 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 target application 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.

A. Improving directory cache effectiveness

Recent directory-based protocols only keep directory information for a small fraction of the memory pages, those that have at least a block cached, in small directory caches with the aim of reducing the memory overhead [22]. 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 blocks 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 [5]. 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, that are cached by a single core, have not been tracked by the directory. In particular, the OS triggers a *recovery mechanism* that just evicts the corresponding blocks from the cache of that core that holds the blocks of the recovered page.

B. 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 [10]. 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.

C. 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 [3]. 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, a recovery mechanism is not 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.

III. TLB Miss Resolution through TLB-to-TLB Transfers

CMPs have different characteristics than traditional multiprocessors. One key aspect is that core-to-core communication is much faster in CMPs. While in traditional multiprocessors this latency is about hundreds of cycles, in CMPs it is just a few cycles. This means that TLBs in a nearby core can be accessed with low latency. Due to this, in this paper we propose, on a TLB miss, to obtain the page sharing information in a fast way from the TLBs of other cores, and as a consequence also the page address translation. Some recent proposals have made the most of this short-latency translation, although with different aims to the ones pursued in this paper [18], [19].

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 [19]. 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]).

Our mechanism works as follows. On a TLB miss, a page table walk process is started to get the address translation from the page table. However, at the same time, 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 response is received, the memory request can proceed and the page table walk is cancelled. 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 VII-C.

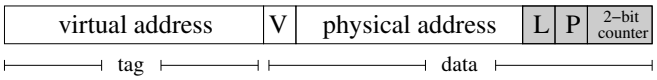

Fig. 1: TLB entry with the extra fields in gray

As for cache misses, TLB misses also allocate an entry in the Miss Status Holding Register (MSHR), where the information about ongoing misses is placed. Although the TLB miss can be solved once the first TLB response is received, to avoid complex race conditions the MSHR entry is not deallocated until all TLB responses have been received.

IV. 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 they are 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 VIII-D.

A. 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), as shown in Figure 1. 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 an entry for that page, their P bits are clear.

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, the page_info request is sent to the other TLBs. The goal of this request is to get the address translation, but also the information about the use of the page by the other cores. If any of the requesting core is going to access the page, then it is marked as private. Otherwise, the page will be classified as shared.

B. Page access prediction

To detect the private use of data within different phases of applications, each processor has to predict if it is going

TABLE I: 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)

to accessing a page in a near future. To make predictions independent of the TLB size, we introduce a TLB decay technique, similar to the one proposed by Kaxiras *et al.* to save power in data caches [24]. 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 (see Figure 1). 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 I shows the possible TLB entry states, the response messages to page_info requests, and the information included in the responses.

C. 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 2. 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 2 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 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

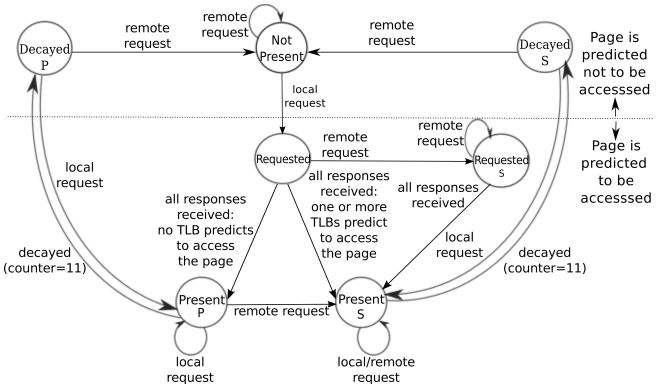

Fig. 2: TLB state transition diagram.

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 other TLBs are not notified about the evictions.

D. 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. Hence, when a TLB entry transitions to *Not Present*, the cached blocks belonging to the corresponding page are evicted from L1 cache. This will hardly affect performance (as shown in Section VII-B) 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.

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 (see Figure 1). 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.

E. 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 classification considering thread migrations. When a migration happens, all the TLB entries corresponding to pages accessed by the process of the migrated

TABLE II: 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

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. If in the interim some pages are accessed in the previous core (e.g., by another thread of the same process), the decay counter of the accessed pages will be reset and the pages will be consider as shared when requested by other core.

V. Requirements for private-shared optimizations

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. 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. This actions are summarized in Table II.

When applied to improve the cache directory effectiveness (*coherence deactivation*) all blocks within the page becoming shared that are stored in the L1 cache need to be flushed (and written back to the home LLC bank if they are dirty). This is because cached private blocks are not tracked by the directory cache. By flushing them we restore the coherence status.

When applied to reduce NUCA access latency (*reactive NUCA*), blocks cached in the local NUCA bank must be evicted and written back to the home NUCA bank (if the local and the home bank are not the same). This is because when a page becomes shared its mapping in the NUCA cache may vary. In addition, when evicting a TLB entry the blocks belonging to the evicted page must be also replaced from the local NUCA bank (and from the L1 cache due to the TLB-L1 inclusion policy).

Finally, when applied to reduce traffic in snooping protocols (*subspace snooping*), no recovery action is needed. This is because, once the page becomes shared, cache-misses for blocks belonging to that page will be treated as coherent, and the request will be broadcast. This will discover in a natural way the existence (if any) of other cached copies of the requested block.

VI. Simulation Environment

We evaluate our proposal with full-system simulation using Virtutech Simics [25] along with the Wisconsin GEMS toolset [26], which enables detailed simulation of multiprocessor systems. The interconnection network has been modelled using the GARNET simulator [27]. We simulate a 16-tile CMP architecture implementing directory-based cache coherence and with the parameters shown in Table III. 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 [28] assuming a 32nm process technology.

TABLE III: 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

We evaluate our proposal 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 [29]. *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 [30]. *Blackscholes* (simmedium), *Swaptions* (simmedium), and *x264* (simsmall) come from PARSEC [31]. Finally, *Apache* (1000 HTTP transactions), and *SPEC-JBB* (1600 transactions) are two commercial workloads [32]. All the reported experimental results correspond to the parallel phase of benchmarks.

VII. Evaluation Results

We first analyze how the TLB-to-TLB miss resolution improves performance with the aim of knowing its contribution to the improvements of our mechanism. We also study the influence of the decay technique for TLBs. Then we show how our classification mechanism can improve the accuracy in the classification of private pages. Finally, we study the benefits entailed of our proposal when applied to the *coherence deactivation* technique.

A. TLB-to-TLB Miss Resolution

Here, we analyze the benefits and overheads of the TLB-to-TLB miss resolution mechanism by a sensitivity study done on the TLB size which ranges from 256 sets to 64 sets (all of them are 4-way associative).

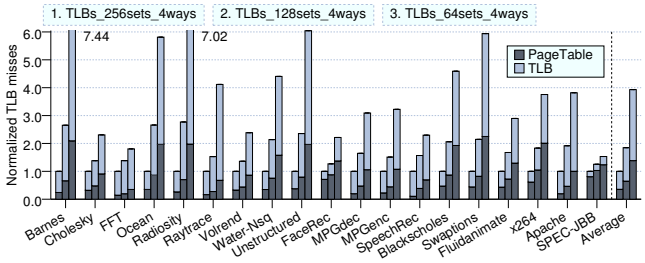

Fig. 3: Distribution of TLB misses resolved by other TLBs or by the page table

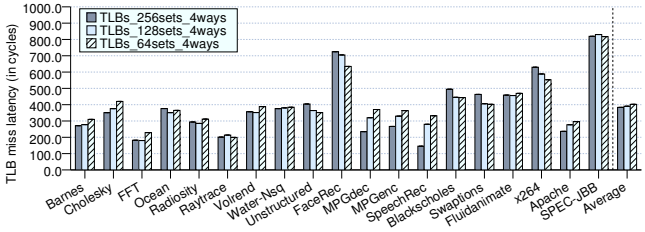

Fig. 4: Average TLB miss latency

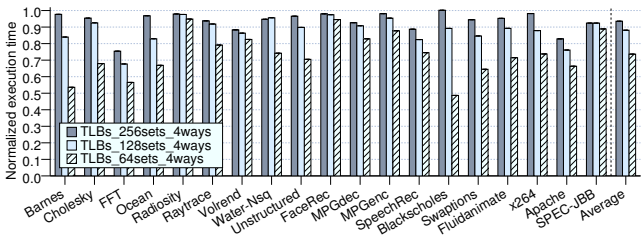

Fig. 5: Improvements in execution time when using TLB-to-TLB transfers

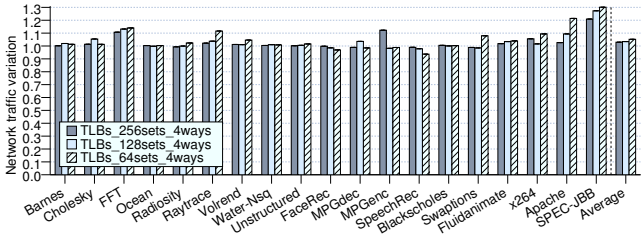

Fig. 6: Variations in network traffic when using TLB-to-TLB transfers

TLB-to-TLB miss resolution saves access to the page table. Figure 3 shows the number of TLB misses normalized to those caused when using a 256-set TLB. Each bar shows the distribution of misses resolved by another TLB or the page table. We can see that the number of TLB misses increases for smaller TLBs. The number of TLB-to-TLB resolutions also increases, thus making its percentage almost independent of the TLB size. Particularly, 61.7%, 64.8%, and 64.8% of misses are resolved from a neighbor TLB for 256-, 128-, and 64-set TLBs, respectively.

Since almost two out of three TLB misses are resolved without accessing the page table, their average latency is considerably reduced, as plotted in Figure 4. We can see that the applications with a significant amount of TLB misses resolved by the page table (*FaceRec* and *Spec-JBB*) are the ones with higher average latency. Similarly, the ones that are mainly resolved by means of TLB-to-TLB transfers (e.g., *FFT* and *Raytrace*) have a low miss latency. On average and regardless of the TLB size, the TLB miss latency can be reduced by 60% when the TLB miss resolution mechanism is applied.

Low TLB miss latency results into reductions in execution time, as Figure 5 plots. In this figure, each bar shows the reduction in the number of cycles when compared to a configuration that does not implement TLB-to-TLB transfers but with the same TLB size. 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.

Finally, Figure 6 shows the normalized network traffic due to the issue of page_info requests. Since TLB misses are not frequent, the traffic increase 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. Finally, we believe that this low traffic overhead can be paid since the reduction in execution time is significant.

B. 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. 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 with a low value.

Figure 7 shows the number of TLB misses for configurations with different decay timeouts, all of them normalized with respect to the absence of decay timeout (*TLB* bar). We classify the misses into *3C* misses (cold, capacity, and conflict) and *Forced* misses, which come as a consequence of invalidations of decayed TLB entries. As can be seen, shorter decay timeouts lead to increase the number of TLB misses, which could finally impact negatively performance. As expected, a shorter timeout only increase *Forced* misses. Observe that a timeout greater than or equal to 50000 is able to minimize the number of *Forced* misses ($\leq 3\%$, on average). Also, the negative impact of the decay technique will be partly mitigated because those extra misses will probably find the address translation in the TLB that caused the invalidation of the decayed entry.

Invalidations and replacements of TLB entries cause the eviction of L1 cache blocks due to our TLB-L1 inclusion policy. However, these events only occur for decayed or LRU entries. Since these entries have not been accessed for a long

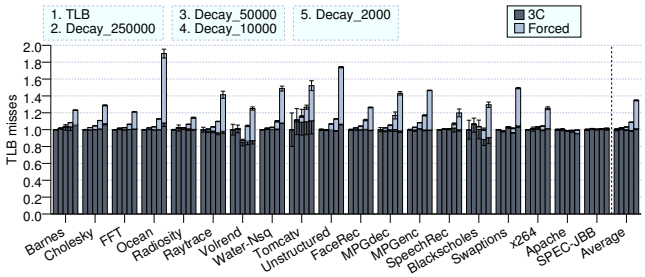

Fig. 7: TLB misses

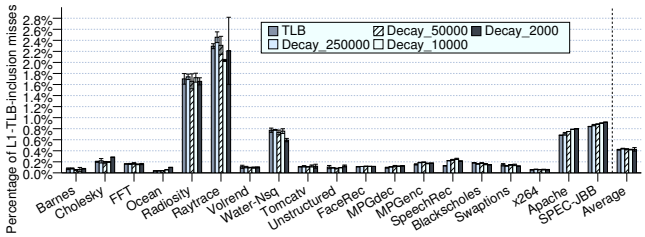

Fig. 8: Percentage of L1 cache misses due to our TLB-L1 inclusion policy

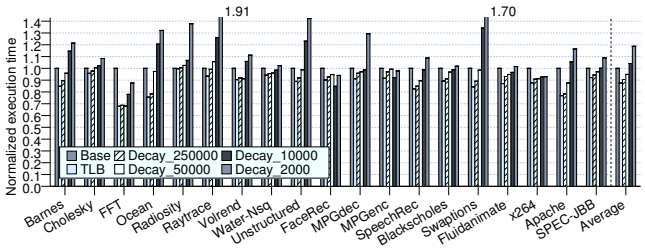

Fig. 9: Execution time degradation of the decay technique

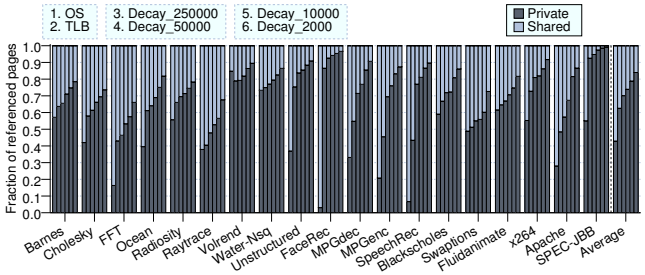

Fig. 10: Private vs. shared pages

time, the L1 cache is likely not holding many blocks from the invalidated or evicted page. Also, these blocks are probably not going to be accessed soon. Hence the effect of flushing is negligible. As we can see in Figure 8, only 0.4% (on average) of cache misses are due to the flushing effect of the inclusion policy. We also can observe that this number does not vary with the decay timeout.

Figure 9 shows the impact in execution time of our TLB-based mechanism for different decay values compared to the traditional TLB miss resolution that accesses the page table (*Base*). We can see that a decay value smaller than or equal to 10000 cycles mitigates the benefits of TLB-to-TLB transfers. As expected, the TLB-based classification without decay (*TLB*) gets the most significant improvements. The decay technique degrades performance by 3% and 7% for 250000 and 50000 timeouts, respectively. However, it is still better than the base approach.

Although a low decay timeout leads to extra TLB misses, it also helps to detect more private pages. Figure 10 shows the pages considered as private and shared according to each mechanism. Particularly, we show numbers for the OS-based detection (*OS*), our basic idea without employing decay techniques (*TLB*), and a more elaborate detection using decay timeouts of 2500000, 50000, 10000, and 2000 cycles. We consider 128-set TLBs. If a page has been considered as shared at least once during the execution of the application, it will be seen 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 20% more pages as private than *OS*. Additionally, employing decay techniques we can improve this detection up to 41% (for a short decay timeout of 2000 cycles) but at the cost of increasing TLB misses, as shown above. Finally, it is important to note that thread migration is rare in the evaluated applications due to its short execution time. As explained in Section IV-E, thread migration would have a dramatic impact for the OS classification, but not for our proposal.

C. A case of study: coherence deactivation

Previous sections analyze the impact on performance of our proposal without taking advantage of the temporal-aware page classification. This section evaluates its impact when applied to the coherence deactivation scheme proposed for directory caches [5]. 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, and therefore, we need another statistics. For the coherence deactivation proposal we are interested in the number of required directory entries, as shown in Figure 11. Particularly, this figure shows the average number of directory entries required per cycle normalized with respect to no deactivation at all. We can observe that 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: *TLB* avoids the storage of 38.3% of entries and, when applying the decay technique, up to 78.3% of entries do not require to be tracked.

This large reduction in the number of required directory entries results in less directory evictions and consequently less invalidations in the private caches. Invalidations cause extra cache misses (named as *coverage* misses). Since coherence deactivation reduces the number of coverage misses, the execution time can be improved as shown in Figure 12. The figure is normalized with respect to the baseline configuration that does not detect private pages. The OS-based classification can just reduce 5.8% the execution time. When compared to

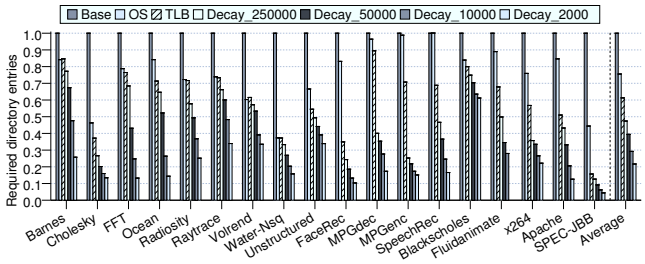

Fig. 11: Average directory entries required (per cycle)

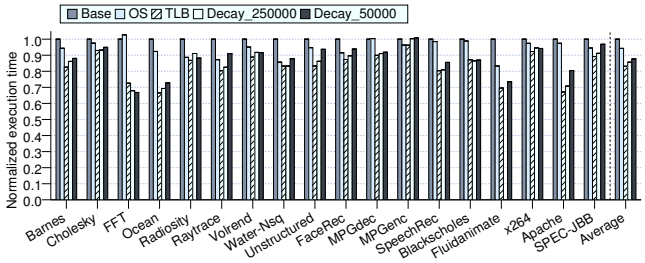

Fig. 12: Execution time improvements for coherence deactivation

the base configuration, *TLB* reduces the execution time by 16.9% due to both the reduction in the required directory entries and the TLB miss resolution mechanism. Furthermore, when compared to the OS-base classification, *TLB* reduces the execution time by 13%. For most applications the introduction of the decay technique hurts performance with respect to *TLB*. *FFT* is the exception. This is due to the "large" size of the directory. 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.

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 mitigates this increase. Figure 13 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, a decay timeout of 250000 can outperform *TLB* by 5% for a 16-set directory. As we can see, the smaller the directory is, the smaller decay timeout is recommended.

As a conclusion, we can affirm 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 optimizations are higher, a smaller decay timeout can translate into more benefits.

Configuration	256	128	64	32	16
Base	1.00	1.03	1.06	1.31	1.90
TLB	0.83	0.86	0.92	1.07	1.38
Decay_250000	0.85	0.88	0.93	1.06	1.32
Decay_50000	0.88	0.91	0.94	1.08	1.30
Decay_10000	0.98	0.99	1.03	1.09	1.30

Directory sets

Fig. 13: Normalized execution time depending on the directory size and the decay timeout employed

VIII. Discussion

A. 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 2 % 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 (similar to a directory) to avoid broadcasts, or by employing a Token-like protocol [33] to avoid most TLB responses.

B. Large or multilevel private caches

Our proposal scales with the size of the private caches since on a TLB eviction, only the content of the evicted page needs to be flushed (due to the TLB-L1 inclusion policy). Therefore, the number of visited cache lines on a TLB eviction is given by the page size (64 cache lookups for 4KB pages), and not by the cache size.

Additionally, our proposal 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.

C. Large pages

Our proposal can also work in systems implementing large pages. Naively, the eviction of the entries for large pages from the TLB will require a more expensive flushing. However, more elaborate approaches can be employed to lessen the cost of flushing, particularly in terms of latency. 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, considerably reducing the amount of required lookups.

D. 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 before accessing the cache (e.g., physically indexed, physically tagged –PIPT– caches).

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.

IX. 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 [34] or improve data placement [11], [14]. In [11], 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 [14] 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 [34], private data is not stored at the LLC with the aim of avoiding cache thrashing for private blocks. Unlike our approach, these works mark statically data as private either by the memory allocator or at compile time, when privacy of some data cannot be guaranteed.

SWEL [12] 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, so removing the need of 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 [13] 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 [16] 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 II. 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 [17]), because of the requests sent by the directory upon private-to-shared changes.

Our proposal discovers the private-shared page status by benefitting from the resolution of TLB misses through TLB-to-TLB transfers. Related to the fast resolution of TLB misses, Synergistic TLBs [19] employs the snooping of other TLBs to propose a distributed-shared organization of TLBs. Also, UNITD Coherence [18] employs the TLB snooping to integrate the existing cache coherence protocol with the maintenance of the TLB coherence. Differently, we employ the TLB snooping to account for the temporality of private accesses and, so, improve the private-shared classification. In [35], neighbour 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 [36]. 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 [37]. Both can be use in conjunction with our proposal.

Some other proposals, such as *RegionScout* filters [38], *Region Coherence Arrays* [39], and *RegionTracker* [40], also share with the ours the idea of removing coherence for private data. However, these approaches require additional storage resources and are focused only on reducing snoop traffic in snooping-based coherence. On the contrary, our proposal can be applied for different purposes, and not only for directory reductions, and does not require additional storage resources, just a few additional bits at the TLBs.

X. Conclusions and Future Work

This paper proposes an effective temporal-aware detection of private pages for medium-scale CMPs based on predicting whether the page is going to be accessed in the near future by the other cores in the system on a TLB miss. This way, our mechanism classifies pages accessed by several cores at different points in time (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, that results in a significantly increase in the number of pages considered as private compared to an OS-based classification (from 43% to 74%). Furthermore, unlike OS-based page detection, our approach is compatible with recently proposed low-latency TLB miss resolution mechanisms. When applied to improve the effectiveness of directory caches by deactivating coherence for private blocks, our proposal can remove up to 78% entries from the directory cache, reducing execution time by 13%, on average, compared to an OS-based mechanism.

Our future work focus on the scalability of the proposed classification for larger CMPs. Directory-like information could be added at page level by means of shared TLBs. Also, Token-like protocols could be employed at the TLB level to avoid responses from TLBs not holding page translations.

These two options will result in reducing traffic and latency in many-core systems.

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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *10th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Oct. 2002, pp. 211–222.

[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] C. Liu, A. Sivasubramaniam, and M. Kandemir, "Organizing the last line of defense before hitting the memory wall for CMPs," in *11th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2004, pp. 176–185.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[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] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] J. Meng and K. Skadron, "Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling," in *Int'l Conference on Computer Design (ICCD)*, Oct. 2009, pp. 282–288.

[35] 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.

[36] 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.

[37] 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.

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

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

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