TLB-based Block-Grain Classification of Private Data

Bhargavi R. Upadhyay
Department of Computer Science and Engineering,
Amrita School of Engineering, Bengaluru
Amrita Vishwa Vidyapeetham, INDIA
u_bhargavi@blr.amrita.edu

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

Murty N S
Department of Electronics and Communication Engineering,
Amrita School of Engineering, Bengaluru
Amrita Vishwa Vidyapeetham, INDIA
ns_murty@blr.amrita.edu

***Abstract*—Sequential and parallel applications use most of the data as private in a multicore system. Recent proposals made use of this observation to reduce the area of the coherence directories or the memory access latency. The driving force of these proposals is the classification of private/shared memory data. The effectiveness of these proposals depends on the number of detected private data. The existing proposals perform the private/shared classification at page granularity, leading to a noticeable amount of miss-classified memory blocks.**

We propose a mechanism that works on block granularity using the translation lookaside buffer (TLB) to make accurate detection of private data, which increases the effectiveness of proposals relying on a private/shared classification. Simulation results show that the block-grain approach obtains 17.0% more accessed private miss data than the page-grain approach, which translates to an improvement in system performance by 6.02% compared to a page-grain approach.

***Index Terms*—directory-based cache coherence, private/shared classification, system performance**

I. Introduction

Shared memory multiprocessors with hardware support for cache coherence offer an easy-to-program shared memory model and are ubiquitous in general purpose systems. One of the main challenges of shared memory multiprocessors is scalability. Recent work builds on the observation that there is no need for equal treatment of all memory data accesses. The private and read-only shared data can be treated differently to achieve better scalability and performance [1]–[11].

In order to optimize private memory accesses, efficient and accurate classification of data as private or shared is required. Data can be classified as private if it is accessed by a single core and as shared if it is accessed by more than one core. There are many approaches in the literature regarding data classification performed at different levels: compiler [6], [12], operating system [5], and hardware [13]–[17].

Hardware-based approaches are more accurate as they can gather complete and fine grain information about data sharing. Classification techniques in hardware can be done at the directory level [14]–[16] or at the translation lookaside buffer (TLB) level [17]–[19]. The advantage of the TLB-based classification techniques is that the private/shared nature of the data is readily available at the time of the cache access, as every memory request generated by the processor has to access the TLB to find the translation of the virtual address to the physical address, which enables most optimization techniques [1], [5]. Directory-based classification obtains the private/shared nature only when the directory is accessed and after a cache miss, thus preventing the use of most optimizations.

The main disadvantage of current TLB-based approaches is the detection of private and shared data at page-level granularity, as if a page is considered as shared, all the blocks belonging to the page are also considered as shared. Hence, there is a noticeable loss of classification opportunities, or miss-classification. On the other hand, if classification could be done at a finer granularity, e.g., at block level, this miss-classification of data can be reduced and the optimization techniques will count with more private data.

This work proposes a TLB-based mechanism that operates at block granularity and it can address the missing opportunities of page-grain approaches, thus improving the accuracy of the classification. The proposed classification extends the TLB entries with information about accessed blocks and private blocks within each page in the TLB. An efficient communication protocol is responsible for updating this information at run time. We explore several techniques to reduce the broadcast overhead.

The proposed block-level approaches are evaluated for a 16-core tiled CMP architecture for 15 parallel workloads, and without loss of generality, we employ our classification technique to reduce the directory bookkeeping by not tracking private blocks [5]. The increased 17.0% of private misses which helps to avoid 63.0% entries in the directory caches compared to a directory that tracks all accessed blocks and 43.9% of the entries compared to page-grain approach. A less occupied directory generates fewer invalidation requests, which consequently reduces the 12.6% L1 cache misses. Finally, the overall improvement in execution time is 6.0%.

The outline of the paper is as follows. TLB-based classification mechanisms are analyzed in Section II. Section III describes the proposed block-grain techniques to improve the accuracy of the classification. The simulation environment is detailed in Section IV and performance results are shown in Section V. Section VI covers the related work and Section VII offers the conclusions of this work.

II. Background

This work presents new proposals for improving TLB-based private/shared data classification and it applies them

to the coherence deactivation technique. This section offers a background for the classification and optimization approaches employed in order to make the paper self-content.

A. TLB-based classification techniques

TLB-based classification techniques dynamically detect data as private, read-only, or shared based on the information stored at the TLBs [17], [18]. These techniques rely on querying the other TLBs in the system about their use of the data. The communication among TLBs is done through TLB-to-TLB requests and responses that use the same interconnection network as the memory accesses. On every TLB miss, a broadcast message is sent to all the other TLBs in the system and they reply with information about their usage of the page and, additionally, with the page translation (if they hold it), which accelerates the page table walk process [20]–[22].

Obtaining the page translation for remote TLBs leverages low-latency core-to-core communication of current chip multiprocessors, which is lower than page table accesses. As page table walk happens parallel to TLB-to-TLB communication, address translation can be achieved from the page table if none of the core TLB holds the page. On new access to a memory page by a core, the page may change from private to shared. In this situation, a recovery mechanism has to be triggered, to inform the system about the new nature of the blocks in the page, and perform the appropriate actions depending on the optimization applied, as described in next section.

B. Coherence Deactivation

Directory-based cache coherence is the most scalable alternative to the cache coherence problem. Directory-based protocols employ directory caches that track all memory blocks stored in the private caches, such that they can keep the coherence of such blocks on memory write. In the case of directory cache evictions, invalidation requests are sent to the private caches holding memory blocks for the evicted address, since the directory will not be able to track those blocks anymore. These invalidation requests generate misses known as directory coverage misses [23], which may degrade the system performance.

Cuesta et al. propose to deactivate cache coherence for private blocks [5] and later for read-only blocks [24]. The mechanism consists in storing only those blocks in the directory which need coherence management. The nature of the block must be detected before the cache miss takes place, to deactivate its coherence maintenance. Since coherence deactivation bypasses the coherence protocol, a hardware recovery mechanism is required when a page becomes coherent, to avoid inconsistencies. Blocks that transition from private to shared must be either flushed from the cache (flushing-based recovery) or updated in the directory cache [5]. Once the recovery mechanism is executed, the directory cache is in a coherent state according to the new page classification, and coherence is maintained as established by the cache coherence protocol.

III. TLB-BASED BLOCK-GRAIN CLASSIFICATION

This section describes our proposed fine-grain mechanism to detect the private/shared nature of the memory blocks accessed by each core. The mechanism extends the TLB entries with per-block information which is kept updated using coherence messages exchanged by the TLBs in the system.

A. The concepts

We detect and classify a memory block as private if it is not being accessed currently by any other core. Block is considered as an accessed block if the core has recently accessed the block and virtual-to-physical page translation of that block is still present in the core's TLB. When a core accesses a memory block for which its private/shared nature is not known, the core will query the other cores in the system to know if the block is being accessed by the other cores. If the answer is positive, then the block is classified as shared. Otherwise, the block is classified as private. A block that is classified as private may become shared when another core accesses it in the future. A recovery mechanism ensures that the status of the corresponding block is restored, and after that, both cores can access them as a shared block.

B. Tracking the information

The information about accessed blocks can be stored along with the TLB translation in order to be retrieved when cores ask for the private or shared nature of a block. An *Access* bit vector, where each bit represents one block in the page, stores access information for each block. A bit set to one indicates that the corresponding block has been accessed by this core. A bit set to zero indicates that it has not been accessed. The information about private or shared is also stored in order to avoid queries for every memory access. This information is stored again per block, using a *Private* bit vector. A bit set to one indicates that the corresponding block is private and a bit set to zero indicates that the block is shared.

VPN	PPN	Access Bit Vector	Private Bit vector.
		0 | 0 | 0 | 0 | 0|0	1 |1 |1 | 1 | 1 |1 | 1

Fig. 1. TLB entry with extra fields

Figure 1 shows the structure of the TLB entries with its most relevant fields: the Virtual Page Number (VPN) and the Physical Page Number (PPN), and the two bit vectors with access and private information for the blocks belonging to the corresponding page. When a block has been accessed the information about the Private bit gives the classication of the block. However, when a block has not been accessed, the Private bit is used to provide information on future accesses to avoid extra queries. Table I shows the meaning of the different combinations of the Access bit and the Private bit. In the remainder of the document, we will represent these two values as a pair (Access, Private). When the block is found in the state (1,*), the block has already been classified and no extra actions are necessary. When the block is in the state (0,1) it is known

that the core is the only one in the system with this status for the block the status of the same block in other cores is in (0,0). Therefore, (0,1) can silently transition to (1,1) without communicating with other TLBs. Only in the case the block is in (0,0) state, or the page is not found in the TLB, a query to the other TLBs in the system has to be performed.

TABLE I
ACCESS AND PRIVATE INFORMATION AND MEANING

Access	Private	Meaning
1	1	The block has been accessed by the core. The block is private. Only one core can have this block in this state.
1	0	The block has been accessed by the core. The block is shared.
0	1	The block has not been accessed by the core. No other core has accessed it and the core has permission to access it in private mode without needing to ask. Only one core can have this block in this state.
0	0	The block has not been accessed by the core. On access, it is necessary to check if the other core is accessing it as private.

C. The classification protocol

The TLB-to-TLB communication protocol proposed to classify blocks into private and shared uses the same interconnection network as the memory accesses. The next sections detail how the protocol is initiated, how other TLBs reply to the request, and how the private/shared information is gathered.

1) Issuing TLB-to-TLB requests: TLB-to-TLB communication protocol is initiated in two situations: i) there is a TLB miss or ii) there is a TLB classification miss [case (0,0)]. The communication protocol is initiated by broadcasting TLB-to-TLB requests to all other TLBs in the system. The TLB-to-TLB request carries the virtual address of the block (which includes the address of the page) that generated the TLB-to-TLB request and the type of request (translation miss or classification miss). This information helps to discover when to transition to shared and when to stay private. The access to the block is stalled until the classification for the block is determined, which happens when all responses from other TLBs are collected.

2) Receiving a remote TLB-to-TLB request: The TLB-to-TLB request is received by each of the TLBs in the system. The TLBs receiving the request have to perform two actions: 1) Issue a TLB response indicating if they use the requested block (the response also carries the address translation if found in order to accelerate the virtual-to-physical translation [17], [20]) and 2) Update the *Private* bit of the requested block. In the case of a TLB request due to a TLB miss, the requester TLB does not have private/shared information about any of the blocks belonging to the page. In this case, the information about the use of blocks is sent for all blocks in the page in a bit vector field along with the response message. This provides a preliminary classification of blocks that have not been accessed yet (see Table I). If the TLB is not currently issuing a TLB request, the information sent in the reply corresponds to the *Access* bit vector in absence of the TLB entry, or a *Nack* in presence of the TLB entry. If a TLB request has been initiated, then the block that initiated the request is set to one in the bit vector, since the core is currently accessing the block.

TABLE II
RECEIVING REMOTE TLB-TO-TLB REQUESTS

State		TLB requests			Send	New State
A	P	L_P	L_B	R_B	Use	P
-	-	0	*	*	0	-
-	-	1	0	*	0	0 when L_P resolves
-	-	1	1	0	1	1 when L_P resolves
-	-	1	1	1	1	0 when L_P resolves
0	*	*	0	*	0	0
0	*	1	1	0	1	1 when L_P resolves
0	*	1	1	1	1	0
1	0	*	*	*	1	0
1	1	*	*	0	1	1
1	1	*	*	1	1	0 (Recovery)

There are two main cases to consider regarding the update of the *Private* bit vector depending on the TLB has initiated a TLB request or not. If the TLB has not initiated a TLB request (and there is an entry for the requested page in the TLB), all blocks that have not been accessed are set to potentially shared (0,0), thus offering private permission to the requester TLB. The blocks that have been accessed do not modify the *Private* bit, except for the requested block. If the requested block is in the state (1,1), then it is considered private, but another TLB is requesting it. In this case, the block transitions to (1,0), that is, shared, and a recovery mechanism (see Section III-E) is initiated. Until the recovery mechanism does not finish, the TLB response cannot be sent, to avoid race conditions (when a TLB is considering a block as private and another TLB is considering it as shared). When the TLB has initiated a request for the page, it is important to know the block that generated the request, since if a remote request is received for this block, then the block has to be set as shared, even if all cores respond that they are not using this block. Table II summarizes the previous behavior for each block on the page. The leftmost information (state and requests) refers to the information known when receiving a remote TLB request and the information on the right (send) represents the information sent back in the reply and how the *Private* bit vector transitions. The first two columns represent the *Access* bit of a block in the TLB (A) and the *Private* bit of a block in the TLB (P). A dash (-) represents that the block is not found in the TLB and a star (*) indicates that the value of that field is not relevant. The next three columns represent if the local TLB has initiated a TLB-to-TLB request for the page in question (L_P), if that local request is for the block in question (L_B) and if the remote request is for the block in question (R_B). The next column represents what it is sent in the reply regarding that block and the last column represents the value for the *Private* bit after the request is processed.

3) Collecting all TLB responses: After receiving all TLB responses, the private/shared nature of the block is calculated. An OR operation of all bit vectors received is performed to find the use by remote cores, and then a NOT operation will activate the not used blocks. The result is stored in the *Private* bit vector. Note that if there is a concurrent TLB request in this node, the information of the *Private* bit vector is updated

as indicated in the last column of Table II.

D. Protocol state transitions

This section explains the different states and transitions for a block, depicted in Figure 2. Blocks start in state Not Present (-,-) when their page has not been requested yet by the core and are reset to that state when the page is evicted from the TLB. If a core accesses the block then, a TLB miss happens and a request is sent to the other TLBs. The accessed block transitions to state Requested (1,-) while the other blocks belonging to the same page move to the Requested state (0,-). After receiving all TLB responses, the accessed block (1,-) transitions either to private (1,1) or shared (1,0), while other blocks belonging to the same page move to potentially shared (0,0) or potentially private (0,1). If while performing the TLB miss in state Requested (1,-), a remote request for the accessed block is received, the state changes to Local&Remote request (1,-) forcing the block to be shared (1,0) after collecting all replies. This way it guarantees coherence between TLBs when more than one TLBs request happens at the same time. The invariant here is that a block can be in (1,1) state in only one TLB. A similar situation happens for blocks that are not the accessed ones. If in Requested (0,-) state, a remote request for that block comes, the state will be (0,0) when all replies are collected. The invariant here is that a block can be in (0,1) state in only one TLB. When local access for a block in the state (0,1) is received, the block is silently transitioned to state private (1,1). The reason is that this is the only TLB in the system holding potential privacy for the block (0,1). On the other hand, if on (0,1) a remote request is received, the state changes to potentially shared (0,0) without the need of doing recovery since the block has not been accessed yet. If a block in the state (0,0) is locally accessed by the core, a TLB-to-TLB request has to be issued because of the TLB classification miss. The request informs the other TLBs about the access of the block and gets the actual classification for the block. The state changes temporally to Requested (1,-). It will receive only *Acks* —if used— or *Nacks* —if not used— from other TLBs, since only one block is requested in a TLB classification miss. Once it receives all replies it will become shared (1,0) if any *Ack* is received or private (1,1) if only *Nacks* are received.

E. Recovery mechanism and TLB evictions

Blocks classified as private (1,1) are not tracked by the directory cache, since they do not require coherence maintenance while they are private. Thus, cache misses for private blocks override the coherence protocol. When a private block (1,1) changes to a shared state (1,0), a recovery action is needed to ensure that the directory cache keeps the proper track of the block. We opt for evicting the private block from the cache and writing it back if the block is dirty, such that the directory information is consistent and that the last modifications on the block are not lost. Once this recovery finishes, the reply will be sent to the requester TLB, to classify the block as shared. It is important to note that recovery mechanism of block grain is much simpler and faster compared to a page-grain approach as block grain recovery evict the requested block only and page grain evicts all blocks in a page. This is shown in Section V.

In the case a TLB entry is evicted or flushed due to either lack of TLB capacity or a TLB shootdown, the information about the private or shared of a block is lost. Furthermore, the core will be considered as not accessing currently any block within the evicted page, and the blocks in the page may be classified as private from that point on by another core. Therefore, in order to keep the coherence of the data, all blocks in the evicted page have to be also evicted from cache. This is an iterative process that only affects the blocks previously accessed by the core (1,*). This process also takes more time than in a page-grain approach, since a page-grain approach requires the lookup and eviction of each block in the page as the access information is not present.

F. Reducing TLB classification misses

The disadvantage of a block-grain classification with respect to a page-grain classification is the extra TLB traffic generated due to TLB classification misses. In order to reduce this extra traffic we propose the following two optimizations that exploit spatial locality and make a better use of the available information.

1) Spatial locality optimization: The previously explained classification protocol does not consider spatial locality in the accesses. If the block is found in state (0,0) a TLB classification miss happens and TLB requests are issued. In order to minimize TLB classification misses, we propose an optimization that takes advantage of the spatial locality. On the receipt of a remote TLB request, the TLB does not give away all potentially private states for the block in the requested page. The requester gets potentially private blocks ranging from the requested one to the first block already accessed by the local core (not included) in ascending address order. It is a high probability of accessing a block after accessing the previous one by a core (spatial locality). This way, blocks marked as potentially private (0,1) by a TLB that are not expected to be accessed by the requester core are kept as potentially private.

2) Access permission prefetch optimization: The key idea behind this optimization is to avoid having blocks in (0,0) state when they will be classified as shared. In this case, it is preferable to have the block in (1,0) state, even if it has not been accessed before. When the core issues the access for the block, then the TLB classification miss is avoided, and the access continues as shared. The previous approaches only send a bit vector indicating the use of the block in the TLB response, and the block is classified as private or shared according to this vector. The *Access* bit vector is not modified by the TLB responses. This optimization proposes to send the *Use* bit vector along with the *Access* bit vector (the one sent in the basic block-grain approach). The *Use* bit vector is employed to update the *Private* bit vector, as previously described, with an OR and a NOT operation. The *Access* bit vector will be also ORed with the vectors collected from all responses. If a block is classified as shared, and we know that

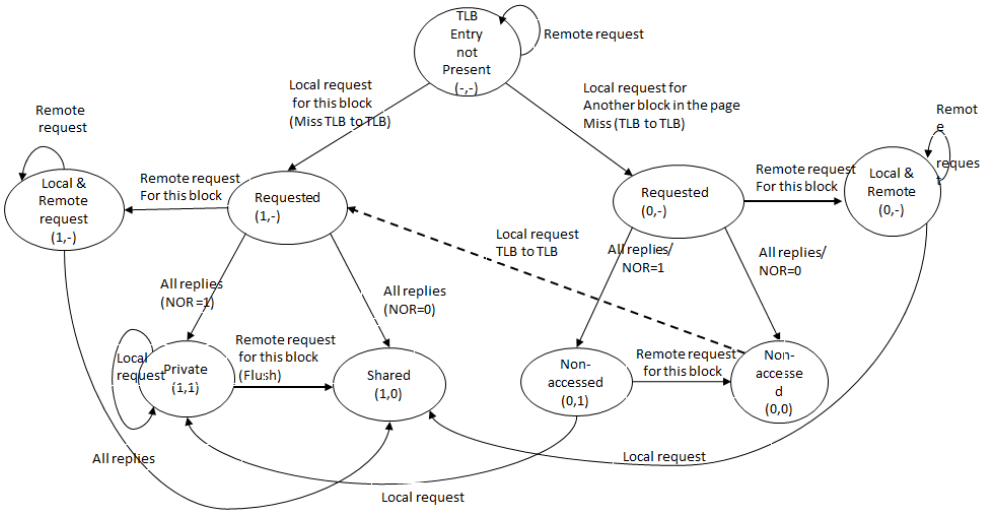

Fig. 2. Protocol state transition diagram

TABLE III
BASELINE SYSTEM CONFIGURATION

Memory configuration	
Processor	2.20GHz, 16-core in order CPU
Cache hierarchy	Non-inclusive
Split instr. and 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 per core	512 sets, 4 ways, 1 cycle
Memory access time	160 cycles
Split instr. and data TLB	128 sets, 4 ways, 1 cycle
Page size	4KB (64 blocks)
Network configuration	
Topology	2-dimensional mesh (4x4)
Flit size	16 bytes
Routing technique	Deterministic X-Y
Routing, switch, and link time	2, 2, and 2 cycles
Data and control message size	5 flits and 1 flit

any of the remote cores are using it (the OR result is 1 for that block), then we set the *Access* bit for that block to 1. The state will be (1,0), and future access will not generate a TLB classification miss (0,0).

G. Dealing with synonyms

Our mechanism works at the TLB level where the information about the page translation and other permission bits for the page reside. Synonyms, that is, two different virtual addresses pointing to the same physical page, could be miss-classified as private by the described mechanism. However, the operating system is aware of synonyms and solutions to this problem have been already proposed [25] when using an operating-system classification. The operating system can be aware of synonyms. Memory pages containing synonyms are therefore treated as shared by default. When a page containing synonyms is accessed for the first time, the TLB vectors are filled with the status (1,0) and it will remain with that status until page eviction.

IV. Simulation Environment

The proposed fine-grained block approaches are implemented with a full-system simulation using Virtutech Simics [26] and Wisconsin GEMS toolset [27]. The simulated architecture is a 16-tile CMP architecture with directory-based cache coherence. Table III shows the configuration for the simulated system. The proposed schemes are evaluated with 15 parallel workloads from the SPLASH-2 [28] and PARSEC [29] benchmark suites. Barnes (8192 bodies,4 time steps), Cholesky (tk15.O), FFT (64K complex doubles), FMM (16K particles), LU (512512 matrix), LU-NC (512x512 matrix), Ocean (258x258, contiguous partitions), Radiosity (room, -ae 5000.0-en 0.050 -bf 0.10), Radix (524288 integers), Volrend (head), and Water-Sp (512 molecules) belong to SPLASH. Blackscholes (simmedium), Fluidanimate (simsmall), Swaptions (simsmall) and x264 (simsmall) are from PARSEC.

V. Results

This section shows a quantitative comparison of our block-grain classification approach, comparing it to a TLB-based classification approach that works on the page level (Page) and to a baseline system that does not employ any classification (Base). For the block-grain approaches we show the effect of each of the optimizations: *Block* represents the basic approach without optimization, *Block+SL* includes the spatial locality optimization, and *Block+SL+APP* adds on top the access permission prefetch optimization. In particular, we show how our proposals classify more accesses as private, and how the overhead of the block-based classification is kept low thanks to our optimizations. Then, we evaluate the impact of the classification when the coherence deactivation approach is employed. The increased number of private misses translates into fewer blocks tracked by the directory caches. A less occupied directory generates fewer invalidation requests, which consequently reduces the L1 cache misses. Finally, we show the overall improvement in execution time.

A. Private accesses

The main aim of the block-grain proposal is to detect more private blocks by removing the miss-classification of blocks in page-grain approaches. Figure 3 shows the percentage of

L1 accesses for private blocks (hits and misses) in a page-grain scheme and the proposed block-grain schemes: Block, Block+SL and Block+SL+APP. Results show that Page, Block, Block+SL, and Block+SL+APP detected, on average, 22.4%, 31.1%, 33.1% and 33.3% private L1 hits respectively and 0.1%, 0.2%, 0.3% and 0.3% private L1 misses, respectively. Our block-grain approaches remove the miss-classification of page level approaches and access on average 2.5 times more private blocks that miss in L1 than the page-grain approach. The Block, Block+SL, and Block+SL+APP accessed, on average 10.0%, 17.0% and 16.0% more private miss blocks than the page level scheme. This more accurate classification of private blocks helps to reduce the amount of records used in the directory cache.

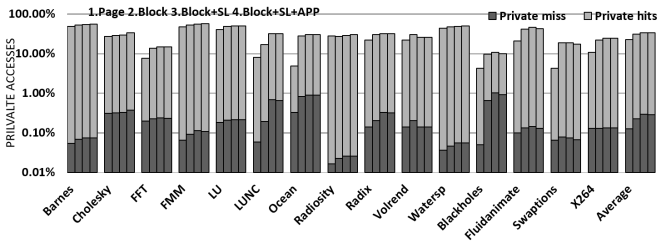

Fig. 3. Private hit and private miss accesses in percentage

B. TLB traffic overhead

Our block-grain approach increases the number of TLB requests issued with respect to a page-grain approach, as when the block state is potentially shared (0,0) a TLB classification miss happens. We analyze in this section the overhead of the TLB classification misses over the TLB misses that happen in our baseline configuration when the page translation is not found. Figure 4 shows this overhead, differentiating the number of TLB-to-TLB requests because of TLB misses (normalized with respect to Page) and because of TLB classification misses. On average, the overhead of the number of TLB requests in Block, Block+SL, and Block+SL+APP is 3.1%, 2.9%, and 1.1% respectively compared to the page-grain approach. Block+SL+APP reduces the number of TLB requests due to TLB classification misses thanks to the access permission prefetch optimization. The overall overhead in TLB requests is not as critical as the time resolution for this request is lower in block-grain approaches as we discuss in the next section. LU and Cholesky are the applications showing more TLB request overhead.

C. Coherence recovery latency

An advantage of the block-grain approaches is that the recovery mechanism entails the eviction of a single block rather than all blocks belonging to the page, as happens in the page-grain approach. Block-grain approaches have a shorter recovery mechanism and hence require less latency to resolve TLB misses compared to page-grain approach. This low TLB miss latency results in reductions in execution time. Figure 5 shows the latency of the recovery mechanism both in the page-grain approach and in the block-grain approaches. The

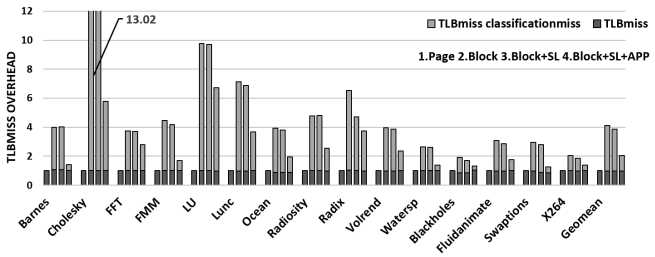

Fig. 4. TLB miss broadcast overhead in Block grain

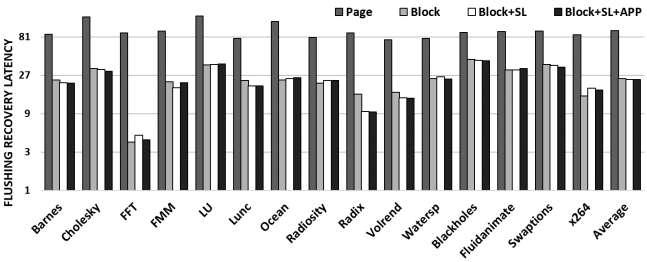

Fig. 5. The latency of the recovery mechanism in clock cycles

page requires on average 97.7 clock cycles for the recovery operation. On the other hand, block-grain approaches require considerably less time: 24.8 cycles on average for Block, 24.1 cycles on average for Block+SL and 23.9 cycles on average for Block+SL+APP.

D. Average directory entries required

The main metric to measure the benefit of our block-grain approaches on the coherence deactivation technique is the number of directory records required to keep track of the cached blocks. Figure 6 shows the normalized number of directory entries required with respect to Base, where all cached blocks are tracked by the directory. When employing our fine-grain approach the required directory entries fall dramatically. Block avoids the storage of 51.0% of the entries, Block+SL avoids the storage of 51.9% of the entries and Block+SL+APP avoids the storage of 63.1% of the entries. Additionally, when compared to the page-grain scheme, our block-grain schemes reduce the number of required entries by 25.3%, 26.7%, and 43.9%, respectively, for the Block, Block+SL, and Block+SL+APP. Blackscholes just require 5.0% of the entries in the directory cache as being a highly scalable benchmark it has more private blocks. Ocean requires only 19.0% entries in directory compared to the page-grain approach.

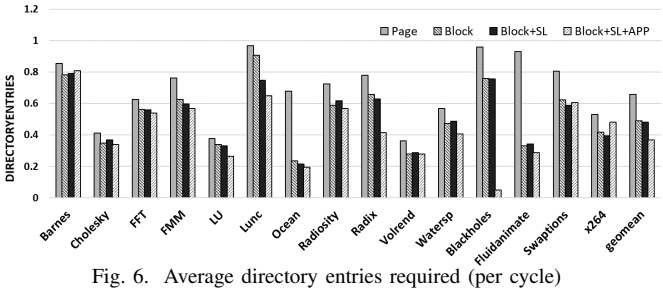

Fig. 6. Average directory entries required (per cycle)

E. L1 cache misses

Thanks to the reduction in the directory cache requirements which cause fewer evictions in the directory cache and less coverage misses, and thanks to having less flushed blocks from the L1 cache, a reduction in L1 cache misses is found in block-grain schemes. Figure 7 shows the L1 cache miss ratio normalized with reference to Base. Cache misses classify in 3C (Compulsory, Conflict, Capacity), Coherence miss (because of invalidation due to other core write), Coverage miss (because of invalidation due to eviction from the directory cache), and Flushing miss (because of the recovery mechanism or TLB evictions). Block reduces L1 cache-miss of 54.9% of the entries, Block+SL avoids the L1 cache-miss of 57.9% of the entries and Block+SL+APP reduce L1cache-miss of the 58.0%. Additionally, when compared to Page, our block-grain schemes reduce the L1 cache-miss by 9.4%, 12.5%, and 12.6%, respectively, for the Block, Block+SL, and Block+SL+APP. Page has on average 19.0% of coverage misses compared to the total amount of misses while Block has on average 6.7% of coverage misses compared to the total amount of misses of the page-grain approach. x264 does not reduce the cache miss rate as it does not increase the number of classified private-miss accesses.

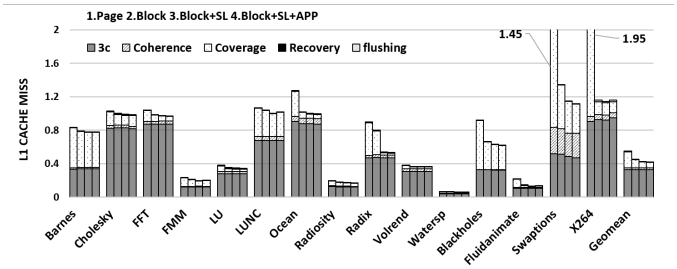

Fig. 7. Normalized L1 cache miss rate with respect to baseline

F. Normalized network traffic

Figure 8 shows the overall impact on network traffic after the TLB communication overhead and the reduction in messages happens because of fewer invalidation messages from the directory cache and a lower L1 cache miss ratio. Results are normalized with respect to Base, and each bar differentiates the traffic based on cache request, cache response control, cache response data, TLB request, TLB response control, and TLB response data. Block, Block+SL, and Block+SL+APP reduce the network traffic by 33.0%, 38.0%, and 48.0% respectively compared to the baseline setup. Block and Block+SL increases network traffic 7.5%, 2.6% compared to Page as having more TLB communication overhead. Block+SL+APP reduces 7% network traffic compared to Page with the help of reduction in TLB broadcast. Watersp reduces around 92.0% traffic as it has a 95.0% reduction in L1 cache misses with respect to the baseline cache protocol. Cholesky increases traffic 3.5% as it has a number of TLB broadcast overhead.

G. Execution time

Given all the previous analysis (lower L1 cache miss rate, a larger number of private misses, less network traffic, and less

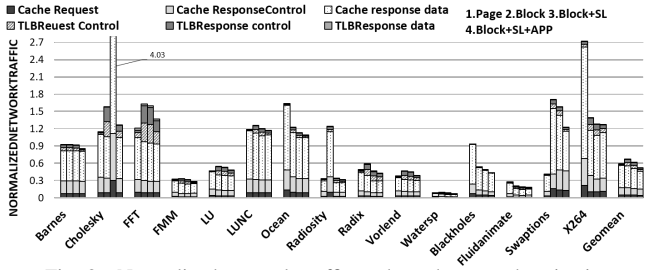

Fig. 8. Normalized network traffic under coherence deactivation

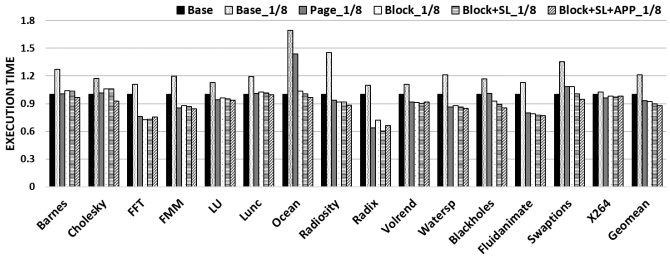

Fig. 9. Execution time for coherence deactivation

directory occupancy) we can expect reductions in execution time with the block-grain approaches. Figure 9 shows the execution time for Base (first bar) with 100% coverage directory, Base_1/8 (8 times less entries in the directory - second bar), Page_1/8 (third bar), Block_1/8 (fourth bar), Block+SL_1/8 (fifth bar), and Block+SL+APP_1/8 (sixth bar) normalized with respect to Base, which does not detect private blocks.The directory cache employed for both Page and Block in this study has been reduced to 1/8 of its original size, in order to stress the advantages of the classification approaches. Block, Block+SL and Block+SL+APP reduce the execution time by 7.8%, 10.3% and, 12.2% considering 8 times smaller directory cache when compared to Base. Furthermore, when compared to Page, they reduce the execution time by 1.4%, 4.0%, and 6.0%, respectively.

H. Memory overhead

Our block-grain approaches trade off non-scalable directory entries with scalable TLB bit vectors. Therefore, as the amount of cores grows in the system, more memory will be saved compared to a baseline approach. For the configuration employed in this work, the memory pages (4KB) and memory blocks (64B). Since, each TLB entry includes two bit vectors and there are a total of TLB 1K entries per core, the overhead is 16KB per core. On the other hand, block-grain techniques allow us to reduce the directory size. The directory is a non-scalable structure, as many implementations employ a bit vector that grows linearly with the number of cores. A directory with the same number of entries as the private caches in this work requires 2K entries per core. Each entry accounts for a tag, state bits, and the non-scalable bit vector (6 bytes). The directory requires 12KB. We can improve performance with one-eighth of this size, that is 10.5KB storage savings. As the number of cores increases, the directory size increases too. For example, a similar directory for just 64 cores requires 4 (tag) + 8 (bit vector) bytes per entry, that is a directory of 24KB. Reducing it to one eighth would mean to save overall storage

requirements of 21KB compared to the baseline configuration. Most of the operating system have a standard page size of 4KB as it provides more granular control. However, in some cases large pages can be supported. Large pages would require larger bit vectors in the TLB. To compensate this effect, or even to reduce further the size of the TLB bit vectors, the implementation could use one bit to represent a group of consecutive blocks (coarse grain representation). For example, using 64 bits in the bit vector and considering 64KB pages (1024 blocks), each bit would represent 16 blocks that would be classified as private or shared all together.

VI. Related work

There have been recently several research works aimed at classifying data between private and shared to employ the classification for optimizing cache coherence protocols [30]. Classification of data can be done at either fine grain (cache block or even data access) or coarse grain (memory page), and it should be as adaptive as possible [31].

A. Fine-grain approaches

Fine grain approaches include compiler- and directory-based approaches. In a compiler-based approach [6], [12] it is hard to know at the time of compilation what will be the sharing status of a variable at run time, mostly if the status is detected in a certain period of time. Our fine-grain proposal works at run time which gives a more accurate classification.

Directory-based techniques work at block level but the classification is detected after a cache miss, disabling the use of coherence deactivation techniques. In SWEL [14] the L1 cache stores non-coherent blocks and the L2 cache stores coherent blocks. POPS [32] optimizes the protocol by combining private and shared memory blocks on various L2 cache slices in the NUCA architecture. Valls et al. [33] designed a two-level directory where the first level is small and stores shared data cache and the second level is large and stores private data. The reason behind this is that most hits occur for shared records. Multigrain [16] is a directory-based mechanism that allocates a single record temporarily to a private region rather than allocating a record for each private block. It adjusts the area of the region at run time, thus saving directory storage. Our block-grain approach can support coherence deactivation and any other optimization technique that requires the classification to be known before the cache miss. Additionally, it does not entails any modification to the directory structure.

B. Coarse-grain approaches

Coarse grain approaches perform a classification of memory pages with the help of the page table and/or the TLB [5], [9], [17], [34]. Cuesta et al. [5] uses the page table to detect the nature of the memory pages and deactivate coherence for blocks in private pages. Ros et al. [17] improve the classification accuracy thanks to TLB-to-TLB communication and a late discovery of TLB evictions. TokenTLB [18] is a token-based coherence mechanism [35], that reduces TLB communication. With a predictions of the use of pages [19] the accuracy of classification can be improved, as TLB evictions can be detected earlier. The forced-sharing predictor [34] helps to reduce extra classification traffic introduced when a block is private to different cores for a small period of time. Some of these optimization are orthogonal to our work, but we increase classification accuracy thanks to a fine-grain classification.

Recently, Caheny et al. [10] have proposed a hardware-software co-design proposal using a parallel programming model to deactivate the cache coherence protocol, which helps to reduce the directory area and energy consumption with the help of extra hardware. Studies at sub-page granularity have also been performed recently [36], by using an on-chip page table to find address translation. Our proposals on the other hand obtain the accuracy of block-grain classification.

Finally, a real time prototype of private/shared data classification has been recently implemented on LEON SPARC multiprocessor [37] using a page-grain approach.

VII. Conclusions and future work

Categorizing memory accesses into private and shared helps to achieve scalability and efficiency in a multicore system. We present a novel approach to categorize data as private or shared that works at a finer granularity than existing approaches using the TLB structures. The proposed scheme detects, on average, more accessed private miss blocks 17.0% than previous approaches and results in performance improvement of 6.0% compared to a page-grain approach. Additionally, our block-grain approach trades off non-scalable directory entries for scalable bit vectors at the TLB. Our future work aims to eliminate the non-scalable directory structure using only scalable TLB bit vectors.

ACKNOWLEDGMENTS

This work was supported by the Spanish MCIU and AEI, as well as European Commission FEDER funds, under the grant "RTI2018-098156-B-C53".

References

[1] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive nuca: near-optimal block placement and replication in distributed caches," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, 2009, pp. 184–195.

[2] B. R. Upadhyay and T. Sudarshan, "Task-enabled instruction cache partitioning scheme for embedded system," in *First International Conference on Smart System, Innovations and Computing*, 2018, pp. 603–612.

[3] D. Kim, J. Ahn, J. Kim, and J. Huh, "Subspace snooping: Filtering snoops with operating system support," in *19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 111–122.

[4] D. Kim, H. Kim, and J. Huh, "Virtual snooping: Filtering snoops in virtualized multi-cores," in *43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 459–470.

[5] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *38th International Symposium on Computer Architecture (ISCA)*, 2011, pp. 93–103.

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

[7] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 241–252.

[8] S. Shukla and M. Chaudhuri, "Tiny directory: Efficient shared memory in many-core systems with ultra-low-overhead coherence tracking," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 205–216.

[9] A. Esteve, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Tlb-based temporality-aware classification in cmps with multilevel tlbs," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 8, pp. 2401–2413, 2017.

[10] P. Caheny, L. Alvarez, M. Valero, M. Moretó, and M. Casas, "Runtime-assisted cache coherence deactivation in task parallel programs," in *International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2018, p. 35.

[11] J. Dumas, E. Guthmullcr, and F. Petrotl, "Dynamic coherent cluster: A scalable sharing set management approach," in *29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2018, pp. 1–8.

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

[13] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "Improving multiprocessor performance with coarse-grain coherence tracking," in *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2, 2005, pp. 246–257.

[14] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian, "Swel: Hardware cache coherence protocols to map shared data onto shared caches," in *19th international conference on Parallel architectures and compilation techniques*, 2010, pp. 465–476.

[15] H. Zhao, A. Shriraman, S. Dwarkadas, and V. Srinivasan, "Spatl: Honey, i shrunk the coherence directory," in *International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 33–44.

[16] J. Zebchuk, B. Falsafi, and A. Moshovos, "Multi-grain coherence directories," in *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 359–370.

[17] A. Ros, B. Cuesta, M. E. Gómez, A. Robles, and J. Duato, "Temporal-aware mechanism to detect private data in chip multiprocessors," in *42nd International Conference on Parallel Processing (ICPP)*, 2013, pp. 562–571.

[18] A. Esteve, A. Ros, A. Robles, M. E. Gómez, and J. Duato, "Tokentlb: A token-based page classification approach," in *International Conference on Supercomputing (ICS)*, 2016, pp. 26:1–26:13.

[19] A. Esteve, A. Ros, A. Robles, and M. E. Gómez, "Tokentlb+cup: A token-based page classification with cooperative usage prediction," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 29, no. 5, 2018.

[20] 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 International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[21] S. Srikantaiah and M. Kandemir, "Synergistic tlbs for high performance address translation in chip multiprocessors," in *43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 313–324.

[22] D. Lustig, A. Bhattacharjee, and M. Martonosi, "Tlb improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level tlbs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 1, p. 2, 2013.

[23] A. Ros, B. Cuesta, R. Fernández-Pascual, M. E. Gomez, M. E. Acacio, A. Robles, J. M. García, and J. Duato, "Emc 2: Extending magny-cours coherence for large-scale servers," in *International Conference on High Performance Computing*, 2010, pp. 1–10.

[24] B. Cuesta, A. Ros, M. E. Gomez, A. Robles, and J. Duato, "Increasing the effectiveness of directory caches by avoiding the tracking of non-coherent memory blocks," *IEEE Transactions on Computers*, vol. 62, no. 3, pp. 482–495, 2013.

[25] S. Kaxiras and A. Ros, "A new perspective for efficient virtual-cache coherence," in *40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 535–547.

[26] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[27] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," *ACM SIGARCH computer architecture news*, vol. 23, no. 2, pp. 24–36, 1995.

[29] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[30] N. Parvathy, B. R. Upadhyay, and T. Sudarshan, "Cache coherence: A walkthrough of mechanisms and challenges," in *International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, 2016, pp. 2251–2256.

[31] M. Davari, A. Ros, E. Hagersten, and S. Kaxiras, "The effects of granularity and adaptivity on private/shared classification for coherence," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 3, p. 26, 2015.

[32] H. Hossain, S. Dwarkadas, and M. C. Huang, "Pops: Coherence protocol optimization for both private and shared data," in *International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 45–55.

[33] J. J. Valls, A. Ros, J. Sahuquillo, M. E. Gómez, and J. Duato, "Ps-dir: a scalable two-level directory cache," in *21st international conference on Parallel architectures and compilation techniques*, 2012, pp. 451–452.

[34] 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)*, vol. 27, no. 3, pp. 748–761, 2016.

[35] 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 International Symposium on High-Performance Computer Architecture*, 2005, pp. 328–339.

[36] M. Soltaniyeh, I. Kadayif, and O. Ozturk, "Classifying data blocks at subpage granularity with an on-chip page table to improve coherence in tiled cmps," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 4, pp. 806–819, 2018.

[37] N. Ho, I. I. Ashraf, P. Kaufmann, and M. Platzner, "Accurate private/shared classification of memory accesses: a run-time analysis system for the leon3 multi-core processor," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017, pp. 788–793.