TokenTLB: A Token-Based Page Classification Approach

Albert Esteve
Parallel Architectures Group
Universitat Politècnica
de València.
alesgar@gap.upv.es

Alberto Ros
Departamento de Ingeniería y
Tecnología de Computadores
Universidad de Murcia
aros@ditec.um.es

Antonio Robles
arobles@disca.upv.es

María E. Gómez
megomez@disca.upv.es

José Duato
jduato@disca.upv.es

Department of Computer Engineering
Universitat Politècnica de València.

ABSTRACT

Classifying memory accesses into private or shared data has become a fundamental approach to achieving efficiency and scalability in multi- and many-core systems. Since most memory accesses in both sequential and parallel applications are either private (accessed only by one core) or read-only (not written) data, devoting the full cost of coherence to every memory access results in sub-optimal performance and limits the scalability and efficiency of the multiprocessor.

This work proposes TokenTLB, a page classification approach based on exchange and count of tokens. The key observation behind our proposal is that, opposed to coherence management, data classification meets all the benefits of a token-based approach without the burden of complex arbitration mechanisms, which has discouraged the implementation of token-based coherence protocols in commodity systems. Token counting on TLBs is a natural and efficient way for classifying memory pages. It does not require the use of complex and undesirable persistent requests or arbitration, since when two or more TLBs race for accessing a page, tokens are appropriately distributed classifying the page as shared. TokenTLB also favors shareability of translation information among TLBs, which improves system performance and constrains much of the TLB traffic compared to other broadcast-based approaches. It is achieved by requiring only TLBs holding extra tokens provide them along with the page translation (about one response per TLB miss). TokenTLB effectively increases blocks classified as private up to 61.1% while allowing read-only detection (24.4% shared-read-only blocks). When TokenTLB is applied to optimize the directory, it reduces the dynamic energy consumed by the cache hierarchy by nearly 27.3% over the baseline.

ICS '16, June 01-03, 2016, Istanbul, Turkey
© 2016 ACM. ISBN 978-1-4503-4361-9/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2925426.2926280

CCS Concepts

•Computer systems organization → Parallel architectures;

Keywords

Data classification; Token counting; TLB; Private-shared; Read-only data

1. INTRODUCTION

Chip multiprocessors (CMPs) are composed of an ever growing number of cores. They require multi-level cache hierarchies for performance reasons and support a shared memory model for ease of programmability. Shared memory models require scalable cache coherence protocols to provide high performance. Directory-based protocols are the best suited for the new scalability challenges [14, 19, 47, 53]. They require less network bandwidth compared to snooping-based protocols. However, they do not distinguish whether the accessed data is private or shared. Consequently, performance and scalability opportunities are not completely exploited.

A large number of recent proposals employ a classification of data and/or accesses into private or shared in order to overcome scalability limitations of current CMPs [8, 14, 15, 16, 17, 18, 20, 21, 23, 24, 26, 27, 28, 36, 42, 43, 44, 45, 52]. The key idea in these works is that the nature of private and shared data is different, and therefore, references to this data can be optimized according to its nature. Data classification has consequently become a key mechanism to design scalable and efficient multiprocessors.

Ideally, the classification should be performed with low-overhead in terms of traffic, performance, and area. However, data classification often entails high storage requirements to track the sharing status and/or does not account for data sharing status transitions from shared to private [14, 20, 23], thus diminishing the classification accuracy and ultimately limiting the potential performance benefits. Furthermore, some classification mechanisms store its sharing status on the directory or the cache memory [8, 16, 21, 36, 52], which limits its applicability. In general, the sooner the data classification is obtained for a memory access, the better. Finally, write detection has also been explored, extending classic private-shared dichotomy by discerning written condition for data, extending the scope of the classification scheme and improving its effectiveness [15, 20]. Note that accesses to read-only data represent an important fraction of memory accesses.

In addition, address translation latency is added to the critical path of memory accesses and a lot of effort has been made to reduce its impact, such as multilevel TLB hierarchies, prediction in super pages [35], TLB prefetchers [11], shared TLB structures [10], Synergistic TLBs [48], or TLB-to-TLB transfers [18, 42]. Some of these proposals are based in slight changes on the TLB organization in order to reduce the number of TLB misses. In a different manner, Synergistic TLBs and TLB-to-TLB transfers focus on reducing TLB miss penalty through inter-core TLB cooperation.

Specifically, TLB-to-TLB transfers are suited for adaptive data classification, aiming for coherence deactivation [14, 15], an effective approach recently proposed to increase directory scalability in CMPs. However, these transfers flood the network with responses after every broadcast TLB miss, many of them being multiple replicated messages with the page translation, thus critically increasing network consumption. Even though TLB misses are infrequent (only 2% of TLB accesses are misses), this does not scale to large systems. In addition, data reclassification from shared to private is not immediate, but it is postponed until the page is evicted from main memory and accessed again, thus penalizing classification accuracy.

In this paper we propose TokenTLB, a novel classification mechanism implemented directly in the TLB structure and inspired by Token coherence [30, 32, 34] protocols. TokenTLB is based on the observation that, unlike Token coherence, applying tokens for classification does not require issuing persistent requests nor complex arbitration mechanisms. Persistent requests are a special type of request meant to solve races occasionally caused when several cores want to write data at the same time. These requests are a source of complexity for the coherence protocol, being one of the main causes why Token coherence has not been implemented in commodity systems. However, TokenTLB avoids these races by only aiming at classifying data. When two or more TLBs race for accessing a page, tokens are naturally distributed among the TLBs, and the page is consequently classified as shared. In other words, classification does not require an owner token. The main contributions of TokenTLB are:

1. TokenTLB is designed as a token-based TLB page-level classification approach. It allows shareability among TLBs, which accelerates page translations, favoring a better system performance.

2. TokenTLB reduces network consumption compared to previous similar TLB-based proposals. Only TLBs holding extra tokens provide them along with the page translation, which leads to about one response per TLB miss.

3. TokenTLB extends classification characterization by detecting write accesses to pages, while performing an adaptive classification based on count and exchange of tokens. Token-based classification makes it possible to identify naturally and immediately a shared TLB page entry transitioning to private, being the first adaptive classification mechanism with write detection proposed to the best of our knowledge.

4. TokenTLB introduces a predictor capable of resolving TLB misses through unicast messages, thus increasing scalability. The predictor relies on a small buffer called Token Predictor Buffer (TPB), which is in charge of short-time storage of potential token-holding TLBs.

Simulations of a cycle-accurate 16-core CMP running a large variety of scientific and commercial workloads show that TokenTLB increases the number of blocks classified as private at miss time up to 61.1%, and shared-read-only blocks up to 24.4%. Furthermore, when the classification is applied to coherence deactivation it reduces the average directory entries requirements to merely a 28.8%, and overall cache hierarchy consumption by 27.3% over the baseline. Finally, TPB inclusion has proved to further reduce TLB request traffic by nearly 20% over base TokenTLB approach. In particular, TokenTLB combined with TPB only generates, on average, less than one response per TLB miss. When the translation is resolved in the page table no TLBs are expected to respond. Therefore, TPB provides, in conjunction with TokenTLB, a more Sscalable classification approach.

2. MOTIVATION AND RELATED WORK

2.1 Data Classification

Data classification mechanisms are gaining interest as they allow many optimizations regarding block management based on their sharing status. There are many recent examples in the literature showing the huge variety of applications for a classification scheme. Specifically, Kim *et al.* [23] avoid requesting coherent data through broadcast messages on *snooping* protocols when accessing private blocks, thus leading to network traffic reductions. Alternatively, Y. Li *et al.* [27] introduce a small buffer structure close to the TLB, namely partial sharing buffer (PSB). When a page becomes shared it will feasibly be present on the PSB upon a TLB miss, obtaining the page translation with both lower latency and lesser storage resources. Moreover, Hardavellas *et al.* [20] and Kim *et al.* [26, 28] keep private blocks on the local NUCA bank in order to reduce access latency to NUCA caches. Ros and Kaxiras [46] 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. Finally, Cuesta *et al.* [14, 15] propose to avoid directory storage of private blocks, therefore deactivating coherence maintenance for those blocks and leading to smaller and faster directories, namely *Coherence deactivation.*

Most proposals described above use classification approaches that take advantage of currently existing OS structures (i.e., TLBs and page table) in order to perform the page classification and store the page status, and therefore they do not require additional hardware structures. Differently, compiler-assisted approaches [26, 28] deal with the difficulty of knowing at compile time (a) whether a variable is going to be accessed or not, and (b) in which cores the data will be scheduled and rescheduled. Furthermore, *directory-based* approaches [8, 16, 21, 36, 52] only reveal the sharing status of data after accessing the cache or directory structure, therefore limiting its applicability to optimizations where

	A-priori	Read-Only	Adaptive	Accurate
Directory	✗	✓	✓	✓
TLB	✓	✗	✓	✓
OS	✓	✓	✗	✓
Compiler	✓	✓	✓	✗
TokenTLB	✓	✓	✓	✓

Table 1: Properties of classification schemes

the a-priori knowledge of the status of the accessed data is not required. Finally, approaches based on the properties of programming languages [44, 45], despite of being very accurate, are not applicable to most existing codes. However, OS-based approach performs a run-time classification for any code, thus avoiding these difficulties.

The main problem of the OS-based classification is that it performs a non-adaptive classification. When a page transitions from private to shared it remains in that state for the rest of the execution time (unless evicted from main memory). In applications running for a long time, many pages may be considered shared at some point along the execution, thus neglecting the advantages of the classification.

In order to perform an adaptive classification that accounts for temporarily private pages and thread migration, *TLB-based classification* [18, 42] was introduced, relying on TLB-to-TLB transfers to inquire other cores' TLBs in the system to naturally discover whether blocks belonging to a page may be currently stored on a remote cache and therefore the page is shared, or, on the contrary, the page is currently private. TLB-to-TLB transfers are based upon the observation that core-to-core communication in CMPs is much faster compared to traditional processors. Other works benefit from this observation with different aims [38, 48]. In addition, a TLB decay mechanism [22] was introduced in order to accurately predict when a page is not going to be accessed in the near future, and thus improve the private detection by avoiding accounting those entries as potential sharers.

However, TLB-to-TLB transfers generate replicated responses, possibly including the translation, from every core in the system after every TLB miss. With lower TLB decay timeouts (which increment the TLB miss rate) network consumption increases dramatically [18]. Furthermore, frequent broadcast requests and responses are not supposedly scalable to large-scale systems, as the number of message-passing steps increases proportionally with the system size. Also, page reclassification to private requires the translation to be completely removed from all TLBs in the system to occur, thus limiting the accuracy of the classification mechanism. Finally, write detection is not explored, limiting the classification scheme to the private-shared dichotomy for an adaptive approach.

Table 1 summarizes the main properties of the state-of-the-art classification approaches compared to TokenTLB. In the first place, knowing the classification *prior* to accessing the cache is critical to the applicability of the classification mechanism. Some approaches store the sharing status in the directory, and thus they cannot be employed for techniques such as *coherence deactivation* [14] or *reactive NUCA* [20] among others. Also, *adaptive* classification entails a huge improvement in the precision of the mechanism, naturally detecting thread migration and data accesses within different private phases. *Read-only* classification (i.e. detecting non-written regions of data) is also a far-reaching property, as shared-read-only blocks can account up to 48.7% of all accessed blocks [15]. However, no previous TLB-based classification approach explores write detection. Finally, compilers need to be conservative, because they do not have information about data sharing status at run-time, therefore they are not compelled to perform an *accurate* classification, as privacy cannot be always guaranteed.

2.2 Coherence Deactivation

On current large CMPs, the directory cache suffers from scalability issues. Directory area and latency overhead increase in order to avoid evictions, as the eviction of a directory entry usually entails the invalidation of blocks on the lower memory hierarchy levels. Due to the limited size or associativity of directory caches or the lack of a backup directory, a system with large number of cores may produce frequent invalidations, which dramatically increases the number of *Coverage* misses [41] (cache misses caused by invalidation on the directory cache due to the limited capacity), and therefore results in performance degradation.

In this regard, *Coherence Deactivation* [14] was proposed, relying on an OS-based page classification scheme in order to identify private (non-coherent) blocks and avoid the storage of those blocks on the directory cache as they do not require coherence maintenance. Thus, directories exploit more efficiently their limited storage capacity as far as the classification mechanism becomes more accurate. Detection of non-coherent blocks was further explored under an OS-based page classification scheme [15] avoiding the tracking of shared blocks that are never modified (read-only), which can account up to 48.7% of total memory accesses.

Furthermore, coherence deactivation was also explored using a temporal-aware classification approach [18, 42], reporting better classification accuracy and, consequently, better directory usage and overall performance.

2.3 Address Translation and TLB Consistency

Address translation is the process regulating the access to physical memory given a virtual address. Modern memory management units (MMUs) divide address space into pages, and therefore divide memory into a set of multi-level hierarchical structures called page tables. Retrieving the page translation (*page table walk*) require multiple memory accesses, as page tables are composed by four hierarchical levels for common 64-bit systems. Moreover, the number of levels required for address translation dramatically grows with systems supporting virtualization (e.g., up to twenty-four memory accesses on x86-64 virtual address space [9], or fifteen memory accesses for the recent 32-bit ARMv7 virtual address space [2]), which is added to the critical path.

The number and size of TLBs is growing to effectively address the increasing application memory footprints and constrain the potential performance loss. Additionally, on page modifications initiated by the operating system it becomes necessary a coherency transaction, namely TLB shootdown, to recover state among TLBs.

Several solutions have been proposed to improve TLB performance, and many of them have similarities with classic cache coherence solutions, revealing how either TLB-based classification or historical TLB consistency could be addressed as TLB coherency [38]. Note how TLB consistency is an infrequent but costly operation, while data classification is a frequent low-cost operation, thus the latter also needs to be efficiently addressed. Furthermore, cache coherence solutions for TLBs also share part their issues.

Snooping-based solutions: are based on broadcasting messages, which dramatically augment bandwidth requirements with core count. There is lot of effort put in order to reduce it [6, 13, 23]. TLB-to-TLB transfers [42] is an example of snooping solution for classification in TLBs, avoiding the penalty of "walking" the page table.

Directory-based solutions: rely on a directory located in the home node to track cached memory blocks. Directories also act as an ordering point for cache requests, naturally avoiding races with unordered networks. Unfortunately, directory size grows exponentially with the system size. It also adds an indirection to the critical path, both in the case of cache and TLB transactions. DiDi [50] is an example of directory-like solution for TLB coherence, which introduces a small shared TLB directory designed to reduce the impact of TLB shootdowns in large-scale CMP systems by enabling lightweight TLB invalidation.

2.4 Token Coherence

TokenB [32] was introduced by Martin *et al.*, capturing the best aspects of snooping and directory protocols: low latency cache-to-cache misses and not reliance on totally ordered interconnects. Token tenure [37] was also proposed by Raghavan *et al.*, relying on an underlying directory cache to track tokens.

Token protocols guarantee coherence safety through token counting: a processor can only write if it holds all tokens in the system and can only read if it holds at least one token for that block. However, as requests are sent to all processors through broadcast requests, they may produce protocol races when contending for a memory block, and thus fail at resolving cache misses. In order to avoid starvation and guarantee cache misses completion, Token protocol invokes *persistent requests* after ten average miss times unsatisfied.

Persistent requests cause major problems, as they require arbitrage, adding some inflexible latency overhead and requiring extra non-scalable structures in the die, being the main reason for its limited roll-out in commodity processors. However, TLBs do not modify translations directly in the TLB cache. Consequently, using tokens for classification and distributing them in the TLBs can avoid these major protocol races. When disputing a page translation they will be simply classified as shared.

3. TOKENTLB

Our goal is to reach all desirable properties for a classification mechanism: performing the classification prior to accessing the cache hierarchy; implementing a fully-adaptive classification able to carry out an accurate reclassification; improving classification characterization by discerning write accesses and recognizing read-only pages; and performing an accurate run-time classification valid for any code.

To this end, this paper proposes TokenTLB, an adaptive classification technique based on token counting. TokenTLB accelerates TLB misses through efficient TLB-to-TLB translation resolution, while coping with the traffic overhead that entails its usage.

3.1 Token-Counting Classification: Concept

TokenTLB associates a fixed number of tokens with each translation entry. In a system with N cores, there must be N tokens per entry. New tokens cannot be generated, and tokens cannot be destroyed. Tokens are exchanged through TLB-to-TLB messages alongside with the page address translation. A TLB page entry is classified according to its token count: private if it holds all tokens (N), shared while holding a subset of all page's tokens (from 1 to $N-1$), and invalid when holding no tokens. Finally, only valid TLB entries (i.e. holding at least one token) may reply to a translation request.

Additionally, in order to track whether the page has been written or not, there is a *written* flag (W) associated with each translation entry, which is sent alongside with the tokens on TLB transactions. The *written* flag increases the classification scope by adding extra classification categories:

- Private Read-only (PR) page: Only one processor is currently accessing the page blocks. All accesses has been loads during the page lifetime.
- Private read-Write (PW) page: Only one processor is currently accessing the page blocks. It has been written at least once during current page lifetime.
- Shared Read-only (SR) page: At least two processors are currently accessing the page blocks. All accesses has been loads during the page lifetime.
- Shared read-Write (SW) page: At least two processors are currently accessing the page blocks. It has been written at least once during current page lifetime.

In sum, TokenTLB (i) accelerates TLB misses through TLB-to-TLB communication, while (ii) optimizes TLB bandwidth requirements, as only TLBs with tokens are in charge of supplying the translation in response to TLB miss requests. Moreover, as page classification relies on the token count, TokenTLB (iii) immediately and naturally reclassifies pages multiple times both from private-to-shared and shared-to-private during each *local page generation time* (i.e. the time spent since the page is first accessed on a core's TLB to the moment it is finally evicted from that TLB) [18], which provides the classification mechanism with full-adaptivity. Finally, TokenTLB squishes classification to its maximum by (iv) extending classic private-shared classification with write detection and applying it for the first time with a fully-adaptive page classification mechanism.

3.2 Token Request upon TLB Miss

TokenTLB initiates the page table walk process after a TLB miss in parallel with a broadcast request snooping other cores' TLBs. Initially, the page table holds all N tokens for each page translation. Consequently, after the first TLB miss for a memory page, the page table delivers all the tokens to the requestor TLB. From now on, tokens are held by TLBs and sent through messages on response to TLB miss requests, spreading across the core' TLBs. When a TLB receives a TLB translation request, it checks if it *owns* the translation entry (i.e. holds the page translation with two or more tokens in it) and if so, it answers the request with a short response message, keeping one token and sending the rest. When the first translation response with tokens is received by the requesting TLB, the page table walk is canceled, tokens are annotated privately in the corresponding page TLB entry, and the memory access proceeds. By doing this, response traffic is constrained as only one TLB (usually the most recent in acquiring the translation) is allowed to

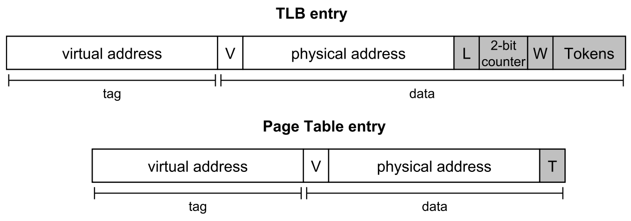


Figure 1: TLB and page table entry format. Shaded fields represent additional fields required.

answer in the common case. Furthermore, page access is unlocked sooner compared to previous similar approaches [18, 42], thus improving execution time (as seen in Section 6.1).

Tokens are stored in an additional TLB field, namely *Tokens*, as seen in Figure 1. This field adds $log_2(N)$ bits to each entry (e.g. only 4 extra bits for a 16-core CMP) in the TLB. When all tokens are given away we rely on the valid/invalid bit (V) of the TLB entry to track it. In the case of the page table, it does not require dedicated hardware, but just one extra bit per entry (whether it has or has not all tokens), namely T, that can be one of the reserved bits in the page table entry. Compared to an OS-based approach with write detection [15], which requires $3 + log_2(N)$ bits, our solution represents far lesser and more scalable overhead.

TLB misses allocate an entry in the Miss Status Holding Register (MSHR), which is deallocated only after acquiring both the page translation entry and at least one token for that page. Therefore, if the page walk process ends without delivering tokens with the page table translation (tokens are held by other TLBs), we have to wait for the first token response. Page access cannot be unlocked without sharing information (i.e. tokens). Note how, in some cases, more than one TLB may respond to the TLB miss request (e.g. when using TLB decay, see Section 3.4.1). Consequently, a page access can be classified as shared although it may be effectively private, as some tokens may be still in-flight. However, once the token reception finishes, if the TLB has all N tokens, the page naturally becomes private.

Additionally, when a page is written for the first time, i.e., the W bit is not set, this bit needs to be set in all copies of the translation stored in other TLBs. This bit remains set until the *global page generation time* (elapsed time from a page is first cached on a TLB to the moment it is evicted from the last TLB in the system) [18] for that page ends. When the write happens in a private page, no actions are required. Otherwise, a message is broadcast to update the W bit in all TLBs holding tokens for that page, which produces a transition to shared-written (SW). Written information is sent alongside tokens as part of the page *sharing* information on TLB miss responses.

3.3 Token Release for Correctness

Tokens are neither created nor destroyed, but transferred. This means that the system must always guarantee the existence of N tokens for any given page translation. The page table either holds all N tokens or none. Otherwise, accessing the page table looking for tokens could become a costly frequent operation, which should be avoided.

When a TLB evicts a page entry that is holding a subset of tokens, a message is sent looking for a new holder for those tokens. On the contrary, if a TLB entry is holding all N tokens, they are sent back to the page table. Consequently, TLB evictions are required to be non-silent, adding some extra traffic consumption. However, non-silent evictions do not hurt system performance, as they are out of the critical path for memory accesses. Contrary to that, non-silent evictions grant a more dynamic classification and are the key for a natural reclassification to private.

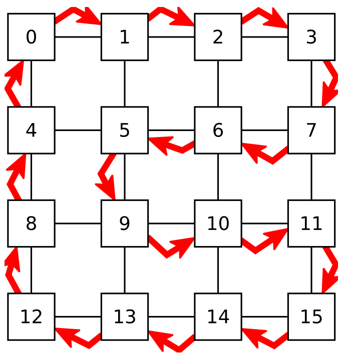


Figure 2: Path that token messages follow after TLB evictions for a 16-cores (4x4) mesh interconnect.

As previously noted, a subset of tokens on an evicting TLB entry must be transferred to a new holder. To do so, a message (*token_evict*) that only carries tokens, but not the translation, is sent to another TLB. Ideally, a TLB receiving a *token_evict* request should only accept tokens if it is already holding a valid TLB entry for that page or if it has a MSHR entry allocated as a consequence of a TLB miss (Section 3.2). Otherwise, the message is sent to the next designated TLB. When the *token_evict* request finds a new holder, tokens are annotated in the new TLB, which responds with an acknowledgment to the original sender.

In order to avoid possible potential livelocks and to minimize traffic, the search of the new holder requires a neat exploration of the network. To this end, we add the concept of a logical network ring, which is a path that covers all nodes in the system minimizing the number of links traversed. The path followed is dependent on the network topology. Figure 2 illustrates an example path for a 4x4 mesh interconnect. This path represents one of the minimum circular routes traversing all nodes in the interconnect.

However, livelocks can still occur when all the TLBs holding a page (and all of its tokens) try to simultaneously evict the associated entry. To solve this problem, tokens being carried on a *token_evict* message can be stored in the corresponding MSHR of a core whose TLB is involved in an eviction process, provided that its ID is greater than the requestor core ID. Token reception is acknowledged afterwards.

Note that the key condition for starvation avoidance in a scenario where multiple evicting TLBs for the same page endlessly pass on their tokens is the core ID comparison. If all TLBs simultaneously evict the same page, all N tokens will eventually end up in the TLB of the core with the greater ID, which will send all tokens back to the page table.

In short, now a TLB evicting tokens can simultaneously store tokens. Thus, when that TLB receives an acknowledgment indicating that the evicting tokens were acquired by another TLB, it has to check its own MSHR again. If it is holding a subset of tokens for that page, it sends a new request message looking for another holder. In case the MSHR is not holding any tokens, the eviction process simply ends.

Finally, observe how a TLB-to-TLB request may fail to acquire tokens after a miss if all TLBs *owning* the translation entry have evicted their tokens short before receiving the request, and thus tokens are currently in-flight (entries with a single token are not allowed to answer). This is prevented by setting a timeout after a TLB miss, which resends the broadcast request if the page has failed to acquire any token at the expiring time. This is not a frequent event and it does not cause a notable increase in network traffic, as shown in Section 6.1. Also, as *token_evict* messages only take tokens and not the translation, the TLB miss is not resolved when receiving a *token_evict*. The missing TLB annotates the tokens in the corresponding MSHR entry and acknowledges its reception, but must remain waiting for a TLB response with the translation or the page table walk process to finish.

3.4 Implementation Optimizations

The following section discusses different *optional* optimizations for TokenTLB and their hardware implementation details.

3.4.1 TLB usage prediction

As sharing condition of a page is settled by the concurrence of accesses to it, a prediction mechanism is required to decide whether or not a page is currently in use by a core (e.g. a TLB entry is valid). To this end, *Forced sharing TLB decay* technique [18] was introduced, similar to the one proposed by Kaxiras *et al.* [22], in order to make predictions independent from the TLB size. Note how, using larger TLBs or including more private TLB levels to the TLB hierarchy makes TLB Decay usage key to perform an accurate classification. This technique can be straightforwardly adapted to TokenTLB. Accordingly, a 2-bit saturated counter is kept by each TLB entry (Figure 1). This counter is periodically increased according to an internal timeout and is reset after every memory access. When the counter saturates, it is considered as *decayed*, implying that it will not be accessed in the near future. When a *decayed* TLB entry is accessed from the network due to a nearby TLB miss, the local TLB gives all of its tokens to the requestor (regardless of whether it *owns* the translation or not) and loses permission to access that TLB entry (it is invalidated).

Due to the variability in the page access intervals among applications or even page live times, *decay* could, in some cases, aggressively invalidate an entry that will presumably be accessed again in the near future. To mitigate this factor, *Forced sharing* TLB decay technique performs a *decay* override when a premature invalidation is detected (i.e. the accessed TLB entry is still present but it has lost all of its tokens). Take into account that our TLBs are implementing a slightly modified LRU policy for replacements, prioritizing TLB entries without tokens when selecting a victim. Thus accessing an invalidated entry is a good indicative for premature invalidation. In this case, a special request is sent, which, rather than delivering all the tokens for decayed entries, is allowed to keep the entry and answer normally to the request, reseting the 2-bit decayed field.

3.4.2 Token Predictor Buffer

In the search for a more scalable classification approach, TokenTLB reduces response TLB traffic and translation replication. However, a broadcast request is still sent after every translation miss. TLBs do not have previous information of possible token *owners* (TLBs holding two or more tokens for a page translation) at the time of the miss, therefore TLB misses still need to be resolved by flooding the network. However, some TLB misses occur shortly after an invalidation, and thus a potential token holder could be anticipated. Consequently, TLB traffic would be further reduced, contributing to a more scalable classification approach. To this end, we introduce a predictor in charge of revealing other TLBs as potential token *owners*. Hitting on the predictor after the TLB miss issues an unicast request, thereby reducing TLB request traffic. This prediction based on previous recent history is similar to the one proposed by Martin *et al.* [31]. Other works also benefit from this observation with different aims [4, 39, 40].

The predictor consists of a small data buffer situated in parallel with the L2 TLB called Token Predictor Buffer (TPB), storing the process ID, virtual address, and core ID. After giving tokens away in non-silent evictions, the receiver becomes a known potential *owner* of tokens and is therefore stored into the TPB. If an L1 TLB miss occurs, both the L2 TLB and the TPB are checked in parallel. If the L2 TLB misses and the TPB keeps information of a potential token holder for that page, an unicast request is sent and the TPB entry is deallocated. If the TLB receiving the request is still holding tokens, it positively responds with the translation and classification information. TLB entries discovered as decayed after being inquired give all of their tokens away. Otherwise, if the consulted TLB is not a token holder anymore, it negatively answers the TLB request and the conventional token broadcast TLB miss resolution mechanism is invoked in parallel with the page walk. The results in Section 6.1 demonstrate how TPB effectively reduces TLB request traffic and cache structure dynamic consumption.

3.5 Classification with Multi-level TLBs

TLB structures in contemporary architectures include at least two private first level TLBs for data and instructions respectively, and a unified private second level TLB. Among the most common architectures we find AMD's K7, K8, and K10, Intel's i7, and Xeon, ARMv7, and ARMv8 [1, 3, 2] which have already adopted private two-level TLB structures.

Therefore, we extended TokenTLB classification mechanism to work with a private L2 TLB. In this scheme, TLB-to-TLB requests are issued only after missing in the L2 TLB.

Inclusion Policy: In order to favor the classification mechanism (avoiding replicated translation entries), we use an exclusive policy between L1 and L2 TLBs. This means we have to lookup both TLB levels when answering a TLB-to-TLB request. However, the small L2 TLB sizes usually considered make this assumption not too time consuming. Also, although a broadcast TLB request is stalling the core in the critical path while waiting for answers, it is unlocked after receiving the first positive answer, greatly diminishing its impact.

TLB entries: L2 TLB entries store the same information as the L1 TLB, including the full context or process ID (PID) bits, and the extra bits required for classification.

Consistency: The TLB hierarchy is shootdown-aware. As both TLB levels implement an exclusive policy, they can be checked in series. Furthermore, this property avoids incurring in wrong (outdated) page classification, by flushing both the TLB and the cache. As a consequence, tokens are transferred back to the page table.

4. COHERENCE DEACTIVATION WITH TOKENTLB

In order to test the benefits of the classification scheme provided by TokenTLB, we apply it to the *Coherence Deactivation* mechanism, which is proved to improve directory usage under a non-adaptive OS-based classification scheme [15]. According to this classification approach, only blocks belonging to shared written (SW) pages may require coherence maintenance, dramatically increasing data accesses in which coherence maintenance is not necessary.

As previously noted, page classification provided under TokenTLB can naturally transition from shared-to-private or private-to-shared multiple times during every TLB *local page generation time*. Unlike previous classification proposals, TokenTLB is able to immediately detect shared-to-private transitions (as soon as the TLB obtains all N tokens), which allows us to efficiently detect private intervals of global page lives, averagely representing around 200,000 cycles per interval [18].

However, when applying coherence deactivation, we must take special care in the implications and penalty of page reclassification. Note that even under a coherent state (SW), classification could transition again to non-coherent during the same *local page generation time*, provided that a TLB recovers all tokens, becoming private anew (PW).

Specifically, when a reclassification from private or shared-read-only to shared-written occurs in a core's TLB, all the blocks (non-coherent copies) of that page must be evicted from all core's private cache (flushing-based recovery). To this end, a message is broadcast, and a page flush is performed through all system private caches for TLBs holding a valid page entry whenever a transition to shared-written occurs. Therefore, once the recovery mechanism is finished, the directory cache must have a coherent state according to the new page classification. This process must be atomic to avoid inconsistencies (accesses to that page are stalled until the recovery process is finished in all TLBs).

Page flushing occurs also whenever a TLB entry is either evicted or invalidated (i.e. it gives all of its tokens away) from the last private level TLB. The presence or absence of a TLB entry must signify the presence or absence of cache blocks for that page in the upper level cache structure in order to accurately classify data based in the presence of a translation entry in the TLB structure. Also, note that TLBs maintain information affecting coherence management, thus block presence must be prevented if their sharing information is lost. This will hardly affect performance since for a TLB to be evicted, it must not have been accessed for a while, and it is likely that blocks are neither present in the cache structure nor will be accessed in the near future.

Finally, when a reclassification to non-coherent occurs (i.e. from SW to PW) no actions are required. However, we may potentially have accessed blocks of that page as coherent, and allocated the corresponding block entry in the directory cache. When evicting the directory entry due to a conflict, if it produces an invalidation in a currently private cache entry, an unnecessary cache miss may occur afterwards. Notice how the status for a non-coherent block would be naturally restored by the recovery mechanism when transitioning to coherent again (after being accessed from another TLB). To prevent this to happen, if a block is found as non-coherent when evicting a directory entry (which sends an invalidation request to the cache pointed by it), an acknowledgment is sent to the directory but the block is allowed to remain in the cache as non-coherent.

Memory Parameters	
Processor frequency	2.8GHz
TLB hierarchy	Exclusive
Split instr & data L1 TLBs	8 sets, 4-way (32 entries)
L1 TLB hit time	1 cycle
Unified L2 TLB	128 sets, 4-way (512 entries)
L2 TLB hit time	2 cycle
Token acquisition timeout	1200 cycles
TPB	32 sets, 4-way
TPB hit time	1 cycle
Page size	4KB (64 blocks)
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
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

Table 2: System parameters for the baseline system.

5. SIMULATION ENVIRONMENT

We evaluate our proposal with full-system simulation using Virtutech Simics [29] along with the Wisconsin GEMS toolset [33], which enables detailed simulation of multiprocessor systems. The interconnection network has been modeled using the GARNET simulator [5]. We simulate a 16-tile CMP architecture implementing directory-based cache coherence and with the parameters shown in Table 2, which are considered as the *base* architecture for evaluation. L2 TLB miss latency considers four memory references to walk the page table, as in the 48-bit x86-64 virtual address space. The cache and TLB latencies and energy consumption have been calculated using the CACTI tool [49] assuming a 32nm process technology. Through experimentation we have observed that a token acquisition timeout of 1200 cycles (after which the TLB miss broadcast is resent) offers a good balance between performance and network traffic.

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-opt* (teapot), *Volrend* (head), and *Water-NSQ* (512 molecules, 4 time steps) are from the SPLASH-2 benchmark suite [51]. *Tomcatv* (256 points, 5 time steps) and *Unstructured* (Mesh.2K, 5 time steps) are two scientific benchmarks. *FaceRec* (script), and *SpeechRec* (script) belong to the ALPBenchs suite [25]. *Blackscholes* (simmedium), *Swaptions* (simmedium), and *x264* (simsmall) come from PARSEC [12]. Finally, *Apache* (1000 HTTP

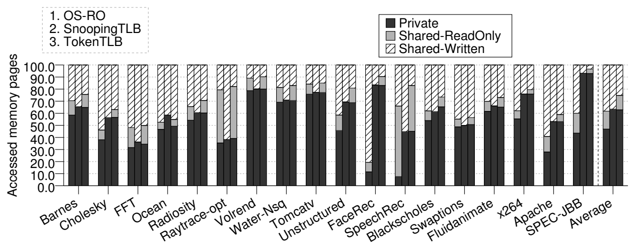


Figure 3: Private, Shared, and Written page proportion.

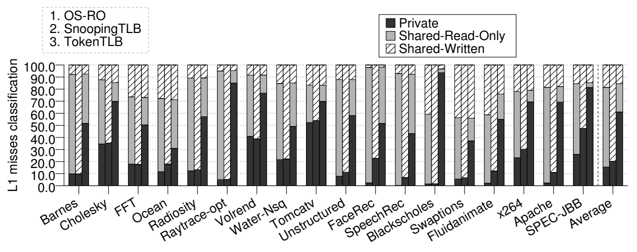


Figure 4: Data L1 Misses proportion classified as Private, Shared-Read-Only and Shared-Written.

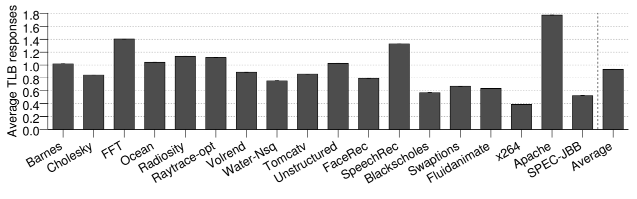


Figure 5: Proportion of TLB Responses issued after an L2 TLB miss.

transactions), and *SPEC-JBB* (1600 transactions) are two commercial workloads [7]. Raytrace-opt optimizes the original Raytrace application by removing a lock acquisition for a ray ID which is not currently used anymore. All the reported experimental results correspond to the parallel phase of benchmarks.

6. EVALUATION RESULTS

Results show firstly how TokenTLB classification behaves compared to previous classification approaches, including the sensitivity analysis of the Token Predictor Buffer for different sizes. Secondly, we introduce coherence deactivation in the analysis in order to compare the benefits obtained when applying the diverse classification mechanisms studied. Finally, a comparative study is made showing how the different classification mechanisms scale with the core count.

6.1 Fully-Adaptive Page Classification

This section demonstrates the classification accuracy and efficiency of TokenTLB compared to previous proposals, with and without applying TLB decay mechanism.

Private and Read-Only data. The percentage of private and shared (read-only/written) pages is a good general metric for measuring the goodness of a classification approach. Figure 3 shows how pages are classified as Private, Shared-ReadOnly or Shared-Written by different classification mechanisms. *OS-RO* is a non-adaptive OS-based classification mechanism with Read-only detection [15]. *SnoopingTLB* is an adaptive broadcast TLB-based classification approach [18], and *TokenTLB* is our fully-adaptive token-based TLB classification approach. As *SnoopingTLB* is not able to distinguish shared-read-only or shared-written pages, all shared pages fall under the same classification category. However, for the sake of clarity, in the graph it appears as *Shared-Written* in all *SnoopingTLB* configurations. We observe as, averagely, the sum of private and shared-read-only pages for *OS-RO* does not suffice to outmatch private pages for *SnoopingTLB*, which represents a 63.4% of all accessed pages, proving the relevance of an adaptive approach. However, in some cases, as *Raytrace-opt* or *SpeechRec*, a lot of potential classification precision is lost when write detection is not performed, as *OS-RO* overpasses *SnoopingTLB*.

However, this metric is unfair for adaptive classification mechanisms, where shared pages are frequently reclassified as private, since reclassification is not reflected in the figure. Specifically, this situation is favored by the fact that TokenTLB unlocks page access after the first TLB response, which accelerates TLB miss resolution, but with ongoing evictions it might end up in a shared access while tokens are still in-flight. Therefore, in the figure it appears as shared while it is naturally reclassified as private short after its first access. However, computing both private and shared-read-only pages for *TokenTLB*, it improves page classification to 74.9%.

The aim of all classification mechanisms is to precisely classify memory accesses. While an adaptive mechanism may be able to freely reclassify pages, blocks are never individually reclassified during a local block generation time. Therefore, the more the page classification is kept as private or read-only, the more L1 cache misses will end up being treated as such. Figure 4 shows L1 data cache misses classification, which will determine how accesses to data blocks will be treated. Even though classification of private pages in Figure 3 was close between *SnoopingTLB* and *TokenTLB*, L1 data misses considered as private is greatly increased using *TokenTLB*, since, unlike *SnoopingTLB*, page reclassification occurs in a natural way during a page generation time. Specifically, *TokenTLB* is able to classify 61.1% of L1 data cache misses as Private on average, 40.8% more than *SnoopingTLB*. Also, note as, contrary to *SnoopingTLB*, *TokenTLB* and *OS-RO* are capable of recognize read-only pages, representing the 24.4% of L1 cache misses for *TokenTLB*, thus greatly enhancing the classification accuracy.

Token TLB-to-TLB exchange. One key benefit of TokenTLB classification over previous proposals is how it handles TLB-to-TLB transfers, obtaining their benefits (page classification, usage prediction allowance, and translation acceleration), while limiting the required responses. As a result, TLB traffic is reduced, whereas system blockage waiting for collecting answers is avoided. Figure 5 represents how many average responses are sent after a TLB miss using *TokenTLB*. Take into account that broadcast TLB transfers for *SnoopingTLB* mechanisms require invariably $N-1$ (being N the number of cores in the CMP) responses after every TLB miss. On the contrary, *TokenTLB* requires just 0.93 responses per L2 TLB miss on average. The average falls below one due to the fact that, using TokenTLB, no TLB responses are sent nor expected when the tokens are held in the page table. In some cases, as *Apache* or *SpeechRec*, it goes beyond one response per L2 TLB miss. Note that TokenTLB allows more than one translation *owner* in the TLB structure, as evictions may be accepted by the first valid TLB in the eviction ring, therefore in those cases two

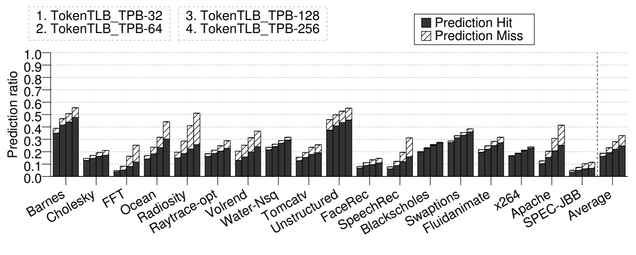


Figure 6: Success rate for TPB predictions.

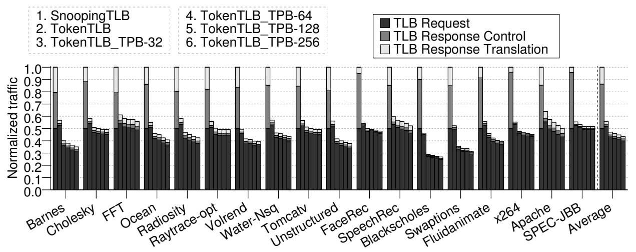


Figure 7: Relative TLB network traffic issued.

or more responses may be issued after the next TLB request.

Token prediction effectiveness. Hereby we briefly analyze the Token Predictor Buffer (TPB), which is conceived to avoid recurring to broadcast on TLB misses when there may be a known potential token owner. Therefore, it has an impact in the traffic and network consumption, but not in the execution time. TPB itself is just a small 4-way associative short-term buffer memory, and up to four different sizes have been evaluated in the study (32, 64, 128, and 256 entries).

Figure 6 shows the proportion of successful and failed predictions (i.e. number of remote TLBs currently holding tokens or not after being requested through an unicast prediction) with respect to total L2 TLB misses. It demonstrates that increasing TPB size affects positively to the accuracy of its predictions. Specifically, a 256-entry TPB avoids the broadcasts by 24.7% out of a total of nearly 33% of TLB miss prediction tryouts on average. In some cases, as *Barnes* or *Unstructured*, around 45% of broadcast TLB misses are prevented by using the TPB.

Figure 7 details the TLB traffic compared to base *SnoopingTLB*. It can be observed as *TokenTLB* slightly increases TLB request traffic compared to *SnoopingTLB* due to the non-silent evictions performed. However it is greatly offset with the reduction in TLB response and translation traffic, as can be deduced from Figure 5, reducing overall TLB traffic by 44%. Additionally, TPB usage further decreases TLB traffic. Specifically, the greater TPB considered (256 entries) reduces TLB request traffic to a greater extent, as it entails making more predictions, reducing solely the TLB request traffic by nearly 20% regarding *TokenTLB*, and total TLB traffic up to 56.3%.

As a conclusion, *TokenTLB* detects written condition at page granularity, while achieving a private detection similar to previous adaptive mechanisms. But seemingly loses accuracy, specially when using TLB decay mechanism. However, the strength of *TokenTLB* is the benefit obtained from immediate shared-to-private reclassification (full adaptivity), dramatically increasing the number of blocks classified as Private or Read-Only when missing over any previous classification approach. Finally, TLB response and translation messages are greatly bounded using our approach. TPB inclusion reduces the total TLB traffic by more than half using TPB, thus increasing the scalability of the system.

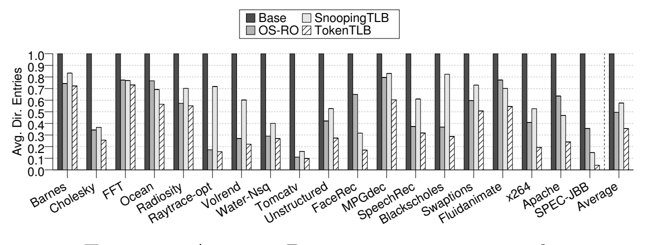


Figure 8: Average Directory usage per cycle

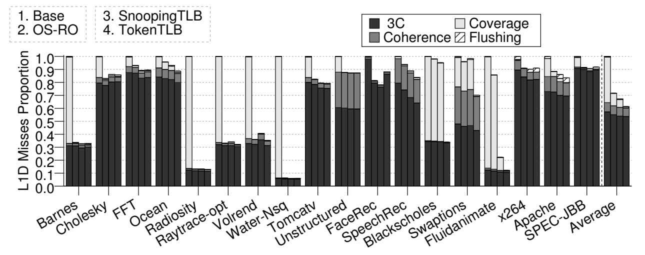


Figure 9: Data L1 Misses classified by its cause.

6.2 Coherence Deactivation

Coherence deactivation mechanism greatly benefits from the accuracy of a classification approach. It can easily benefit from both a private/shared classification dichotomy and a private/read-only/shared-written classification scheme. This section shows the benefits and overheads of applying TokenTLB to coherence deactivation compared to previous classification approaches.

Coherence maintenance is deactivated when a block access is considered non-coherent by the classification mechanism. For *SnoopingTLB*, which only characterizes memory accesses into private/shared scheme, all shared pages are coherent. Differently, both *OS-RO* and *TokenTLB* also distinguish written pages, so only shared-written pages are considered coherent. Moreover, *SnoopingTLB* and *TokenTLB* are adaptive mechanisms, transitioning back and forth from private to shared. Figure 8 shows the average number of directory entries required per cycle for the different approaches studied, normalized to *Base*, which is a system with the same configuration but without coherence deactivation. Constraining the directory usage is the ultimate goal for coherence deactivation and is strongly dependent on the accuracy of the classification mechanism. We observe as *OS-RO* improves directory usage by 52.8%, reducing it over *SnoopingTLB* due to its read-only detection capability. Finally, *TokenTLB* greatly improves directory usage, requiring only 34.1% of the directory entries per cycle compared to *Base*.

Classifying more pages as non-coherent and improving directory usage also reduces L1 cache misses to a greater extent, especially Coverage misses. Figure 9 classifies data misses based on its cause, and thus, shows how all proposals lessen Coverage misses on the L1 due to apply coherence deactivation. However, *TokenTLB* is actually capable of eliminating nearly all Coverage misses without applying a TLB decay mechanism. On the contrary, on average, Coverage misses with *OS-RO* still represents 13.4% of the L1 cache misses. Finally, *SnoopingTLB* is somewhere in between, as it still suffers 8.7% of Coverage misses.

Reducing L1 data cache misses leads to a network usage

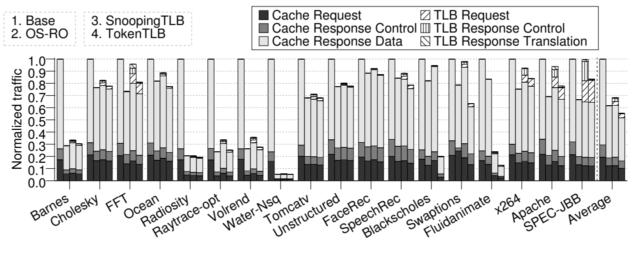


Figure 10: Network flits injected, classified into cache- or TLB-traffic.

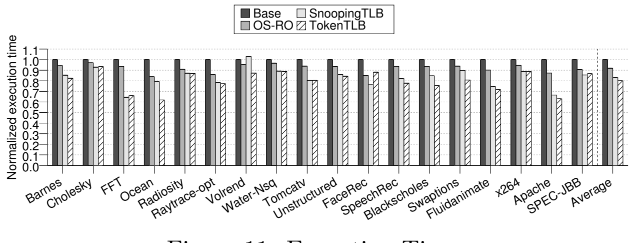


Figure 11: Execution Time.

reduction. Figure 10 shows the total flits injected into the network, classified into cache or TLB traffic. It can be observed as *TokenTLB* prevents nearly half the total network usage on average due to both the fully-adaptive classification and the write detection performed. However, *SnoopingTLB* only reduces it by 31%. Comparatively, as *OS-RO* does not require additional TLB traffic and has write detection capability, it reduces overall traffic over *SnoopingTLB* even though it performs a non-adaptive classification.

Reducing cache misses and accelerating page translation through TLB-to-TLB transfers has a direct positive impact on execution time, which is improved by 20% using *TokenTLB* compared to *Base*, as shown in Figure 11. Also, execution time is reduced by 3% compared to *SnoopingTLB*, as *TokenTLB* unblocks page access earlier after TLB misses, and the L1 cache misses are reduced to a greater extent. As *OS-RO* does not benefit from fast TLB miss resolution through TLB-to-TLB transfers, its gaining is provided solely by coherence deactivation, reducing execution time by 8.8%.

Our proposal also entails a reduction in the cache hierarchy dynamic energy consumption, as shown in Figure 12. *TokenTLB* reduces overall consumption by 21.9% compared to the baseline, particularly L1 cache energy consumption, since cache pressure is reduced through coherence deactivation. Network consumption is also significantly decreased, although proportionally to the total consumption its impact is diluted. Comparatively, *TokenTLB* reduces the dynamic consumption by 5.7% with respect to *SnoopingTLB*, and 9% with respect to *OS-RO*.

Conclusively, applying TokenTLB to coherence deactivation improves the system scalability. It greatly reduces directory entries due to both its full-adaptive classification and the write detection capability, including accesses to Read-Only pages among non-coherent data. As a consequence, TokenTLB allows smaller and more efficient directory structures, while it effectively reduces L1 cache misses, network traffic, dynamic consumption, and improves system execution time over any previous classification mechanism.

6.3 Comparative Analysis

This section offers a general comparative analysis of how the different classification approaches perform when applied to the coherence deactivation mechanism, including TLB Decay with two different timeout values: 250,000 and 50,000 cycles. Note that TLB Decay is a key mechanism to uncouple classification from TLB size and that it cannot be applied on OS-RO.

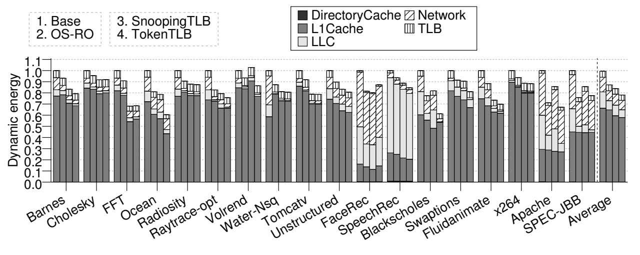


Figure 12: Dynamic energy consumption normalized to the base system.

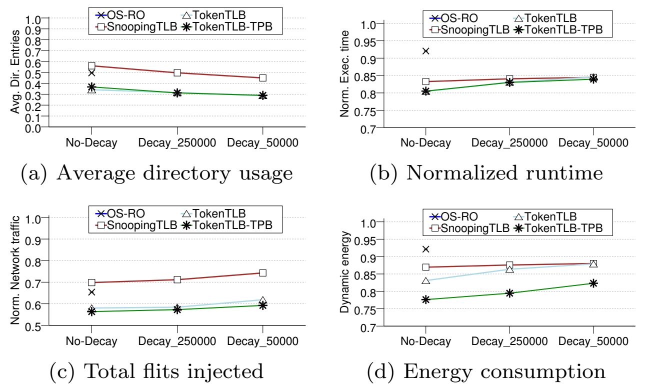
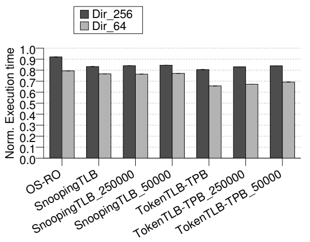

(a) Average directory usage (b) Normalized runtime (c) Total flits injected (d) Energy consumption

Figure 13: General overview of different classification mechanisms applied to coherence deactivation

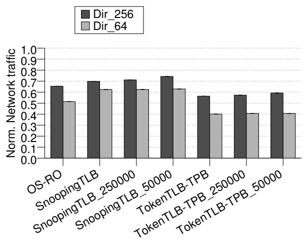
Figure 13 represents a summary of how the classification approaches studied in the paper affect to the global system performance applied to coherence deactivation. All results are normalized to the baseline system without coherence deactivation but with the same system configuration. TPB optimization is included in the study for *TokenTLB*.

First, Figure 13a shows the average directory entries required per cycle for the different classification approaches studied. As previously observed, *OS-RO* prevents more directory entries per cycle compared to *SnoopingTLB* due to its write detection capability. However, it still performs a non-adaptive classification, and with TLB decay appliance, *SnoopingTLB* reduces directory entries per cycle by up to 55% on average. Finally, *TokenTLB* reduces directory usage by up to 71% using the lowest decay timeout considered.

Next, Figure 13b evidences how both *SnoopingTLB* and *TokenTLB* improve the execution time over the baseline and *OS-RO* to a greater extent, specially prior to TLB decay appliance, where *TokenTLB* reduces the execution time nearly by 20% on average over the baseline. However, applying TLB decay entails a small shrinkage in the system performance as it slightly increases the misses in the L1 cache structure due to decay forced flushing. Nonetheless, even using the lowest TLB decay timeout considered, it still reduces execution time nearly by 16% for both *SnoopingTLB* and *TokenTLB* over the baseline. TPB usage slightly lessens the performance loss of TLB decay usage. In this case, pages are invalidated to a lesser extent as TLBs are inquired less often using unicast messages after TLB misses.

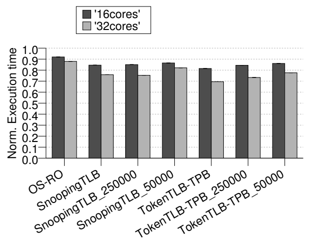


(a) Normalized runtime

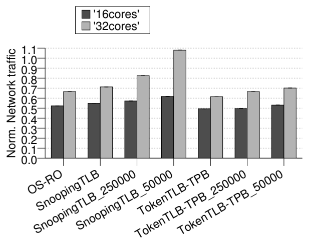


(b) Total flits injected

Figure 14: Execution time and network usage for a 0.25x directory



(a) Normalized runtime



(b) Total flits injected

Figure 15: Scalability analysis of classification approaches when increasing core count.

Figure 13c illustrates how *TokenTLB* reduces traffic with respect to *SnoopingTLB* and *OS-RO*. Moreover, in this case, TLB decay slightly increases traffic as it entails more misses in the L1 cache. Again, TPB prevents and reduces traffic both when used with and without TLB decay.

Lastly, Figure 13d shows how the TPB reduces the total cache hierarchy dynamic energy consumption as it avoids many accesses to the TLB hierarchy and the issue of TLB-to-TLB broadcasts. Specifically, TPB reduces consumption by 5.4% compared to base TokenTLB and 9.4% compared to SnoopingTLB. TLB Decay usage entails a small increase in the energy consumption.

The main goal of coherence deactivation is to reduce directory usage and, thus, favor the possibility of smaller and more scalable directory designs. Furthermore, Figure 9 evidenced how the studied classification approaches reduce Coverage misses to a great extent for the considered directory size. Nevertheless, when reducing the directory size, more accurate classification approaches should provide better performance results when applied to a coherence deactivation approach. Figure 14 shows average numbers for execution time and network usage for a 256-entry directory and a 64-entry directory normalized to the baseline, i.e. a system without coherence deactivation and the same directory size and global configuration as the considered approach. Observe that, as different bars are normalized to different baselines, they cannot be directly compared between them. The figure reveals how *TokenTLB-TPB* prevents the execution time penalization when the directory size is reduced (Figure 14a). Specifically, it reduces execution time by 34.2% for a 64-entry directory, nearly 11% more reduction compared to *SnoopingTLB*.

Similarly, reducing the directory increases the traffic as it entails more Coverage misses in the L1. Again, *TokenTLB-TPB* restricts the traffic to a greater extent (Figure 14b) compared to other classification approaches. Particularly, network traffic is reduced on average by up to 59.9% over the baseline. Moreover, *OS-RO* reduces it by 48.6%, and *SnoopingTLB* by just 37.56% due to the TLB traffic overhead.

6.4 System Scalability

This section shows how the different classification approaches scale when applied to deactivate coherence. Due to the slowness of the simulation tools, this study is only performed using SPLASH 2 benchmarks and scientific applications.

Figure 15a shows how *TokenTLB-TPB* scales better compared to *SnoopingTLB* and *OS-RO*. Specifically, *TokenTLB-TPB* reduces the execution time by 30.5% on a 32-cores CMP, while *SnoopingTLB* reduces it by 25.15% over the baseline. This difference evidences how performing a more accurate classification entails better directory usage and ultimately better system performance.

Note how among the aims of *TokenTLB* is to achieve a more scalable system in terms of network usage compared to previous classification approaches. In this sense, Figure 15b shows how *TokenTLB-TPB* reduces traffic over the baseline to a greater extent, restraining even the traffic that entails the TLB decay usage. Expressly, *TokenTLB-TPB* maintains the traffic reduction up to 38.6% for a 32-cores system, while *OS-RO* in only able to reduce the traffic by 32.5%. However, when applying TLB decay with a 50,000 timeout value the impact of the classification mechanism increases. *TokenTLB-TPB* still reduces network usage by 29.9% over the baseline, while, on the contrary, *SnoopingTLB* increases the traffic up to nearly 8%.

7. CONCLUSIONS

This paper proposes *TokenTLB*, a novel TLB classification mechanism based on counting and exchanging tokens through TLB-to-TLB requests, where only TLBs *owning* the translation are allowed to answer. When tokens are used for classification instead of using them for coherency, the need for persistent requests is avoided since an owner token is not required. Moreover, token counting is highly efficient for performing an adaptive classification into a private-shared scheme, naturally detecting private phases of pages during its page generation time. Finally, classification dichotomy is extended by allowing write detection for the first time for an adaptive classification mechanism. By predicting token holders and sending unicast messages, the scalability of the system is improved. The proposed TokenTLB presents all the desirable characteristics of a classification scheme: the classification is known before the cache access (*a-priori*), detects *read-only* accesses, is *adaptive*, and is *accurate*.

When applied to coherence deactivation, TokenTLB reduces the average directory entries stored to merely a 28.8%, reducing L1 cache misses, network traffic and, consequently, overall system consumption by 27.3% over baseline.

8. ACKNOWLEDGMENTS

This work has been jointly supported by the MINECO and European Commission (FEDER funds) under the project TIN2015-66972-C5-1-R/3-R and the *Fundación Séneca, Agencia de Ciencia y Tecnología de la Región de Murcia* under the project *Jóvenes Líderes en Investigación* 18956/JLI/13.

9. REFERENCES

[1] Advanced Micro Devices. http://www.amd.com. [Online; accessed Jan-2016].

[2] ARM holdings plc. http://www.arm.com. [Online; accessed Jan-2016].

[3] Intel Corporation. http://www.intel.com. [Online; accessed Jan-2016].

[4] M. E. Acacio, J. González, J. M. García, and J. Duato. Owner prediction for accelerating cache-to-cache transfer misses in cc-NUMA multiprocessors. In *ACM/IEEE Conf. on Supercomputing (SC)*, pages 1–12, Nov. 2002.

[5] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A detailed on-chip network model inside a full-system simulator. In *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 33–42, Apr. 2009.

[6] 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)*, pages 67–78, Feb. 2009.

[7] A. R. Alameldeen, C. J. Mauer, M. Xu, P. J. Harper, M. M. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Evaluating non-deterministic multi-threaded commercial workloads. In *5th Workshop On Computer Architecture Evaluation using Commercial Workloads (CAECW)*, pages 30–38, Feb. 2002.

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

[9] 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)*, pages 307–318, June 2011.

[10] 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)*, pages 62–73, Feb. 2011.

[11] 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)*, pages 359–370, Mar. 2010.

[12] 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)*, pages 72–81, Oct. 2008.

[13] 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*, 26(1):70–79, Jan. 2006.

[14] 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)*, pages 93–103, June 2011.

[15] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato. Increasing the effectiveness of directory caches by avoiding the tracking of non-coherent memory blocks. *IEEE Transactions on Computers (TC)*, 62(3):482–495, Mar. 2013.

[16] 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)*, pages 317–330, Oct. 2015.

[17] 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)*, pages 177–188, Feb. 2014.

[18] 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 (TPDS)*, 27(3):748–761, Mar. 2016.

[19] 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)*, pages 169–180, Feb. 2011.

[20] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *36th Int'l Symp. on Computer Architecture (ISCA)*, pages 184–195, June 2009.

[21] 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)*, pages 45–55, Oct. 2011.

[22] 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)*, pages 240–251, June 2001.

[23] D. Kim, J. Ahn, 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)*, pages 111–122, Sept. 2010.

[24] K. Koukos, A. Ros, E. Hagersten, and S. Kaxiras. Building heterogeneous unified virtual memories (uvms) without the overhead. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):1:1–1:22, Mar. 2016.

[25] 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)*, pages 34–45, Oct. 2005.

[26] 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)*, pages 501–512, Sept. 2010.

[27] Y. Li, R. Melhem, and A. K. Jones. PS-TLB: Leveraging page classification information for fast, scalable and efficient translation for future cmps. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):28:1–25:21, Jan. 2013.

[28] 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), pages 231–240, Sept. 2012.

[29] 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*, 35(2):50–58, Feb. 2002.

[30] M. M. Martin. *Token Coherence*. PhD thesis, University of Wisconsin-Madison, Dec. 2003.

[31] M. M. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *30th Int'l Symp. on Computer Architecture (ISCA)*, pages 206–217, June 2003.

[32] 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)*, pages 182–193, June 2003.

[33] 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*, 33(4):92–99, Sept. 2005.

[34] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. Martin, and D. A. Wood. Improving multiple-CMP systems using token coherence. In *11th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 328–339, Feb. 2005.

[35] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos. Prediction-based superpage-friendly tlb designs. In *21th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 210–222, Feb. 2015.

[36] 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)*, pages 465–476, Sept. 2010.

[37] A. Raghavan, C. Blundell, and M. M. Martin. Token tenure: PATCHing token counting using directory-based cache coherence. In *41th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 47–58, Nov. 2008.

[38] 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)*, pages 1–12, Feb. 2010.

[39] A. Ros, M. E. Acacio, and J. M. García. DiCo-CMP: Efficient cache coherency in tiled CMP architectures. In *22nd Int'l Parallel and Distributed Processing Symp. (IPDPS)*, pages 1–11, Apr. 2008.

[40] A. Ros, M. E. Acacio, and J. M. García. Dealing with traffic-area trade-off in direct coherence protocols for many-core cmps. In *8th Int'l Conf. on Advanced Parallel Processing Technologies (APPT)*, pages 11–27, Aug. 2009.

[41] 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 Conf. on High Performance Computing (HiPC)*, pages 1–10, Dec. 2010.

[42] A. Ros, B. Cuesta, M. E. Gómez, A. Robles, and J. Duato. Temporal-aware mechanism to detect private data in chip multiprocessors. In *42nd Int'l Conf. on Parallel Processing (ICPP)*, pages 562–571, Oct. 2013.

[43] A. Ros, M. Davari, and S. Kaxiras. Hierarchical private/shared classification: the key to simple and efficient coherence for clustered cache hierarchies. In *21th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 186–197, Feb. 2015.

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

[45] A. Ros and A. Jimborean. A hybrid static-dynamic classification for dual-consistency cache coherence. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, PP(99), Feb. 2016.

[46] A. Ros and S. Kaxiras. Complexity-effective multicore coherence. In *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 241–252, Sept. 2012.

[47] D. Sanchez and C. Kozyrakis. SCD: A scalable coherence directory with flexible sharer set encoding. In *18th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 129–140, Feb. 2012.

[48] 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)*, pages 313–324, Dec. 2010.

[49] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20, HP Labs, Apr. 2008.

[50] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal. DiDi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 340–349, Oct. 2011.

[51] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Int'l Symp. on Computer Architecture (ISCA)*, pages 24–36, June 1995.

[52] J. Zebchuk, B. Falsafi, and A. Moshovos. Multi-grain coherence directories. In *46th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 359–370, Dec. 2013.

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