Classifying Data Blocks at Subpage Granularity With an On-Chip Page Table to Improve Coherence in Tiled CMPs

Mohammadreza Soltaniyeh, Ismail Kadayif, *Member, IEEE*, and Ozcan Ozturk, *Member, IEEE*

***Abstract*—As shown in some prior studies, a significant percentage of data blocks accessed in parallel codes are private, and not keeping track of those blocks can improve the effectiveness of directory structures in Chip multiprocessors (CMPs). In this paper, we have two major contributions. First, we showed that compared to the classification of cache blocks at page granularity, data block classification (DBC) at subpage level helps to detect considerably more private data blocks. Based on this idea, we propose two different approaches for enhancing the effectiveness of directory caches in tiled CMPs. In the first approach, which is called quasi-dynamic subpage level DBC (QDBC), a data block is assumed to be private from the beginning of the program execution and stays private as long as the corresponding subpage is accessed by only one core. Our second approach, which is called dynamic subpage level DBC, turns a data block into private again after all blocks within the corresponding subpage are evicted from private cache hierarchy. Memory block classification at subpage level, however, may increase the frequency of the operating system involvement in updating the maintenance bits in page table entries. To overcome this, we propose, as a second contribution, a distributed table called as on-chip page table (o-CPT), which stores recently accessed page translations in the system. Our simulation results show that, compared to page level data classification, QDBC and DBC approaches relying on the o-CPT can detect significantly more private data blocks and considerably improve system performance.**

***Index Terms*—Address translation, cache coherence, chip multiprocessor (CMP), directory cache, page table, translation look-aside buffer (TLB).**

I. Introduction

CHIP multiprocessors (CMPs) require efficient and scalable cache coherence protocols to extract maximum performance from shared memory applications and fast virtual-to-physical address translation to efficiently manage the virtual memory. Compared to snooping-based protocols, directory-based cache coherence protocols are a better choice for CMPs with many cores since they avoid broadcasting, reducing the message traffic on on-chip networks. In these protocols, the memory blocks in the last level private caches of the cores are kept coherent by monitoring the status of data blocks in centralized directory structures.

While directory-based cache coherence protocols are the state-of-the-art approaches in many-core CMPs, area overhead, high energy consumption, and high associativity requirement of the directory structures may not scale well with increasingly higher number of cores in a chip. Different directory organizations have been suggested to enable lower overhead and higher scalability in coherence protocols relying on directories [22], [28]. One way of reducing storage requirements for directory caches is to keep track of only cached memory blocks [9], [25], instead of monitoring all the memory blocks in the system. While this enables directory information to be kept in small directory caches, it can cause frequent directory cache evictions due to the lack of a full directory, which in turn leads to invalidations of all the cached copies of the associated blocks in private caches.

In modern CMPs, directory caches are implemented as distributed structures across cores in mainly two different ways. One way is to implement them as separate structures, which are called as duplicate-tag directories [4], [21]. They are simple to implement, but they require associativity proportional to the number of cores, which make them energy inefficient for systems with high core counts. By lowering associativity and/or the number of entries, duplicate tag directories can be turned into more flexible structures called as sparse directories [16]. Because of their limited sizes and/or associativities, these directory structures can introduce huge directory miss rates and consequently cause frequent data block invalidations in private caches, deteriorating the system performance significantly [18]. The second way is to take advantage of the shared last level cache (LLC) and maintain the directory information as part of the LLC's entries. This kind of directory organization is referred to as In-Cache directory [30]. Each directory entry in this design includes a set of sharers to keep track of cache blocks in the private caches, but requires no tag. In-Cache directory organization enforces the inclusion property between private and shared caches, i.e., the shared cache must allocate an entry for each block in private caches, which reduces the effective on-chip storage capacity. This is why we

Manuscript received October 31, 2016; revised February 25, 2017 and June 1, 2017; accepted June 9, 2017. Date of publication July 19, 2017; date of current version March 29, 2018. This work was supported by the Scientific and Technological Research Council of Turkey under Grant 113E258. This paper was recommended by Associate Editor C. Kirsch. *(Corresponding author: Ismail Kadayif.)*

M. Soltaniyeh is with the Department of Computer Science, Rutgers University, New Brunswick, NJ 08854 USA.

I. Kadayif is with the Department of Computer Engineering, Canakkale Onsekiz Mart University, 17100 Çanakkale, Turkey (e-mail: kadayif@comu.edu.tr).

O. Ozturk is with the Department of Computer Engineering, Bilkent University, 06100 Ankara, Turkey.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2017.2729280

consider sparse directories in this paper and try to increase the effectiveness of directory caches with available scarce space and a restricted associativity.

Several prior studies suggest page granularity data classification mechanisms to decrease coherence management overhead in directories [3], [10]. However, we have observed that performing data classification at finer granularity than page granularity can bring additional benefits. Moreover, we also believe that the benefits obtained by page level data classification diminishes significantly especially when working with large page sizes.

In addition to an effective management of directory structures, fast virtual to physical address translation is also important for the system performance in CMPs. This is crucial in our study since our data block classification (DBC) at subpage level can increase the frequency of operating system (OS) involvement in classifying data blocks and applying the coherency recovery mechanism at runtime. To enable fast virtual to physical address translation, recent studies have focused on designing L2 (level 2) translation look-aside buffers (TLBs) for CMPs [6], [24], [32]. In this paper, we tailor L2 TLBs in CMPs to facilitate our DBC and coherency recovery mechanism. We refer to such an L2 TLB as on-chip page table (o-CPT) to differentiate it form regular L2 TLBs in CMPs. Based on these discussions, this paper makes the following contributions.

1) First, we propose a DBC mechanism referred to as quasi-dynamic subpage level DBC (QDBC), which works at subpage level and helps to detect considerably more private data blocks. Consequently, it reduces the percentage of blocks required to be tracked in the directory significantly compared to similar page level classification approaches. This, in turn, enables smaller directory caches with a lower associativity to be used in CMPs without hurting performance, thereby helping the directory structure to scale gracefully with the increasing number of cores.
2) Second, in the QDBC approach, a data block is assumed to be private from the beginning of program execution and stays private as long as the corresponding subpage is accessed by only one core. When another core tries to access any block within that subpage, this data block becomes shared and remains as shared until the program terminates. To take advantage of dynamic behavior of data blocks and hence to detect more data blocks at runtime, we propose our second approach, which is called as dynamic subpage level DBC. The way private data blocks are turned into shared with the DBC approach is similar to QDBC. However, unlike QDBC, in this approach a data block becomes private again after all blocks within the corresponding subpage are evicted from all the private caches of cores.
3) Third, we propose a small distributed table referred to as o-CPT, which stores the page table entries for recently accessed pages in the system. This can be implemented by as a portion of the directory controller by customizing the L2 TLB. Upon a TLB miss, the OS gets involved in address translation only when the requested translation is not found in the o-CPT. It also helps to lower the performance degradation due to the increase in the frequency of the OS involvement in our subpage granularity block classification. This table is vital especially for DBC in dynamic DBC and restoring the coherence status of transitioned data blocks.

A preliminary version of this paper appears in our previous work [31]. Here, we extend this paper with a new DBC approach, which is quite more effective than the QDBC approach. More specifically, we show why monitoring data block evictions from private L1 caches is so important in successful DBC by comparing the effectiveness of the DBC and QDBC approaches. We also explain the hardware support requirements for the DBC approach as well as the required modifications to the coherence recovery mechanism in detail. Finally, we conduct more experiments with varying subpage sizes to show that, unlike QDBC, DBC can benefit from small subpages.

The rest of this paper is organized as follows. Section II discusses the background, related work, and motivation. We present our approaches in Section III. We provide the details of the evaluation methodology in Section IV. We give the performance results in Section V. Finally, Section VI concludes this paper.

II. Related Work and Motivation

Directory-based cache coherence protocols [1], [16], [26] are the common approach for managing the coherence in the systems with many cores in a single chip because of their scalability in power consumption and area compared to traditional broadcast-based protocols. However, the latency and power requirements of today's many-core architectures with their large LLCs brought new challenges.

A directory cache should provide an efficient way to keep the copies of data blocks stored in different private caches coherent since its structure can have momentous influence on overall system performance [16]. Therefore, there are numerous studies aiming at improving the performance of directory caches. It is common to cache a subset of directory entries due to high latency and power overheads with directory accesses. Ros *et al.* [29] proposed a direct cache coherence protocol suited to many-core tiled CMPs, which stores up-to-date sharing information about cache blocks into the cache that must provide the block on a miss, i.e., the owner cache. Therefore, cache miss services are handled more quickly by sending the requests directly to the owner cache. This approach is orthogonal to our proposals and they can be applied together. Fensch and Cintra [15] proposed a coherence protocol without any hardware support, in which data blocks are mapped to processors' caches at the page granularity under OS control. In this approach, data coherence is maintained by not allowing multiple writable copies of pages on private caches. While this technique uses page level data mapping, our proposal makes use of subpage level data classification to detect considerably more private data blocks.

A common scheme for organizing directories in CMPs is duplicate-tag-based directories [4], [27]. Compared to other

directory structures, these kinds of directory caches are more flexible as they do not force any inclusion among the cache levels. However, directories based on duplicate-tag come with overheads. Storage cost for duplicate tags, and more notably, high associativity requirement that grows with the number of cores in the system, are two main overheads. For instance, in a many-core processor with N cores, each of which has a K-way-set associative last level private cache, the directory cache must be NxK-way associative to hold all private cache tags (to avoid any invalidation). Therefore, this approach suffers from high power consumption and high design complexity due to high associative caches.

Like our study, there exist some prior studies in the literature to classify data blocks as private or shared for different purposes in CMPs, such as reducing coherence overhead or the access latency to distributed caches. Hardavellas *et al.* [17] and Li *et al.* [23] tried to categorize data blocks and keep private data blocks in the nonuniform distributed shared cache [nonuniform cache access (NUCA)] slice of the requesting core, where the access latency depends on the physical distance between the core demanding data and the L2 cache slice storing the data. The primary aim of these two studies is to reduce NUCA access latency by employing intelligent placement, migration, and replication mechanisms. On the other hand, in this paper we just focus on bypassing coherence protocol for private data blocks, considering neither block placement nor block replication. Moreover, the private data detection mechanisms in this paper are quite different from ones used in these studies. While in the study [17] cache access patterns are classified via the OS, Li *et al.* [23] try to detect private data offline with compiler assistance. As mentioned in some previous studies [3], [10], we believe that compared to offline, we can detect more private data blocks at runtime. In our study, we employ our own runtime data classification techniques to increase the private data detection rate. To categorize data blocks as accurately as possible, our dynamic data classification techniques work at subpage granularity and the technique used in DBC is able to take advantage of dynamic behavior of data blocks and reclassify a shared data block private again during runtime.

The prior research, which was done by Esteve *et al.* [13], proposes a runtime classification mechanism to take both thread migration and private data within application phases into consideration. In this paper, they make use of TLB lookups to dynamically classify data. More specifically, upon a TLB miss the core broadcasts a specific message to the other cores' TLBs to retrieve information about status of the corresponding page. According to their data classification approach, a page is considered to be private if its page translation resides in a single TLB. If the translation exists in two or more cores' TLBs, the page is assumed to be shared. While our data classification approach in DBC and the classification approach proposed in the work [13] both are dynamic, i.e., with both of them a shared data block can be transitioned into private again during program execution, there is a stark difference between them. Their classification approach count a page as private if its translation resides in a single core's TLB. On the other hand, our classification mechanism in DBC regards a subpage as private if a single core's L1 cache keeps the data blocks within this particular subpage. The accuracy of their data classification approach depends on how accurately TLB entry evictions are predicted. They propose a TLB decay technique based on 2-bit saturated counters in TLB entries to make accurate predictions, which introduces area overheads. Moreover, to keep the information about the private pages in all the TLBs coherent, they introduce a kind of TLB coherence protocol, which not only requires additional on-chip area but also increases on-chip network message traffic. TLB-to-TLB transfers in their study lead to replicated responses from every cores after a TLB miss, which can increase on-chip network bandwidth demand. Esteve *et al.* [14] proposed TokenTLB approach to reduce on-chip traffic caused by TLB-to-TLB transfers in resolving address translation misses.

Davari *et al.* [11] proposed a data classification technique at cache line granularity based on generations of cache lines. According to their technique, a generation for a cache line starts when it is brought into an L1 cache. They use the cache decay mechanism [19] to predict the termination of a generation. A cache line is classified as private if it has only one generation in the L1 cache hierarchy at that moment. In another study, Davari *et al.* [12] proposed a directory scheme called DIR_1-SISD, which employs self-invalidation and self downgrade as directory policy for shared data similar to the study [11]. This directory scheme relies on data-race-free (DRF) programming paradigm and can tolerate multiple-readers and multiple-writers to co-exist at the same time without broadcast messages. While their proposal brings area optimization and complexity reduction in directories, it works for only applications where the DRF semantics are enforced, that is, during any parallel phase of the applications, different cores can only modify different bytes/words in a cache line.

To our best knowledge, the closest study to ours was presented by Cuesta *et al.* [10]. However, there are at least three main differences between this paper and their study. First, the data classification granularities are different. Their study works on page level granularity to support private caches in systems like Magny-Cours [9], where there exist multiple dies and each die has its own processing cores and a die-wide large private cache. On the other hand, our proposal is more suitable for maintaining coherence across L1 private caches in single-die CMPs. Second, since our data classification technique works at subpage level, it can increase frequency of OS involvement in coherence maintenance. This is why we employ the o-CPT for minimizing the OS involvement at runtime in order to accelerate coherence maintenance. Third, unlike their data classification approach, our classification technique used in DBC takes advantage of data block status at runtime and considers a data block as private again after all blocks within the corresponding subpage are evicted from all the private caches of cores, which enables significantly more data blocks to be classified as private.

We believe that, by inspecting private data at finer granularity than page granularity, chances of finding private data and further improvements will be considerably higher. Fig. 1 shows

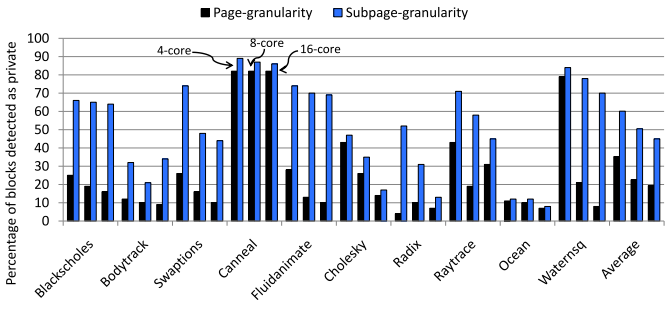

Fig. 1. Percentage of data blocks detected as private at page granularity versus subpage granularity (with 4 subpages per page) for various number of cores.

the amount of private accesses detected at subpage granularity (4 subpages per page) compared to page granularity for ten different multithreaded applications in systems with a various number of cores. In this figure, there are three groups of bars for each benchmark and they correspond to 4, 8, and 16 cores, respectively. While the first bar in each group shows the amount of blocks detected as private at page granularity, the second one indicates detected private blocks at subpage granularity. The reason for such a big difference in the amount of detected private blocks is that the existence of a single shared block within a page is enough to change the status of all the blocks within that page from private to shared. Thus, if we divide a page into subpages, it is less likely that shared blocks can adversely affect private blocks' status at subpage level.

While the page size is selected as 8 KB in our example, we believe that the difference would be more dramatic for those architectures that use larger page sizes for performance reasons. According to Fig. 1, the chances of detecting private accesses at subpage granularity is about two times more than doing so at page granularity. We will discuss later in this paper how we can perform the detection at the subpage level with the assistance of page tables and TLB entries.

Recently, different techniques have been suggested to improve TLB performance in CMPs. The technique presented in the study [32] tries to increase sharing capabilities between the cores by enabling cooperation between the private TLBs. More specifically, it provides capacity sharing among private TLBs by storing victim translations from other TLBs, which allows some address translations to be found in private TLBs, preventing some of relatively costly OS page table accesses. However, this technique can introduce both design complexity and higher network cost. Consequently, some studies [5], [6] suggest a shared last level TLB to improve sharing capacity among the cores in the system.

In this paper, we employ a distributed o-CPT, which stores the page table entries for recently accessed pages in the system. To differentiate it from regular shared L2 TLBs and emphasize that its entries need some extra fields required for maintaining shared/private status of data blocks, we name it as o-CPT. This o-CPT serves two purposes in the system. First, it serves as a shared last level TLB like the study [6] to reduce system-wide address translation misses. Second, it can reduce frequency of the OS involvement in subpage level data classification and coherence management. The details on how the o-CPT works will be explained later.

III. Bypassing Coherence

In this section, we explain the details of our memory management scheme that employs a runtime subpage granularity private data detection motivated by the observations mentioned in Section II. The two main issues here are: 1) the mechanism for detecting private memory blocks at subpage granularity and 2) the approaches that enable us to exploit the results of the data classification to improve the performance.

A. Private Block Detection

A common approach to differentiate between private and shared data blocks is to utilize OS capabilities [3], [10], [17]. The prior work [10] extends TLB and page table entries with some additional fields to distinguish between private and shared pages. To do so, two new fields are introduced in TLB entries: while the *private* bit (*P*) indicates whether the page is private or shared, the *locked* bit (*L*) is employed to prevent race conditions when a private page becomes shared and, in turn, the coherence status of cache blocks that belong to this particular page are restored. To distinguish between private and shared pages, three new fields are also added to page table entries: *P* marks whether the page is private or shared, as in TLB entries; if *P* is set, the *keeper* field indicates the identity of the unique core storing the page table entry in its TLB; and *cached-in-TLB* bit (*C*) shows whether the keeper field is valid or not.

While we also try to detect private data blocks at runtime and bypass the coherence protocol for accesses to those private blocks, our intention is to detect private data at finer granularity since the effectiveness of coherence deactivation mainly depends on the amount of detected private data. To accomplish this, we use most significant bits of page offset for subpage ID and clone *V* (valid bit), *P*, *C*, *L* bits and keeper fields in TLB and page table entries so that each subpage has its own such fields, as depicted in Fig. 2(a). In this paper, we divide each page into a number of subpages. The size of the keeper field grows according to the number of cores in the system. In other words, the size of the keeper is $\log_2(N)$, where N is the number of cores in the system. The storage overhead in a page table entry for four subpages per page will be $4 \times (1+1+4) = 24$ bits. If we assume that our system has 16 cores with 48-bit virtual and 40-bit physical address spaces and 8 KB pages, the overhead will be around 37%.

Now, we list the three main operations that should be performed to properly update the fields discussed earlier and enable detection of private data at subpage level. To make it more clear, we also show the operations in Fig. 2(b) with different colors.

1) *First (Red):* When a page is loaded into main memory for the first time, the OS allocates a new page table entry with the virtual to physical address translation. Besides storing the virtual to physical address translation in the page table entry, all the subpages are considered to be private and thereby, the corresponding (*P*) bits are set.

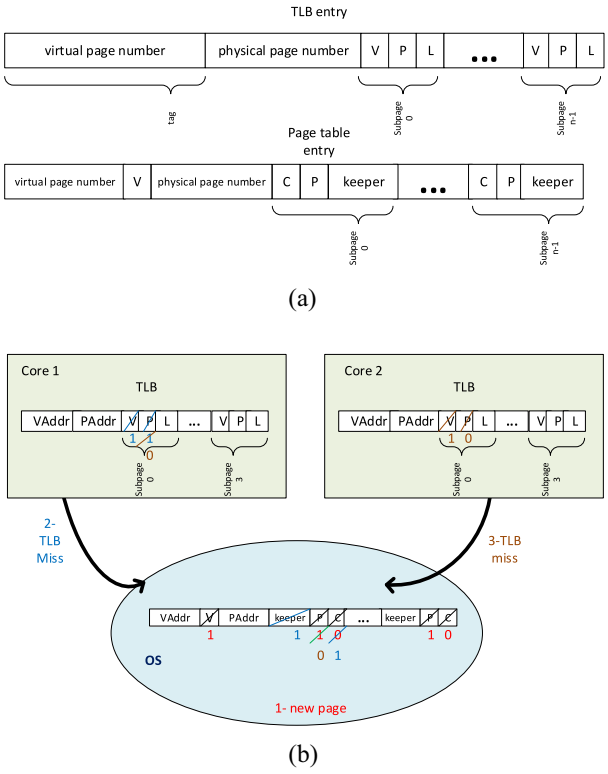

Fig. 2. Private block detection scheme. (a) TLB and page table entry formats. (b) Subpage granularity private data detection mechanism.

All subpages' (*C*) bits are also cleared, showing that no entries have been cached in any TLB yet.

2) *Second (Blue):* Core 1 faces a miss in its TLB for an address translation or there is a hit in the TLB but the (*V*) bit of the subpage which was tried to be accessed is cleared (which means we do not have information about the status of the subpage to be accessed). In either case, core 1 will inquire the OS page table for the translation of the subpage. It finds the (*C*) bit of the subpage cleared, which means that the subpage is not accessed by any other core yet. Thus, the (*C*) bit is set and the identity of the requester core (core 1) is recorded in the keeper field.
3) *Third (Brown):* Core 2 experiences a miss in its TLB for the same subpage like the previous operation. After looking up the page table for that subpage, it turns out both (*C*) and (*P*) bits of the subpage are set. Therefore, the keeper field should be compared against the identity of the requester core. If there is a match, it means that the keeper core has already experienced a TLB miss and the page table entry is brought into the requester core's TLB, considering the subpage as privately accessed only by requester core. If the keeper field does not match with the identity of the core requesting the page table entry (like this example), it means that two different cores are attempting to access the data within the same subpage (core 1 and core 2 in this example). As a result, the OS decides to turn the status of corresponding subpage to shared by clearing the (*P*) bit. Moreover, the OS triggers the coherence recovery mechanism by informing the keeper core to restore the coherence status of cache blocks within that subpage. We will explain the coherence recovery mechanism with more detail in the following section.

B. Coherence Recovery Mechanism

When a page is brought into the memory for the first time, the OS marks its subpages as private in the page table. The directory cache does not need to track private blocks. If at a certain point, we realize that our assumption about the status of a block is no longer valid, we need to recover from this situation. Otherwise, the caches might not remain coherent. In this paper, the o-CPT determines when a private subpage becomes shared. So, rather than the OS as it is the case in the study [10], the o-CPT triggers a coherence recovery, which is responsible for restoring the coherence for all blocks within the subpage. We use a similar recovery mechanism proposed in [10]. In this paper, authors propose two strategies, namely, flushing-based recovery and updating-based recovery mechanisms. Their results show that these two strategies are slightly different in terms of performance. Similar to some earlier studies, our recovery mechanism uses a flushing-based mechanism and performs following operations in order to ensure safe recovery from status change of a subpage from private to shared.

1) First, on the arrival of recovery request, the keeper first should prevent accesses to the blocks of that subpage by setting the subpage's (*L*) bit in the TLB entry.
2) Second, the keeper should invalidate all the blocks corresponding to that subpage in its private cache.
3) Third, the keeper also should take care of the pending blocks in its miss status holding registers (MSHRs). If there are any blocks within that subpage in MSHRs, they should be evicted right after the operation completes.

Once these steps are over, the keeper sends back an acknowledgment to announce the completion of the recovery. At this point, the core which initiated the recovery changes the status of that specific subpage to shared and continues its operation.

C. Directory Cache Organization

High associativity requirement of duplicate tag directories make them energy-inefficient. Sparse directories are viable option for this problem. However, because of their limited associativity, the number of evictions in sparse directories caused by adding a new entry to the directory might increase dramatically. Since any eviction in the directory requires invalidation of all the copies of that block (in all the private caches) in the system, performance of the system will be jeopardized. To overcome this problem, sparse directories should be managed intelligently.

In this paper, we focus on sparse directories and try to utilize our private data detection mechanisms to address their aforementioned limited associativity problem as follows. As we discussed earlier, we do not need to keep track of the private data. Thus, we can avoid polluting the directory cache

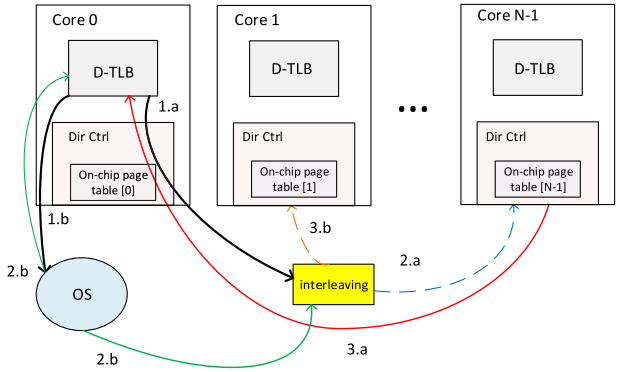

Fig. 3. Structure of the o-CPT in a CMP with private per-core TLBs.

with private data. We simply hold the states for the shared data blocks and not for the private ones. As we will show in the evaluation section, we can dramatically decrease the directory cache eviction rate and mitigate the inevitable performance degradation due to high directory eviction counts. The invalidation of the blocks related to the evicted directory entries is performed as normal. As will be shown later, our approach can get acceptable performance results even for directories with low associativity. Note that in the rest of this paper, we use directory and sparse directory (SD) interchangeably.

D. On-Chip Page Table

Memory block classification at subpage level may increase the frequency of OS involvement in updating the maintenance bits belonging to subpages stored in OS page table entries, nullifying some portion of performance benefits of subpage level data classification. For this reason, as a contribution, we show how we can negate the possible performance degradation by introducing the o-CPT. Moreover, the proposed method also enables us to boost the performance of the virtual memory management in many-core systems by exploiting commonality in address translations across cores in CMPs.

We propose our o-CPT as follows. As discussed before, recently L2 TLBs have been proposed for CMPs to take advantage of the commonality in address translations among the cores. With some modifications (additional storage in L2 TLB entries to keep track of private/shared status of subpages), we can turn L2 TLBs into an o-CPT and implement it as a part of directory cache controller. In address translations and retrieving the status of data blocks, we avoid snooping by distributing pages based on their addresses into o-CPT slices located in directory controllers. Distribution is done by interleaving the page entries according to the least significant bits (LSBs) of virtual page numbers (for example, with 16 cores, we use 4 LSBs). In this way, for each miss in the private TLBs, the core sends the request for the page address translation only to one of the cores' o-CPT. This makes our method more scalable compared to other proposed cooperative TLBs based on snooping. Note that in the rest of this paper, if it is not stated otherwise, when we use TLB we mean L1 TLB.

Fig. 3 depicts the structure of our o-CPT and outlines how a miss in one of the private TLBs can be resolved by one of the o-CPTs. After a TLB miss occurs in core 0 for an address translation of page "*a*," the request for finding the page information for this page is sent directly to the core which may have the requested entry (in this example core $N-1$). Then, the translation is forwarded back to the requesting core in case it is found in the o-CPT (red line). In the second example, the search for finding the translation for page "*b*" was not successful in the corresponding o-CPT. Therefore, with OS involvement, the entry found in the page table in memory will be written to one of the o-CPTs after interleaving (core 1) and also the TLB of the requesting core (core 0).

The proposed approach does not force major hardware costs nor operation overheads. For each TLB miss, there is an address interleaving (which can easily be done by a shift and AND operation) to find the location of the o-CPT that might have the corresponding physical address for a virtual address. As o-CPTs reside in the directory caches, each access to an o-CPT is equivalent to an access to a directory cache. This implies that we can replace a very costly page table walk with a very low cost cache access, each time the access to the o-CPT is a hit.

In our experiments, we show how much we can avoid referring to OS, thanks to exploiting the o-CPT. To show increasing the capacity of TLB cannot solely decrease TLB miss ratio, we also test our method for different TLB sizes.

Until now, we try to explain the issues on how to implement our QDBC approach. Next, we try to explain our DBC approach and additional provisions required to support it.

E. Dynamic Block Classification

The QDBC approach employs a data classification methodology to differentiate between private and shared data blocks at subpage level and the coherence protocol is bypassed for private data blocks. However, the QDBC approach has a major shortcoming: when a data block becomes shared at a specific moment during runtime, it is assumed to stay so. Therefore, all accesses to this data block from that particular moment to the end of program are managed by the coherence protocol. However, it is possible to detect many more private data blocks at runtime by more closely scrutinizing data blocks. The overriding need here is how to decide shared blocks becoming private again.

1) Monitoring Data Block Evictions: There may be different techniques regarding how to transition status of data blocks from shared to private over and over again. For example, we can modify *the cache decay technique*, originally proposed by Kaxiras *et al.* [19] to reduce leakage power dissipation in cache memories, to monitor data blocks in private L1 caches for collecting information that can be used in turning the shared blocks into private. Because of hardware complexity and accuracy concerns for determining when to turn a cache line off, we do not employ this technique. Instead, in our DBC approach, we adopt a technique where we monitor data block evictions in L1 private caches to determine when data blocks become private again. More specifically, when the last data block of a subpage is evicted from the L1 private cache hierarchy, the status of all the data blocks within that particular subpage are

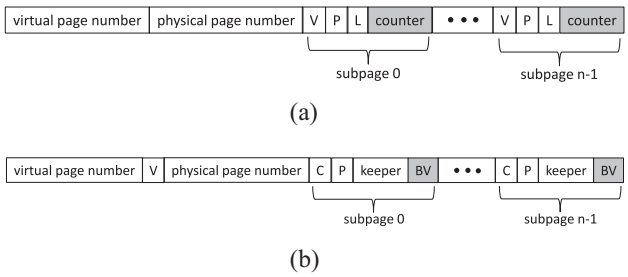

Fig. 4. TLB and page table entry formats used in our DBC approach. The new fields are shown shaded. (a) TLB entry format. Each subpage portion of the entry has an extra field called counter to hold the number of copies of data blocks that belong to the corresponding subpage cached in the core's L1 cache. (b) Page table entry format. Each subpage portion of the entry has an extra field called BV, which includes one bit for each core indicating whether there exists any data block within the corresponding subpage cached in this particular core's L1 cache.

transitioned to private again. To do so, for each subpage we need to count the number of copies of data blocks stored in private L1 caches, whose details are explained below.

To apply our modified data classification approach, first of all, we extend TLB and page table entry formats as shown shaded in Fig. 4. TLB entries contain a new field called as *counter* for each subpage, which is used to keep the number of copies of data blocks within the corresponding subpage cached in the core's private L1 cache, as shown in Fig. 4(a). When a data block is brought into the L1 cache, the TLB searches the entry of the corresponding page and increments the counter of the subpage to whom the data block belongs by 1. On the other hand, when a data block is evicted from or invalidated in the L1 cache as a result of data coherence maintenance (for example, its cached remote copy could be modified in another private cache), the counter for the subpage is decremented by 1. Note that these actions take place inside individual tiles and do not increase the on-chip message traffic.

We also introduce a new field for each subpage in page table entries (as well as in o-CPT entries), which is called as bit-vector (BV) as indicated in Fig. 4(b). BV includes one bit for each core in the CMP system to show whether there exists any data block within the subpage cached in the core's L1 private data cache. If this bit is 1, it means there is at least one data block that belongs to that subpage cached in the private L1 cache. If it is 0, this means that there is no data block within the subpage cached in the core's L1 cache.

Second, we consider L1 caches to be virtually indexed and physically tagged, which is the most prevalent form of cache access due to lower latency and being free from aliasing problems. When we use virtually indexed, physically tagged L1 caches, apart from physical tags of blocks, the virtual tags of page addresses are also required to be stored in cache blocks. We need to know the virtual address of an evicted or invalidated L1 cache block; because, as mentioned prior, the counter for the corresponding subpage is decremented by 1 in the involved TLB entry, and TLBs work with virtual addresses.

Third, we need to adapt the coherence recovery mechanism used in our QDBC approach, which was originally presented in the study [10], to deal with the block classification of DBC. This is particularly necessary for reclassifying shared blocks as private and preventing race conditions in accessing the involved blocks. More details are explained below.

2) Modifications to the Coherence Recovery Mechanism: We need to make some modifications to the coherence recovery mechanism in order to restore the coherence status of the blocks whose status is transitioned from shared to private. As explained prior, the counter field in TLB entries indicates the number of data blocks within the subpage residing in the private L1 cache of the core. Whenever a block is evicted or invalidated from the L1 cache, the corresponding L1 cache notifies the TLB (using the virtual tag of the page) to update the counter for the respective subpage. When the counter becomes 0, it means that there are no data blocks in the L1 cache, so the TLB performs two operations. First, it locks the corresponding subpage portion of the TLB entry by setting the (*L*) bit, which prevents the core from issuing new requests for any data block within this subpage. Second, a message is sent to the involved o-CPT slice to inform that there are no data blocks within the subpage cached in the private L1 cache. Upon arrival of this message, the o-CPT first clears the corresponding bit in the BV field of the subpage. After that, the o-CPT do one of two things depending on the content of BV. If there is at least one bit set in the BV field, this means that some data blocks within the subpage have not yet been evicted from the L1 private cache hierarchy. So, the o-CPT sends a specific acknowledgment message to the TLB in question to unlock the corresponding subpage portion of the TLB entry to allow the core to issue new data requests within the subpage.

On the other hand, if all the bits in BV are cleared, this means that there does not exist any data block in the L1 cache hierarchy of the CMP system. So the o-CPT clears both the (*P*) and (*C*) bits of the subpage in the TLB entry. Then, it issues a recovery request for the corresponding subpage. Since the o-CPT slice does not know which TLBs hold the page address translation, the recovery request is broadcasted to all TLBs in the system. Upon the recovery request arrival, each TLB checks whether it has the page translation. Those TLBs that have the address translation clear the (*V*) bit of the subpage to force the first address translation for data blocks within the subpage in the future to go through the o-CPT.

IV. Evaluation Methodology

A. Simulation Infrastructure

We evaluate our proposal with gem5 full-system simulator [8] running Linux version 2.6. gem5 uses RUBY, which implements a detailed model for the memory subsystem and specifically cache coherence protocol. For modeling the interconnection network, we use GARNET [2], a detailed interconnection simulator also included in gem5. We apply our ideas to *MOESI_CMP_Directory*, which is a directory-based cache coherence protocol implemented in gem5. We present results for a system consisting of 16 cores with level one (L1) private data and instruction caches, and a shared level two (L2) cache. Table I provides the details of our simulation environment. In the rest of this paper, this configuration is considered as the base setup and if not stated otherwise, this setup will be used in our simulations.

TABLE I
DEFAULT SIMULATION PARAMETERS USED IN OUR EXPERIMENTS

Processor	16 Alpha cores, 2 GHz
Private L1 Data Cache	32KB (512 entries), 4-way associative, 2 cycle hit latency, 64B cache-block size
Private L1 Instruction Cache	32KB, 2-way associative, 2 cycle hit latency, 64B cache-block size
Shared L2 Cache	32MB (2MB per tile), 8-way associative, 14 cycle hit latency
Private L1 TLB	64 entries, 4-way associative, 1 cycle hit latency
Shared L2 TLB (per tile)	256 entries, 4-way associative, 2 cycle hit latency
Directory cache (per tile)	512 entries, 16-way associative (base setup), 2 cycle hit latency
On-chip page table (per tile)	256 entries, 4-way associative (only in our QDBC and DBC setups)
Network	2D folded torus (4x4) , fixed Garnet interconnection model, 1 cycle link latency, 2 cycle one-hop latency
Cache coherence protocol	MOESI_CMP_Directory
Page size	8KB

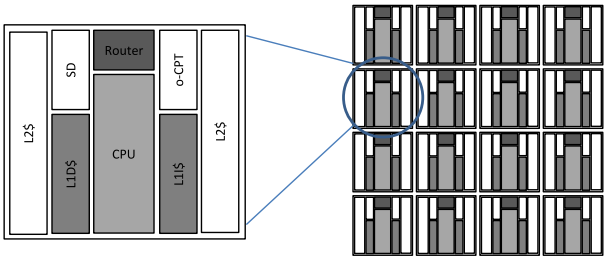

Fig. 5. Structure of a typical tiled CMP architecture. Tiles are interconnected into a 2-D folded torus. Each tile contains a core, shared-L2 slice, L1 instruction and data caches, SD, o-CPT, and a router. Message exchanges among the tiles are performed over routers.

In this paper, we consider tiled CMP architectures. A tiled CMP is made up of multiple identical or close-to-identical tiles each with a core, L1 data cache (L1D), L1 instruction cache (L1I), shared L2 slice, data TLB, instruction TLB, SD, o-CPT (in the figure), and a router, as shown in Fig. 5. Tiled architectures are very appealing from economic, manufacturing, and physical design points of view [17], [35] since these architectures enable developers to concentrate on the design of a single tile that can be replicated across the die, resulting in CMP designs with various number of computing cores. Tiles are connected over a fast on-chip interconnection network. Specifically, in our study, a 2-D folded torus network is used.

In tiled CMP architectures, each tile has its own L2 slice and L2 cache can be organized in two different ways: 1) each L2 slice serves as a private cache for the corresponding tile's core, which is referred to as private L2 organization or 2) each distributed L2 slice is regarded as a portion of total capacity of the system's L2 cache, which is referred to as shared L2 organization. Only one copy of data blocks is kept in the shared L2 organization, which maximizes the effective on-chip capacity and minimizes on-chip cache misses. In this paper, we adopted shared L2 cache organization.

We run our multithreaded applications on a 16-core tiled CMP system. It has a two-level cache hierarchy: L1 private and L2 shared. Cache blocks are distributed across tiles in an address interleaved fashion. The cache blocks on remote L2 slices are accessed through the interconnection. Our tiled CMP architecture leads to varying access latencies for the LLC, i.e., NUCA design [20] in which the load-to-use latency depends on the physical distance between the location of the requesting core and where the requested data resides. In our simulation platform, we assume a 2-D-folded-torus (4×4 tiles) on-chip interconnection based on fixed Garnet network model [2] with 1 cycle link latency and 2 cycle hop latency.

TABLE II
BENCHMARKS AND RESPECTIVE INPUT FILES

Benchmarks	Input
Parsec 2.1	
Blacksholes Bodytrack Canneal Fluidaniate Swaptions	simsmall
Splash-2	
Cholesky	tk15.O
Raytrace	Teapot environment
Waternsq	512 molecules, 3 time steps
Radix	1048576 keys
Ocean	258*258 ocean

Since the shared L2 organization allows a fixed, unique location for each data block in the aggregate L2 cache, the coherence protocol requires to keep track of the status of only the data blocks stored in the private L1 caches of cores. We use an SD structure to store the coherence status of data blocks in private L1 caches. Each tile contains a slice of directory structure and the status of data blocks in L1 private caches are distributed among these slices based on data blocks' physical addresses.

In our base setup, we assume that, in each tile, the number of SD entries is equal to the number of data blocks in a private L1 cache (i.e., 512). All setups use a two-level TLB hierarchy. In QDBC and DBC approaches, we tailored the L2 TLB for shared DBC and, to differentiate it from regular shared L2 TLBs used in the base setup, we call it as o-CPT. The number of L2 TLB entries in the base setup is equal to the number of o-CPT entries in the setups using QDBC and DBC approaches, which is 256. As discussed before, compared to L2 TLB entries in the base setup, o-CPT entries need some extra fields required for maintaining shared/private status of data blocks. In order to make fair comparisons between base setup and our QDBC and DBC approaches, we try to fix the total area devoted to L2 TLB and SD structures as much as possible. Therefore, we make sure that when QDBC and DBC approaches are used, the number of SD entries is equal to half of the number of SD entries in the base setup (i.e., the half of 512, which is 256). Also, we assume that while the associativity of the SD structure in the base setup is 16, our QDBC and DBC approaches employ 4-way SD structures.

B. Benchmarks

We evaluate our approaches with ten different multithreaded workloads from two commonly used suites (SPLASH-2 [34]

and PARSEC 2.1 [7]). As we execute a large set of simulations to provide a comprehensive sensitivity analysis, where each simulation takes a considerable amount of time to run in our full-system simulator, we simulated the applications mostly for small size data-sets, as indicated in Table II. For our experimental results, we only consider the parallel phase of benchmarks or region of interest; and the number of threads used by each application is set based on the number of cores in the system, which is 16 by default.

C. Setups Taken Into Consideration

For each benchmark listed in Table II, we performed experiments by using three different experimental setups. The details of these setups are explained below.

1) *Base:* This reflects the default case where no DBC is employed and every data access is managed by the coherence protocol. In this setup, we assume that the system includes an SD structure across the tiled CMP system and each tile includes a slice of the total directory. Each data block is mapped to a home tile based on $\log_2 n$ LSBs of its physical address, where n is the number of tiles/cores in the system, and a home tile is responsible for keeping coherence information for those blocks mapped to it. The system includes a two level TLB hierarchy. We assume that each tile contains a slice of SD with 512 entries, which is twice as many as entries in the SDs employed in DBC and QDBC approaches. Although a 64-way duplicate tag directory is required in order to avoid any invalidation in private L1 caches due to evictions in the directories—our L1 caches are four-way set associative and we consider a tiled CMP with 16 cores—we use an SD with 16-ways in this setup for some practical reasons, such as excessive power consumption and design complexity concerns.
2) *QDBC:* This is the setup where our QDBC approach is employed. A DBC at subpage level is used to differentiate private data from shared data, and the coherence protocol for the shared data are deactivated. A data block is considered to be private when it is referred to for the first time and it stays private as long as their corresponding subpage is accessed by only one core. During that particular period of time, the coherence protocol is deactivated for all accesses to the data block and no coherence information for that particular page is kept in the duplicate tag directory. When another core tries to access any block within the corresponding subpage, which is detected either by the directory controller's check in the o-CPT (if the corresponding page information exists in the o-CPT structure) or by the OS intervention, the coherence recovery mechanism is called upon to restore the coherence status of all the blocks within this particular subpage and the coherence protocol starts to keep track of those blocks. In QDBC approach, when a data block becomes shared it stays so until the end of program execution, so dynamic block behavior is not exploited sufficiently. We assume that this setup includes an o-CPT and the number of entries of the o-CPT and SD structures per tile is equal to 256, which is half of the entries in the SD structure employed in the base setup. The SD is assumed to be 4-way set associative (compared to 16-way set associative SD used in the base setup).
3) *DBC:* It is the setup where our DBC approach is used. This setup is similar to the setup with the QDBC approach regarding the employed hardware components and their sizes, i.e., both employ the o-CPT and SD hardware structures and these structures are made up of 256 entries. The only difference between these two setups is the data classification approach employed. While they both classify data blocks at subpage level, unlike QDBC, the DBC approach takes advantage of runtime behavior of data blocks and can reclassify data blocks accordingly. More specifically, the DBC approach monitors the data blocks within a subpage and whenever all of them are evicted from the private L1 caches, these data blocks become private again, allowing more accurate DBC.

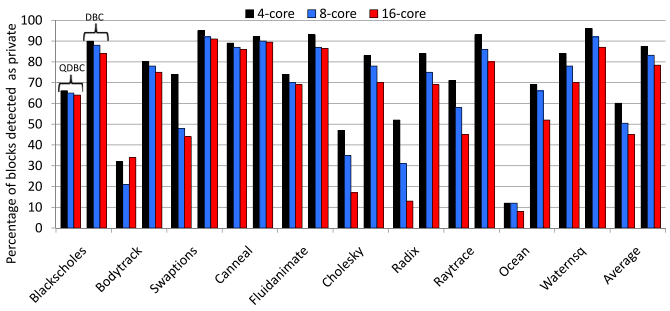

Fig. 6. Percentage of data blocks detected as private when the QDBC and DBC approaches are applied to tiled CMPs with various number of cores.

Note that in the next section besides these three setups, we also provide some experimental results of a page-granularity block classification, which uses the same hardware components that are employed in the QDBC and DBC setups. For this page-granularity block classification we adopted the technique proposed by Cuesta *et al.* [10]. Although they proposed their approach for private caches in systems like Magny-Cours [9], where there exist multiple dies and each die has its own processing cores and a die-wide large private cache, we tailored it to L1 private caches in single-die CMPs.

V. Performance Evaluation

A. Private Blocks

The amount of data blocks detected as private with our QDBC and DBC mechanism is depicted in Fig. 6 for ten different benchmarks tested. We have tested these benchmarks for CMPs with various number of cores ranging from 4 to 8, to 16. Here, pages are assumed to be composed of four subpages. All values are reported as normalized with respect to the base setup. Each benchmark has two groups of bars and each one contains three bars. These three bars are for CMP systems with 4, 8, and 16 cores, respectively. From Fig. 6, it is very clear that exploiting dynamic behavior of data blocks at runtime is crucial. Hence, we make the following observations.

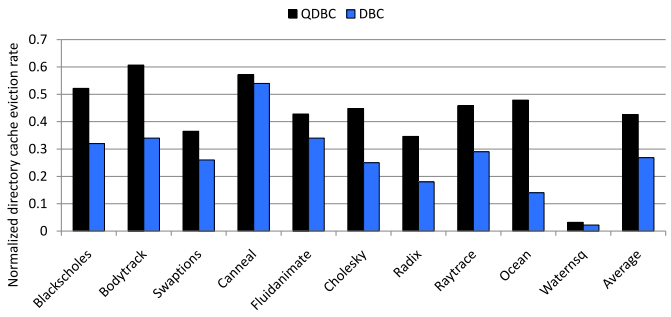

Fig. 7. Normalized directory cache eviction rate. All values are normalized with respect to the base setup.

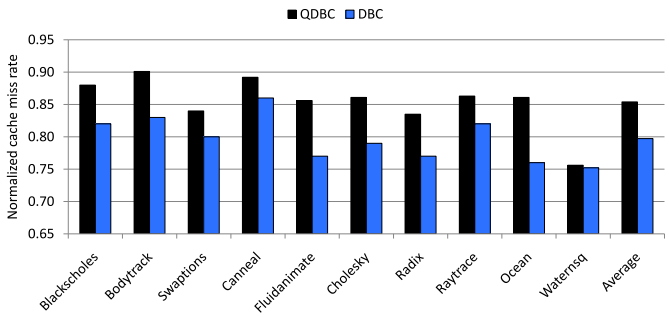

Fig. 8. Normalized L1 cache miss rate. All values are normalized with respect to the values of the base setup.

First, compared to the QDBC approach, the DBC approach is able to detect a considerably larger portion of data accesses as private. The average percentages of blocks detected as private with the QDBC approach are approximately 60%, 50%, and 45% for CMP systems with 4, 8, and 16 cores, respectively. On the other hand, the DBC approach can classify data block accesses as private, on average, with 87%, 83%, and 78% for 4, 8, and 16-core CMP systems, respectively. Second, when we use the QDBC approach in a CMP with higher number of cores, the effectiveness of the data classification drops considerably, from 60% for 4 cores to 45% for 16 cores (an average of 25% degradation). The corresponding values for the DBC approach change just from 87% to 78%, a deviation of 10%. This means that the DBC approach is more resilient to many-core CMPs. Third, some benchmarks which can hardly benefit from the QDBC can indeed enjoy relatively huge benefits from the DBC approach. For example, the QDBC approach can categorize only 13% and 8% of accessed data blocks as private for Radix and Ocean benchmarks for a 16-core CMP while 69% and 52% of data blocks in those benchmarks are classified as private with the DBC approach.

B. Directory Cache Eviction

As we showed earlier, a considerable amount of accessed memory blocks are private; and we do not keep track of those blocks in directory caches. By not polluting the directory cache with status of private blocks that do not need coherence maintenance, we would have less eviction in directory cache even for caches with lower associativity. Fig. 7 shows directory cache eviction rate of our approaches normalized with respect to the base setup.

There are two main factors that enable our approaches to improve the directory cache eviction rate. First one is the ability to detect more private data blocks and, the second one is the shared block access pattern. Arguably, if we can detect more private data, we can avoid more evictions in the directory caches. For instance, with QDBC the number of directory evictions has been reduced for Waternsq more than Cholesky, since we were able to detect higher number of private data blocks for Waternsq compared to Cholesky (70% for Waternsq and 17% for Cholesky as can be seen in Fig. 6). However, this is not the only factor which affects reduction in the directory eviction. The sharing pattern of an application can also play an important role in the number of evictions that happen in the directory. For instance, for Canneal, we were able to detect a high percentage of private data blocks (86%) compared to Bodytrack (34%). But because of the difference in sharing pattern for these two applications, we observe rather same normalized eviction rates. This is due to the temporal communication behavior of these two applications. In Canneal, the communication between the cores take place throughout the execution of the application, whereas, in Bodytrack, for the majority of the parallel phase, there is limited communication between cores [33]. Therefore, in Canneal, the chance for a possible conflict will be higher than Bodytrack. Another observation from Fig. 7 is the dramatic improvement in directory eviction ratio for Waternsq application. Based on the results reported in the study [33], in Waternsq and Waterspa (is not used in this paper!) all cores are actively involved in a producer-consumer pattern. Furthermore, based on this observation, they conclude that a broadcast-based technique is likely to benefit for Waternsq and Waterspa. We also observe that, Waternsq benefits the most, considering the eviction rate aggregation for all the directory caches in the system. The last observation is that, compared to QDBC, the DBC approach can dramatically reduce the directory cache eviction rate. As can be seen, with QDBC and DBC, on average, we have 58% and 73%, respectively, less evictions in the directory cache compared to the base setup without any data classification.

C. Private Cache Misses

One of the primary advantages of reducing directory cache eviction is the reduced invalidations at the last level private caches (in our system, the L1 cache). This is due to the fact that any eviction of a block in directory cache implies invalidation of all the blocks in any of the L1 caches that correspond to that block. Fig. 8 shows the L1 cache miss ratio for ten different multithreaded applications normalized with respect to the base setup. There are two bars for each benchmark. While the first bar shows the cache miss ratio for QDBC, the second bar presents the cache miss ratio for the DBC approach. Through the QDBC and DBC approaches, we have about 15% and 20% average reductions, respectively, in private L1 cache miss ratio. In general, we have better normalized cache miss values, for those applications with fewer misclassified blocks. In other words, most of the time the higher the number of

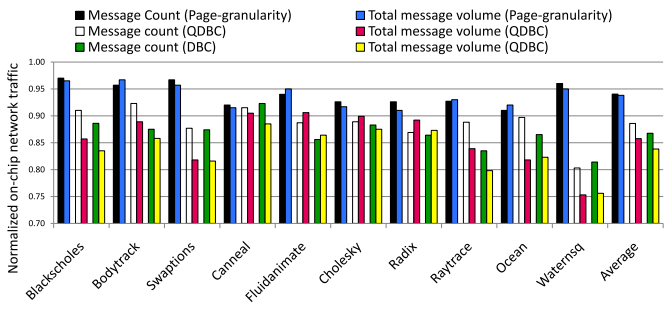

Fig. 9. Normalized network traffic message count and total message volume for different data classification approaches.

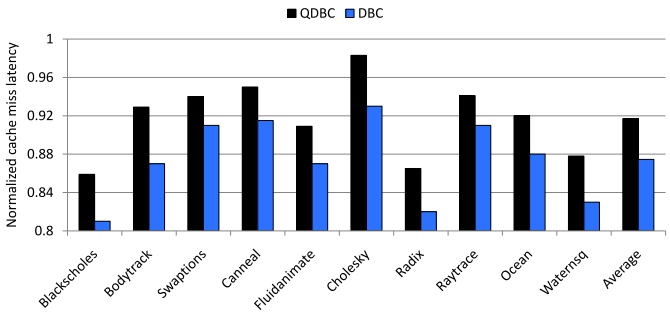

Fig. 10. Normalized L1 cache miss latency. The DBC approach allows faster miss resolution than QDBC.

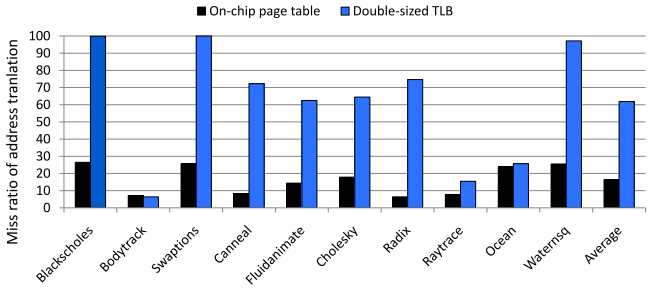

Fig. 11. Normalized miss rate of finding page translation on-chip with the QDBC approach.

private blocks detected out of all actual private data blocks exist, the higher the reduction is in the number of cache misses.

D. On-Chip Network Traffic

The number of messages as well as total amount of message volume exchanged in the system are also reduced by our approaches. This is because evictions in directory caches and processor cache misses impact both the message count and message volume communicated through the on-chip network. For instance, for resolving a miss in the first level private cache, different controllers in the system need to exchange request, forward, and response control/data messages with one another. In Fig. 9, we present the message counts and the total message volume normalized with respect to the base setup. There are three groups of bars for each benchmark in the figure: the bars in the first, second, and third groups correspond to the values for the page-granularity, QDBC, and DBC data classification approaches, respectively. Moreover, in each group the first bar shows the on-chip network traffic in terms of message counts while the second bar presents the network traffic in terms of total message volume.

Our QDBC approach reduces the total number of messages between approximately 8% and 20%, with an average of 11%. It can also decrease the total message volume communicated over the on-chip network between around 9% and 25%, with an average of 15%. According to the figure, when we employ our DBC approach, the number of on-chip messages and the total message volume can be reduced by, on average, 13% and 16%, respectively. With the page-level data classification, the reductions in the number of on-chip messages and the total message volume are, on average, around 6%, which are quite smaller than the network message traffic savings obtained with the QDBC and DBC approaches. Since we have observed quite similar results for different subpage sizes, here we provide the values only for a system with four subpages per page.

An important observation from Fig. 9 is that the effects of our two approaches on message traffic over on-chip network is quite similar. Compared to the QDBC approach, the DBC approach classifies more data blocks as private which in turn enables the coherence protocol to be deactivated more frequently, resulting in more reduction in the amount of on-chip message traffic. On the other hand, the more data blocks are detected as private, the more frequently the coherence recovery mechanism is called to restore the coherence status of data blocks transitioned, which increases the amount of on-chip message traffic. These two opposite factors can roughly balance out, causing the amount of on-chip network traffic to stay more or less the same.

E. Cache Miss Latency

With our approaches, we are also able to reduce the average latency for resolving cache misses. More specifically, we avoid referring to the directory cache for those requests associated with the private blocks. Therefore, some portion of the requests experience less latency when a cache miss occurs, lowering the overall average latency of a cache miss. Fig. 10 depicts the average cache miss latency normalized with respect to the base setup for our applications. As can be seen, on average, we resolve the cache misses 8% and 12% faster than the base system when we employ the QDBC and DBC approaches.

F. Performance of On-Chip Table

In this part, we show the improvements of virtual memory management by introducing the o-CPT. To this end, we have conducted some experiments with special configurations (an o-CPT with 64 entries). Fig. 11 shows the percentage of TLB misses that also experience a miss in the o-CPT. In other words, it shows how much we can avoid accessing the costly OS page table by finding the required page translation in the distributed o-CPT. As an example, for Canneal benchmark, only 8% of page translations which cause a miss in a private TLB cannot be found in the o-CPT. Therefore, the remaining 92% of accesses find the right translation in the o-CPT after they experienced a miss in private TLBs. On average,

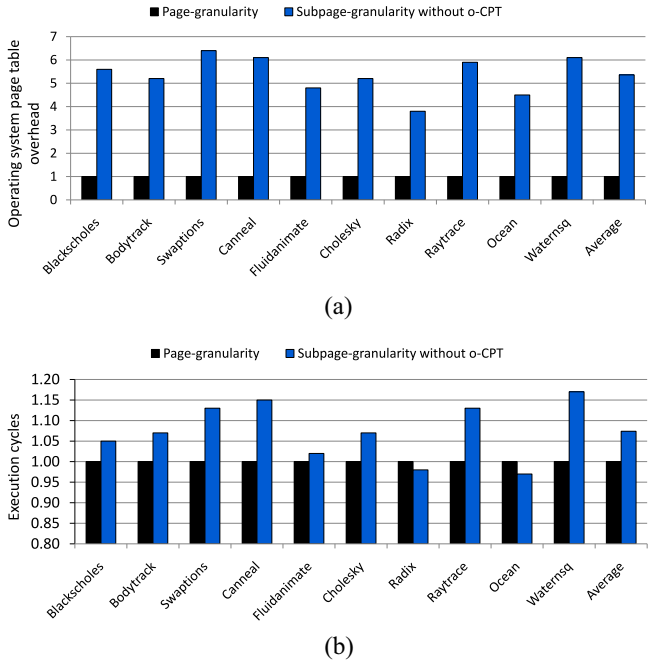

Fig. 12. Performance values are given when a subpage level data classification is applied without using an o-CPT. With our DBC approach, applying subpage data classification without employing an o-CPT severely affects the performance. (a) Normalized values of OS page table accesses. (b) Normalized execution cycles.

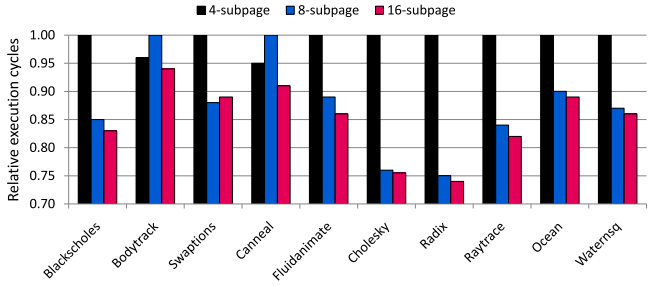

Fig. 13. Relative execution time for different subpage sizes with the QDBC approach. We test our approach for 4, 8, and 16 subpages.

our approach prevents 84% of accesses to OS by introducing the o-CPT.

We also compared the effectiveness of our o-CPT with a modified version of the base setup, where one-level TLB hierarchy is employed and the TLB is double the original size (each core employ a private L1 TLB with 128 entries). As can be seen from Fig. 11, by doubling the TLB size, we can only eliminate 39% of the accesses to the OS page table. So we can conclude that increasing the cache capacity for page translation cannot solely improve the performance. Moreover, exploiting an effective technique which enables sharing page translation between the cores is also crucial for providing a fast virtual page translation.

To understand how important the o-CPT is for the dynamic data classification approach, we have created a specific setup whose experimental results are presented in Fig. 12. This setup is similar to the setup with the DBC approach, but the only difference is that, in this particular setup, we employ a subpage level data classification without an o-CPT component, which means that the data classification process and restoring the coherence status of data blocks in the coherence recovery mechanism are realized through OS page table. All the experimental results are reported as normalized values with respect to the values of a setup in which a page level data classification is applied. We consider a tiled CMP with 16 cores in both setups, and the number of subpages is assumed to be 4. The overheads in terms of the number of OS page table accesses are given in Fig. 12(a). Note that, here we only take page table accesses related to data classification and coherence maintenance into account, not considering page table accesses related to page address translations. As can be seen form Fig. 12(a), applying subpage DBC without employing an o-CPT severely increases the number of OS page table accesses. These increases can vary from 3.8× for the Radix benchmark to 6.4× for Swaptions.

Such large increases in costly OS page table accesses may result in serious degradations on the overall system performance, which are depicted in Fig. 12(b). From this figure, we can see that, for all benchmarks except Radix and Ocean, the page level data classification outperforms the subpage level page classification not employing an o-CPT. These performance values show how important using an o-CPT is for the dynamic data classification approach.

G. Execution Time

First, we explore how much we can increase the granularity of private data detection to improve performance with the QDBC approach. For doing so, we run a set of simulations for different subpage sizes. Fig. 13 shows relative execution times for three different number of subpages per page (4, 8, and 16). As can be seen, in all the benchmarks except Bodytrack and Canneal, a system with eight subpages shows better performance compared to a system with four subpages. The other observation is that a system with 16 subpages shows the best performance for all the benchmarks except Swaptions. Moreover, subpage sizes 8 and 16 show almost similar performance in most of the applications tested. The most important conclusion on subpages is that when we use quasi-dynamic DBC, increasing the granularity does translate into performance improvement up to a certain point. In our experiments, we observe that dividing pages into more than 16 subpages do not bring additional benefits.

Second, to understand the effects of size of subpages on execution cycle when we employ our DBC approach, we have conducted some experiments with 4, 8, 16, 32, and 64 subpages per page. Note that we assume the size of pages is 8 KB, so the involved subpage sizes are 2 KB, 1 KB, 512 B, 256 B, and 128 B, respectively. The results of our sensitivity analysis are presented in Fig. 14. Our major findings from this analysis are as follows. First, the average performance improvements of dynamically classifying data blocks as private are quite significant: 5%, 11%, 15%, 17%, and 18% for 2K, 1 KB, 512 B, 256 B, and 128 B subpage sizes, respectively. Second, as can be easily seen from the figure, for smaller subpage sizes (larger

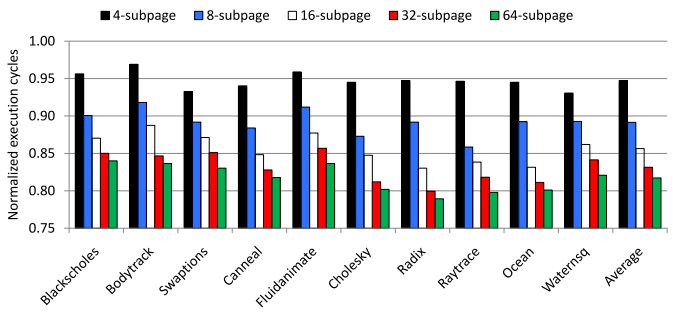

Fig. 14. Execution cycles for the DBC approach. All performance values are reported as normalized with respect to the base setup.

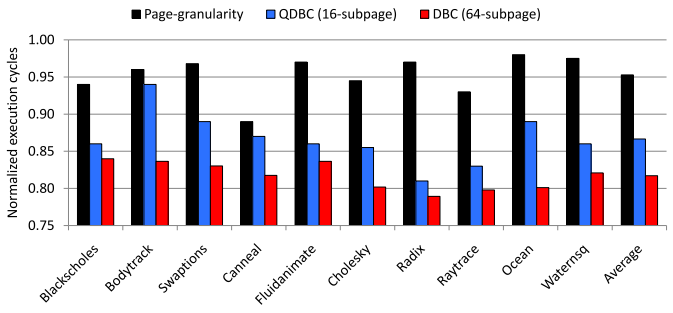

Fig. 15. Comparison of execution cycles of page-granularity block classification, the QDBC approach and the DBC approach. All performance values are normalized with respect to the performance values of the base setup.

number of subpages per page) we have better performance improvements. This can be ascribed to the fact that the possibility of evicting a subpage as a whole from private caches is higher for smaller subpage sizes, which increases the amount of data blocks reclassified as private. Third, beyond 128 B subpage size we do not get any additional benefits, so we do not provide results for smaller subpage sizes.

It can be noticed easily performance gains with DBC is quite larger than QDBC. For example, for 16-core systems with pages composed of four subpages, the average performance improvement is 4% with QDBC while the average performance improvement with DBC is 5%. This argument is valid for pages with larger number of subpages per page. We believe that there are two main reasons for this. First, the DBC approach is able to classify data blocks more accurately, so it can detect many more data blocks as private. Second, the location of private blocks is very important from the performance point of view. The coherence overhead for accessing data blocks in DRAM can be largely hidden due to high DRAM access latency, so deactivating coherence protocol for the private blocks accessed in DRAM usually brings no performance gains. To investigate this, we have conducted some experiments to find out location of accessed private blocks. Due to space restriction, we here provide only a summary of our findings. According to our experimental results, with the QDBC approach, the private blocks are found in DRAM, L2 cache, and L1 cache, on average, 42%, 5%, and 53%, respectively. On the other hand, when we use the DBC approach, the corresponding average values are 9%, 13%, and 78% for DRAM, L2 cache, and L1 cache, respectively. These values indicate that, unlike QDBC, with the DBC approach most of the private data block accesses happen in L1 data caches, which has more benefit on the system performance.

Third, in Fig. 15, we present the performance values of page-granularity block classification, the QDBC approach and the DBC approach. We choose 16 and 64 subpages per page for the QDBC and DBC approaches, respectively, since they perform best with them as shown before. As can be seen from the figure, the QDBC and DBC approaches outperform the page-granularity block classification significantly for most of the benchmarks since our approaches are able to detect many more private data blocks at runtime. While the QDBC and DBC approaches can improve the overall system performance, on average, 13% and 18%, respectively, the page-granularity block classification can bring an average of 5% performance improvement. We believe that the main reason for these huge performance gaps is that page size like 8 KB are too large for the page-granularity block classification approach to detect enough amount of data blocks as private in private L1 caches.

H. Area Overhead

Compared to QDBC, the DBC approach requires larger page table entries because extra space is needed to keep BVs. The extra storage requirement of the DBC approach in a page table entry is $S \times (1 + 1 + \log_2(N) + N))$ bits, where N is the number of cores and S is the number of subpages [(Fig. 4(b)]. For a system with 16 cores, 4 subpages, 48-bit virtual, and 40-bit physical address spaces, 4 maintenance bits (such as valid bit), and 8 KB pages, the overhead will be around 133%. Because of this huge overhead, the DBC approach may seem impracticable. However, the DBC approach can work with very small SD structures without hurting performance. In prior experiments involving the DBC approach, we consider an SD structure with 256 entries, which is the half of SD entries in the base setup. We have made some experiments to find out its resilience to various SD sizes (with 128, 64, and 32 entries). Because of space restriction, here we only summarize our major findings. Our experimental results show that our DBC approach can employ an SD structure with as few as 64 entries and can still perform as well as when it uses an SD with 256 entries. This means that the area overhead caused by larger page table requirement in the DBC approach can be simply compensated by using an SD structure in much smaller size, thanks to its ability of classifying a considerable percentage of data blocks as private.

VI. Conclusion

This paper shows that, compared to subpage level data classification, QDBC and DBC can classify many more data blocks as private and can enhance the overall system performance accordingly. The extensive experimental results indicate that while the QDBC approach can enjoy data classification up to a subpage size of 512 B, DBC can get the maximum benefit with 128 B subpages. With QDBC it is possible to classify 45% of accessed data blocks as private even for 2 K subpages in a 16-core CMP. The corresponding

private block detection percentage when DBC is used is 78%, which emphasizes the importance of monitoring evictions of data blocks from the private L1 hierarchy. QDBC and DBC approaches can reduce the total message volume communicated over the on-chip network by, on average, 14% and 16%, respectively, for 2 KB subpages. Moreover, the DBC approach outperforms the QBC approach significantly, and can improve the overall system performance by 18% on average for 128 B subpages.

REFERENCES

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *Proc. Int. Symp. Comput. Archit.*, Honolulu, HI, USA, 1988, pp. 280–298.

[2] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, Boston, MA, USA, 2009, pp. 33–42.

[3] J. Ahn, D. Kim, J. Kim, and J. Huh, "Subspace snooping: Exploiting temporal sharing stability for snoop reduction," *IEEE Trans. Comput.*, vol. 61, no. 11, pp. 1624–1637, Nov. 2012.

[4] L. A. Barroso *et al.*, "Piranha: A scalable architecture based on single-chip multiprocessing," in *Proc. Int. Symp. Comput. Archit.*, Vancouver, BC, Canada, 2000, pp. 282–293.

[5] A. Bhattacharjee and M. Martonosi, "Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors," in *Proc. Int. Conf. Parallel Archit. Compilation Tech.*, Raleigh, NC, USA, 2009, pp. 29–40.

[6] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level TLBs for chip multiprocessors," in *Proc. Int. Symp. High Perform. Comput. Archit.*, San Antonio, TX, USA, 2011, pp. 62–63.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. Int. Conf. Parallel Archit. Compilation Tech.*, Toronto, ON, Canada, 2008, pp. 72–81.

[8] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, May 2011.

[9] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache hierarchy and memory subsystem of the AMD Opteron processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, Mar./Apr. 2010.

[10] 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 *Proc. Int. Symp. Comput. Archit.*, San Jose, CA, USA, 2011, pp. 93–103.

[11] M. Davari, A. Ros, E. Hagersten, and S. Kaxiras, "The effects of granularity and adaptivity on private/shared classification for coherence," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 3, p. 26, Oct. 2015.

[12] M. Davari, A. Ros, E. Hagersten, and S. Kaxiras, "An efficient, self-contained, on-chip directory: DIR1-SISD," in *Proc. Int. Conf. Parallel Archit. Compilation*, San Francisco, CA, USA, 2015, pp. 317–330.

[13] A. Esteve, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Efficient TLB-based detection of private pages in chip multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 3, pp. 748–761, Mar. 2016.

[14] A. Esteve, A. Ros, A. Robles, M. E. Gómez, and J. Duato, "TokenTLB: A token-based page classification approach," in *Proc. Int. Conf. Supercomput.*, Istanbul, Turkey, 2016, pp. 1–13.

[15] C. Fensch and M. Cintra, "An OS-based alternative to full hardware coherence on tiled CMPs," in *Proc. Int. Symp. High Perform. Comput. Archit.*, Salt Lake City, UT, USA, 2008, pp. 355–366.

[16] A. Gupta, W. D. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *Proc. Int. Conf. Parallel Process.*, Urbana, IL, USA, 1990, pp. 312–321.

[17] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal block placement and replication in distributed caches," in *Proc. Int. Symp. Comput. Archit.*, Austin, TX, USA, 2009, pp. 184–195.

[18] N. D. E. Jerger, L.-S. Peh, and M. H. Lipasti, "Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence," in *Proc. Int. Symp. Microarchit.*, 2008, pp. 35–46.

[19] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *Proc. Int. Symp. Comput. Archit.*, Gothenburg, Sweden, 2001, pp. 240–251.

[20] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, San Jose, CA, USA, 2002, pp. 211–222.

[21] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded Sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, Mar./Apr. 2005.

[22] J. Laudon and D. Lenoski, "The SGI origin: A ccNUMA highly scalable server," in *Proc. Int. Symp. Comput. Archit.*, Denver, CO, USA, 1997, pp. 241–251.

[23] Y. Li, R. Melhem, and A. K. Jones, "A practical data classification framework for scalable and high performance chip-multiprocessors," *IEEE Trans. Comput.*, vol. 63, no. 12, pp. 2905–2918, Dec. 2014.

[24] D. Lusting, A. Bhattacharjee, and M. Martonosi, "TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 1, pp. 1–38, Apr. 2013.

[25] M. R. Marty and M. D. Hill, "Virtual hierarchies," *IEEE Micro*, vol. 28, no. 1, pp. 99–109, Jan./Feb. 2008.

[26] S. S. Mukherjee and M. D. Hill, "An evaluation of directory protocols for medium-scale shared-memory multiprocessors," in *Proc. Int. Conf. Supercomput.*, Manchester, U.K., 1994, pp. 64–74.

[27] A. K. Nanda, A. T. Nguyen, M. M. Michael, and D. J. Joseph, "High-throughput coherence control and hardware messaging in Everest," *IBM J. Res. Develop.*, vol. 45, no. 2, pp. 229–243, Mar. 2001.

[28] B. W. O'Krafka and A. R. Newton, "An empirical evaluation of two memory-efficient directory methods," in *Proc. Int. Symp. Comput. Archit.*, Seattle, WA, USA, 1990, pp. 138–147.

[29] A. Ros, M. E. Acacio, and J. M. Garcia, "A direct coherence protocol for many-core chip multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 12, pp. 1779–1792, Dec. 2010.

[30] R. Singhal, "Inside Intel next-generation Nehalem microarchitecture," presented at Hot Chips 20, Stanford, CA, USA, 2008.

[31] M. Soltaniyeh, I. Kadayif, and O. Ozturk, "Boosting performance of directory-based cache coherence protocols with coherence bypass at subpage granularity and a novel on-chip page table," in *Proc. Int. Conf. Comput. Front.*, Como, Italy, 2016, pp. 180–187.

[32] S. Srikantaiah and M. Kandemir, "Synergistic TLBs for high performance address translation in chip multiprocessors," in *Proc. Int. Symp. Microarchit.*, Atlanta, GA, USA, 2010, pp. 313–324.

[33] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterisation of Splash-2 and Parsec," in *Proc. Int. Symp. Workload Characterization*, Austin, TX, USA, 2009, pp. 86–97.

[34] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. Int. Symp. Comput. Archit.*, Santa Margherita Ligure, Italy, 1995, pp. 24–36.

[35] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *Proc. Int. Symp. Comput. Archit.*, Madison, WI, USA, 2005, pp. 336–345.

Mohammadreza Soltaniyeh, photograph and biography not available at the time of publication.

Ismail Kadayif, photograph and biography not available at the time of publication.

Ozcan Ozturk, photograph and biography not available at the time of publication.