Efficient Classification of Private Memory Blocks

Bhargavi R. Upadhyay[a], Alberto Ros[b], Jalpa Shah[c]

[a]*Department of Computer Science and Engineering, Amrita School of Engineering, Bengaluru, Amrita Vishwa Vidyapeetham, INDIA*
[b]*Computer Engineering Department, University of Murcia, Murcia, SPAIN*
[c]*Department of Electronics and Communication Engineering, Amrita School of Engineering, Bengaluru, Amrita Vishwa Vidyapeetham, INDIA*

Abstract

Shared memory architectures are pervasive in the multicore technology era. Still, sequential and parallel applications use most of the data as private in a multicore system. Recent proposals using this observation and driven by a classification of private/shared memory data can reduce the coherence directory area or the memory access latency. The effectiveness of these proposals depends on the accuracy of the classification. The existing proposals perform the private/shared classification at page granularity, leading to a miss-classification and reducing the number of detected private memory blocks.

We propose a mechanism able to accurately classify memory blocks using the existing translation lookaside buffers (TLB), which increases the effectiveness of proposals relying on a private/shared classification. Our experimental results show that the proposed scheme reduces L1 cache misses by 25% compared to a page-grain classification approach, which translates into an improvement in system performance by 8.0% with respect to a page-grain approach.

Keywords: chip multiprocessor, cache coherence, private-shared data classification

*Corresponding author
Email addresses: u_bhargavi@blr.amrita.edu (Bhargavi R. Upadhyay), aros@ditec.um.es (Alberto Ros), j_shah@blr.amrita.edu (Jalpa Shah)

1. Introduction

Shared memory architectures are pervasive in general-purpose systems. The key reason is their easy-to-program memory model in which communication happens through load and store operations to a shared address space. A cache coherence protocol implemented in hardware is responsible for moving the copies of the data across cores and keeping them coherent.

The two main approaches to implement cache coherence in hardware are snooping and directory protocols [1]. The advantage of snooping protocols is that they do not require to keep track of the copies cached in the private caches. However, they can only scale up to a limited number of cores. The preferred alternative for medium- or large- scale system are directory protocols. However, they require to keep track of the copies cached in the private caches in a *directory* structure.

Recent studies have shown no need for equal treatment of all memory accesses managed by the cache coherence protocol. Private and read-only shared data does not require indeed a cache coherence mechanism. They can be treated differently to achieve better scalability and performance [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14].

Efficient and accurate classification of data as either private or shared is essential to optimize private memory access. A memory block can be classified as private if a single core accesses it and as shared if it is accessed by more than one core. There are many approaches in the recent literature regarding data classification, which differ in the level at which they are performed: compiler [15, 7], operating system [6], and hardware [16, 17, 18, 19, 20]

Compiler-assisted approaches have the limitation of knowing at compile time if a variable is going to be accessed or not, by which core will be accessed, or even if two variables have the same address (alias). On the other hand, operating system schemes are forced to work at page granularity. As a consequence, these alternatives have accuracy limitations.

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 [17, 18, 19] or at the Translation Lookaside Buffer (TLB) level [20, 21, 22]. The advantage of the TLB-based classification techniques is that the memory block's private/shared nature is readily available at the cache access. Every memory request generated by the processor has to access the TLB to find the virtual address's translation to the physical address. Therefore, more optimizations are possible. Directory-based classification obtains the private/shared nature only when the directory is accessed and after a cache miss. Thus, in this work, we focus on TLB-based approaches.

The main disadvantage of current TLB-based methods is the detection of private and shared data at page-level granularity. If a page is classified 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, and therefore, performing better.

This work proposes a TLB-based classification mechanism that operates at memory block granularity. It can address the missing opportunities of page-grain approaches, improving the classification accuracy and, ultimately, performance. Our classification technique adds to each TLB entry information about the accessed and private memory blocks of the corresponding memory page. An efficient communication protocol is responsible for updating that fine-grain information at run time. We explore several techniques to reduce the classification overhead of a block-based classification, including novel proposals that extend our previous conference publication [13]:

- Basic approach: When a TLB miss takes place, a broadcast message is sent to the TLBs of the other cores in a multicore to collect information about the memory blocks accessed by other cores and decide on the private/shared status for each block in the missing page. Each TLB offers

as private any memory block not accessed by them. In addition, the page address translation is provided to the requester if a TLB holds the translation. This accelerates the TLB miss resolution compared to a page table walk.

- Spatial locality optimization: Consecutive blocks to the requested one are offered as private if they have not been accessed. Blocks after the first block accessed are never offered as private, as even if not accessed, spatial locality predicts that they will be accessed by the current core.

- Access permission prefetch optimization: Blocks accessed by more than two cores are collected and marked as shared and accessed (even if the core does not access it). When the core accesses the memory block, the TLB already identifies it as shared, saving extra traffic that otherwise would be generated. This mechanism has been refined with respect to our previous conference proposal, reducing its traffic requirements.

- Opportunistic data transfer optimization: Responses issued by the TLBs also carry the data block of the address that generated the TLB miss. This will save a subsequent cache miss in the requesting core, thus improving performance. This is the main contribution of this work over the previous conference publication and improves further the advantages of fine-grain TLB classification.

In all our designs, the information about the privacy of the block is obtained before the L1 cache miss occurs, i.e., when accessing the TLB. Therefore, we can apply, without loss of generality, Coherence Deactivation [6], a technique to reduce the directory bookkeeping by not tracking private blocks, to our classification mechanism.

We evaluate the proposed block-level approaches assuming a 16-core tiled CMP architecture with a coherence deactivation mechanism. The results are obtained for 15 parallel applications and show that block-grain techniques detect 18% more private miss blocks than page-grain system. This helps avoid 63.1%

of the entries in the coherence directory compared to a directory that tracks all accessed blocks and 43.8% of the entries than techniques that employ a page-grain approach. This results in an overall improvement in execution time by 8% compared to a page grain approach and 13.9% compared to a baseline approach.

The outline of the paper is as follows. A background on TLB-based classification mechanisms is provided in Section 2. Section 3 describes our block-grain classification techniques proposed to improve the accuracy of the classification. The simulation environment is detailed in Section 4, and performance results are shown in Section 5. Section 6 covers the related work, and finally, Section 7 offers the conclusions of this work.

2. Background

This work presents new proposals for improving TLB-based private/shared data classification, and it applies them to the coherence deactivation technique [6]. This section offers a background for the classification and optimization approaches employed to make the paper self-content.

2.1. 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 [20, 21]. 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 [23, 24, 25]. 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. The page may change from private to shared on new access to a memory page by a core. 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. This is described in the next section.

2.2. Coherence Deactivation

Directory-based cache coherence is the most scalable alternative to the cache coherence problem, as it significantly reduces traffic requirements with respect to snoop-based cache coherence. 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. These caches may entail large memory requirements that grow with the system size, and therefore, it is important to reduce number of tracked cache blocks in order to keep their size manageable. Evictions in the directory cache cause the invalidation of blocks stored in the private caches, since the directory will not be able to track those blocks anymore. These invalidations generate misses known as directory coverage misses [26], which may degrade the system performance.

Cuesta et al. proposed to deactivate cache coherence for private blocks [6] and later for read-only blocks [27]. The mechanism consists of storing only those blocks in the directory which need coherence management, thus reducing the directory size. This technique requires that the private nature of the block has to 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 [6]. 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.

3. TLB-based block-grain classification

This section describes our proposed fine-grain TLB-based classification 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 kept updated using coherence messages exchanged by the TLBs in the system.

3.1. The concepts

We detect and classify a memory block as private if it is not being accessed currently by any other core. A block is considered accessed by a core if the core has recently accessed the block and the TLB entry of the page containing that block is still present. When a core accesses a memory block for which its private/shared nature is unknown, the core will query the other cores in the system to know if the other cores are accessing the block. 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 corresponding block status is restored, and after that, both cores can access them as a shared block.

3.2. 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 this core has accessed the corresponding block. A bit set to zero means that it has not been accessed. The private or shared information is also stored in the TLB to avoid sending 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, while a bit set to zero indicates that the block is shared.

Figure 1 shows the structure of a TLB entry with its most relevant fields: the Virtual Page Number (VPN) and the Physical Page Number (PPN), and

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

Figure 1: TLB entry with extra fields

the two bit vectors with access and private information for the blocks belonging to the corresponding page. When a block has been accessed, the Private bit information gives the classification of the block. However, when a block has not been accessed, the Private bit provides 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 additional actions are necessary. When the block is in the state (0,1), 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. If the core locally accesses a block in the state (0,0), a TLB-to-TLB request has to be issued because of the TLB classification miss.

The two vectors added to each TLB entry only need to be accessed and updated on cache misses. Although we represent the vectors as part of the TLB structure, both parts could be decoupled: the TLB structure is accessed in parallel to the cache access and the vector is accessed only in case of a cache miss. This way, the critical path of a cache hit is not affected.

3.3. The classification protocol

Figure 2 shows the basic concept behind the TLB communication which helps to update the TLB vectors to know the classification of blocks. The basic idea divides into three parts.

- First: A TLB sends the broadcast request to the other core TLBs to know the status of the blocks. It broadcasts only in two scenarios: TLB miss

Table 1: Access and private information and meaning

Access	**Private**	**Meaning**
1	1	The core has accessed the block. The block is private. Only one core can have this block in this state.
1	0	The core has accessed the block. The block is shared.
0	1	The core has not accessed the block. 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 core has not accessed the block. On access, it is necessary to check if the other core is accessing it as private.

and TLB classification miss. The later, status (0,0), means even though the TLB entry is present, the classification of the block is not known.

- Second: When the TLBs of the other cores receive the request they update the vectors in the following way: if a block is private, status (1,1), no other core TLB can have the same status for that block; if the block is shared, status (1,0), other cores can be shared or (0,0); if the block is potentially private, status (0,1), the other cores will have a status (0,0) for that block. The information about the accesses for each block is sent back to the requester TLB.

- Third: When the TLB receives all replies from the other TLBs in the system, it updates its vectors as we detail in this section.

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.

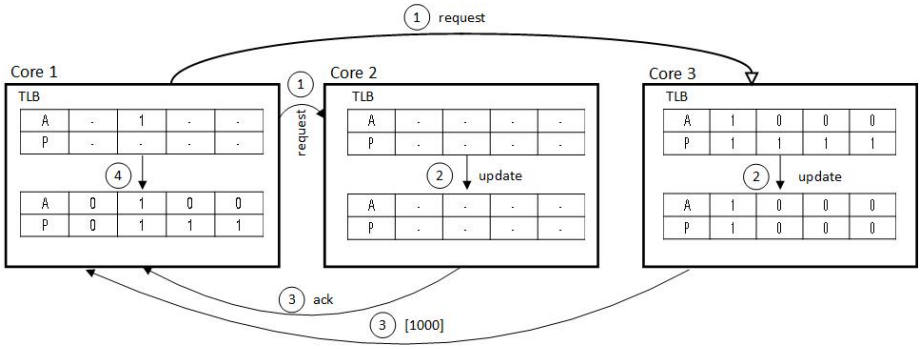

Figure 2: Communication between core TLBs

3.3.1. 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 block's access is stalled until the classification for the block is determined, which happens when all responses from other TLBs are collected.

3.3.2. 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 [23, 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 the 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 1). If the TLB is not currently issuing a TLB request, the information sent in the reply corresponds to the *Access* bit vector in the absence of the TLB entry or a *Nack* in the presence of the TLB entry. If a TLB request has been initiated, then the block that began the request is set to one in the bit vector, since the core is currently accessing the block.

There are two prominent 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), it is considered private, but another TLB is asking it. In this case, the block transitions to (1,0), that is, shared, and a recovery mechanism (see Section 3.5.3) 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 private block and another TLB is considering it as shared). When the TLB has initiated a request for the page, it is crucial 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 2 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

Table 2: 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)

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.

3.3.3. 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 *Private* bit vector's information is updated as indicated in the last column of Table 2.

3.4. Protocol state transitions

This section explains the different states and transitions for a block, depicted in Figure 3. 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

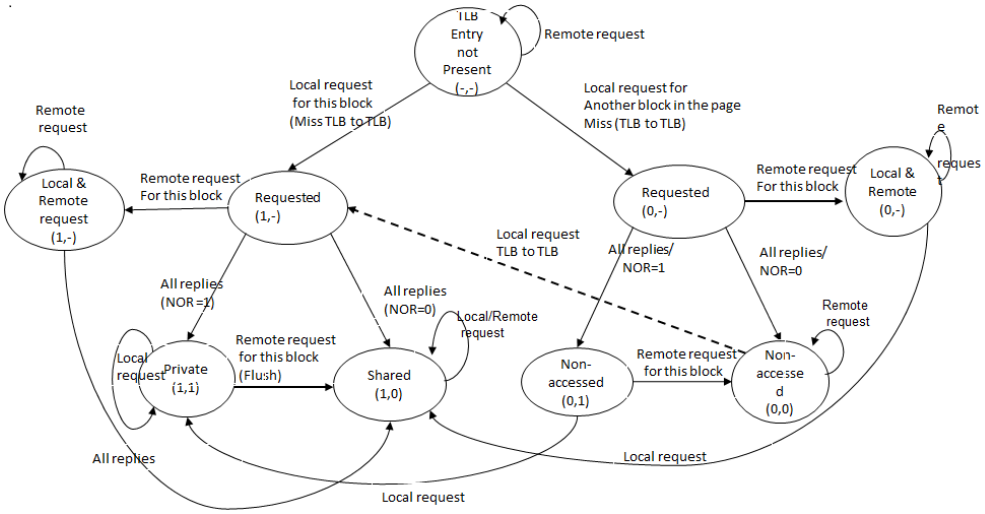

Figure 3: Protocol state transition diagram

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 to do recovery since the

block has not been accessed yet. If the core locally accesses a block in the state (0,0), a TLB-to-TLB request must be issued because of the TLB classification miss. The request informs the other TLBs about the block's access 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. If a block is evicted in a shared state (1,0) state is going to change to the start state, not present(-,-).

If a TLB entry is evicted or flushed due to a lack of TLB capacity or a TLB shootdown, private/shared block information 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. All blocks belonging to the evicted page have to be removed from the cache to keep the data's coherence. 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 each block's lookup and eviction as the access information is not present.

3.5. Reducing TLB classification misses

A block-grain classification disadvantage to a page-grain type is the extra TLB traffic generated due to TLB classification misses. To reduce this additional traffic, we propose the following optimizations that exploit spatial locality, use the available information in a more efficient way, and send data blocks along with TLB responses.

3.5.1. Spatial locality (SL) optimization

The previously explained classification protocol does not consider spatial locality in the accesses, and all potential private blocks –not accessed by the core receiving a TLB request– are offered as private to the requestor. Considering spatial locality can reduce the TLB-to-TLB requests generated as a consequence

of TLB classification misses (0,0), which are extra TLB misses introduced by the classification mechanism.

When employing the spatial locality optimization, on the receipt of a remote TLB request, the TLB does not give away all potentially private states (0,1) for the requested page block. In contrast, it offers only as potentially private blocks those ranging from the requested one to the first block already accessed by the local core (not included) in ascending address order. Due to spatial locality, there is a high probability of accessing a block after accessing the previous one. 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, indicating to the requester that it is in use, even if they are actually not in use. Note that potentially private blocks (0,1) transition to private blocks (1,1) silently when the core accesses them.

3.5.2. Access permission prefetch (APP) optimization

The key idea behind this optimization is to avoid having blocks that are classified as shared in (0,0) state, since an access that find the memory block in that state in the TLB generated a broadcast to collect the classification information. It is therefore preferable to have the block in (1,0) state, i.e, accessed and shared, even if it has not been accessed before. The reason is that in this case, no communication among TLBs is required. We call this optimization access permission prefetch, and it is an optimized version over our previous conference approach [13], that removes the issue of an extra bit vector per response through the interconnection network.

In particular, when the requesting TLB detects that a memory block is shared as it received responses from several cores accessing it, it will mark the block as accessed too, apart from as shared. That is, it sets the block in (1,0) state, and a future access will not generate a TLB classification miss as it occurs in (0,0) state.

3.5.3. Opportunistic data transfer (ODT) optimization

When a private block is changing from private to shared, a recovery mechanism is triggered. Our current solution is to invalidate the block that will be shared, and copy it to the last level cache in case it is dirty. Then, after the block has been invalidated and classified as shared, the core willing to access the new shared block will have a cache miss, will look for the data at the last level cache, and will store it in its local private cache. The opportunistic data transfer optimization aims to prefetch the data block that will be accessed after a TLB miss or a classification miss. If the block is sent from the cache doing the recovery to the requester cache in the TLB-to-TLB transaction, a cache miss will be saved, thus improving performance and traffic.

Since the TLB-to-TLB requests already include the information about the block triggering the TLB or classification miss, the TLB that receives a request for a block classified as private can ask the cache controller to send the corresponding memory block directly to the requester along with the bit vector and address translation. Additionally, the receiver updates the directory information with the new sharer. The requester can also indicate if the TLB request was caused by a cache access requiring exclusive or read-only permissions, and in the first case invalidate the memory block.

A detailed explanation of the coherence transaction implemented to support opportunistic data transfers is depicted in Figure 4 and described here:

1. Core 0 issues a load or store operation to a memory block.
2. The load/store operation accesses the TLB. In case of a TLB or classification miss for the requested memory block, a TLB broadcast request to other core TLBs is generated.
3. The TLB of Core 1 receives the TLB request from core 0. Core 1 TLB checks that the block state is accessed and private (1,1), and therefore a recovery process is required. The state of block changes to shared (1,0).
4. The TLB of Core 1 locks the requested block until the recovery completes. It generates a FWD_GETS_TLB message in case of a remote load opera-

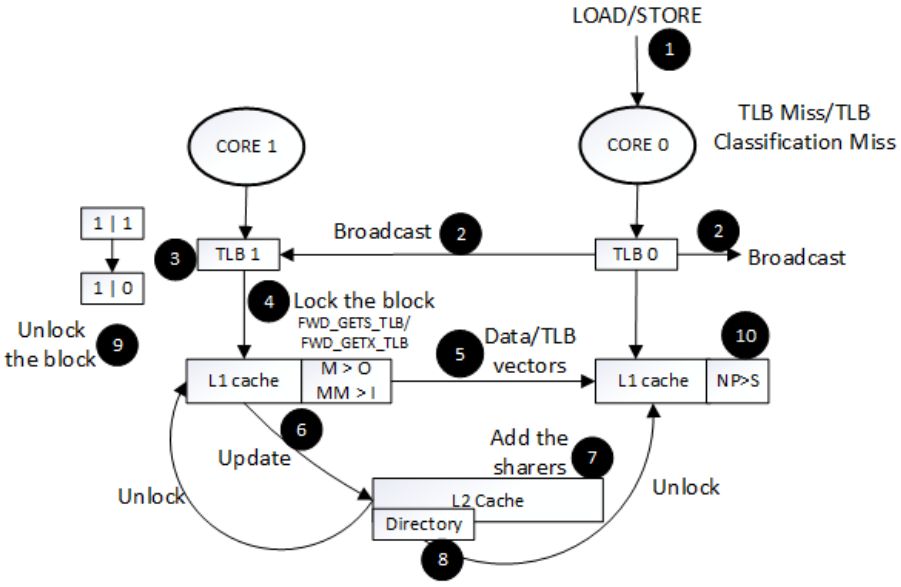

Figure 4: Opportunistic data transfer transaction

tion or a FWD_GETX_TLB message in case of a remove store operation.

5. The L1 of Core 1 updates the state of the block based on load/store operation in the remote core. Core 1 sends the memory block along with the use vector to TLB 0. The load or store operation in Core 0 is resolved at this time.
6. The L1 of Core 1 also sends an Update message to the L2 cache in order to inform the directory about the new sharer.
7. Once the L2 receives the Update message, it allocates an entry in the directory and adds the corresponding sharers.
8. The L2 sends the Unlock signal to the L1s of Core 0 and Core 1 to complete the recovery process.
9. Once the TLB of Core 1 receives the unlock signal from the L2 cache, it unlocks the block doing recovery and finishes the process.
10. Once the TLB of Core 0 receives all replies from other core TLBs and the Unlock message from L2, the L1 of Core 0 state changes to a shared state.

It is important to note that the block grain recovery mechanism is much

simpler and faster than a page-grain approach as block grain recovery expels or updates the requested block only, while page grain evicts or updates all blocks in a page. This is shown in Section 5.

3.6. Size of TLB response messages

Table 3 shows the different types of TLB response messages and size for each optimization. For each scheme we show the larger possible message. Therefore, not all TLB responses contain all the information listed.

Table 3: Types of TLB response messages and size

Scheme	**Message type**	**Size (bytes)**
Page	Control message, address translation	8 + 4
Block	Control message, address translation, use vector	8 + 4 + 8
Block+SL	Control message, address translation, use vector	8 + 4 + 8
Block+SL+APP	Control message, address translation, use vector	8 + 4 + 8
Block+SL+APP+ODT	Control message, address translation, use vector, data block	8 + 4 + 8 + 64

3.7. 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. This is because two blocks with different virtual addresses cached in different cores but mapping to the same physical address will be classified as private, while indeed they are shared.

Synonyms can be introduced by different reasons. The most common are due to page evictions and allocation. However, existing mechanisms in current operating systems, such as TLB shutdown, already take care of preventing these synonyms to coexist in the system.

It is also possible for the programmer to force the presence of synonyms. While we do not encounter such behaviour in our applications, a possible solution could be to rely on the operating system to detect pages with such synonyms. Memory pages containing synonyms would be therefore treated as shared by default. When a page containing synonyms is accessed for the first time, the TLB vectors would be filled with the status (1,0) and it will remain with that status until page eviction.

3.8. Impact of thread migration

Our classification mechanism classifies accesses performed by the cores and it is not aware of thread migration. In presence of thread migration a cache block that is accessed by a single thread can be thus classified as shared by our classification technique. This will only happen however when the page translation information still present in the TLB where the thread was running before. Otherwise, the block will be classified as private. Since thread migration is not a frequent event, takes thousand of cycles, and it happens when cores are contended, thus leaving the thread de-scheduled for sometime, this is a very infrequent scenario. Hence, TLB classification is a good proxy for thread-level classification.

3.9. Supporting large pages

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, requiring larger bit vectors in the TLB, which can dramatically increase the storage requirements of our approach.

To compensate for this effect, or even to reduce the TLB bit-vectors' size when 4KB pages are used, an alternative is to use one bit to represent a group

Table 4: 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

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. The impact of this approach is twofold. On one hand, the classification accuracy would be reduced, but will be always more accurate than classifying full pages. On the other hand, the traffic requirements would be reduced, since with a single broadcast we classify several cache blocks. Analysing the impact of a coarse-grain representation is out of the scope of this work.

4. Simulation environment

The proposed approaches are implemented with a full-system simulation using Virtutech Simics [28] and Wisconsin GEMS toolset [29]. The simulated architecture is a 16-tile CMP architecture with directory-based cache coherence. Table 4 shows the configuration for the simulated system. We have refined our simulation model with respect to our previous conference publication to account for message size overheads accurately. The TLB-to-TLB traffic uses the same interconnect as the coherence messages for data blocks. Contention is modeled in the network using the Garnet simulator [30], and we employ the same network topology and bandwidth for all configurations.

Proposed schemes were evaluated for 15 different benchmarks consisting of parallel workloads from SPLASH [31] and PARSEC [32]. Table 5 shows the simulated benchmarks with their input size. All the reported experimental results correspond to the region of interest of these benchmarks, that is, their parallel phase. Once the slower thread completes its work, the parallel phase concludes and results are reported.

5. Results

This section shows a quantitative comparison of block-grain classification approach, comparing them to a TLB-based classification mechanism that operates at 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, *Block+SL+APP* adds on top the access permission prefetch optimization, and *Block+SL+APP+ODT* improves the previous approach by reducing the L1 cache misses with opportunistic data transfers.

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. Recovery scheme helps to reduce the L1 cache misses. Then,

Table 5: Benchmarks and input sizes

Benchmark	Input size
SPLASH Benchmarks	
Barnes	8192 bodies, 4 time steps
Cholesky	tk15.O
FFT	64K complex doubles
FMM	16K particles
LU	512×512 matrix
LUNC	1024×1024 matrix, 64×64 blocks
Ocean	258×258 ocean
Radiosity	room, -ae 5000.0 -en 0.050 -bf 0.10
Volrend	Head
Watersp	4096 molecules
Radix	8,388,608 integers
PARSEC Benchmarks	
Blackscholes	Simmedium
Fluidanimate	Simsmall
Swaptions	Simsmall
x264	Simsmall

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.

5.1. Private accesses

The main goal of the block-grain approach is to detect more private blocks by removing the miss-classification of blocks in page-grain counterparts. Block-grain achieves more accuracy in detecting private blocks than the page-grain approaches as it detects private blocks in shared pages. Figure 5 shows the per-

centage of L1 accesses for private blocks (hits and misses) in a page-grain scheme (Page, First bar) and the proposed block-grain schemes: Block (Second bar), Block+SL (Third bar), Block+SL+APP (Fourth bar) and Block+SL+APP+ODT (Fifth bar). Results show that Page, Block, Block+SL, Block+SL+APP and Block+SL+APP+ODT detected, on average, 22.4%, 30.7%, 32.9%, 33.4%, and 33.4% private L1 hits respectively and 0.12%, 0.24%, 0.29%, 0.29%, and 0.30% private L1 misses, respectively. Our block-grain approaches remove the miss-classification of page level approaches and access on average 1.5 times more private blocks that miss in L1 than the page-grain approach. The Block, Block+SL, Block+SL+APP and Block+SL+APP+ODT accessed, on average 12%, 17%, 17%, and 18% more private miss blocks than the page level scheme. This clearly shows that block-grain approaches remove the miss-classification of page-level approaches. As we will see, this more accurate classification of private blocks reduces the number of entries required in the directory structure.

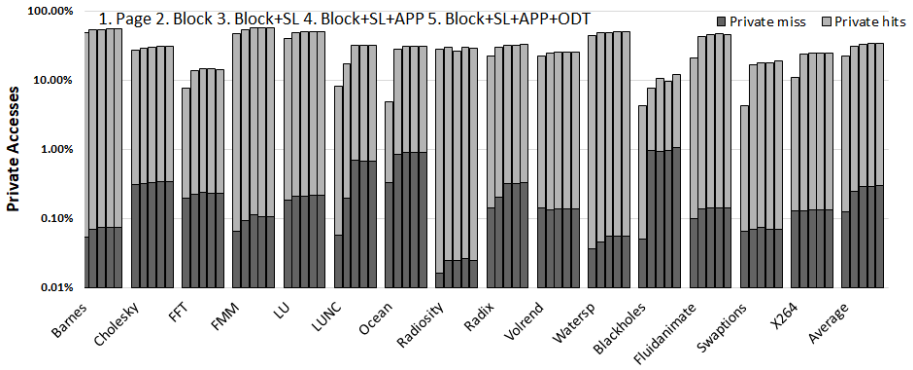

Figure 5: Private hit and private miss accesses in percentage

5.2. 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 6 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, Block+SL+APP, and Block+SL+APP+ODT is 3.1%, 2.9%, 1.0%, and 1.0% respectively compared to the page-grain approach. Block+SL+APP and Block+SL+APP+ODT reduce 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 discussed in the next section. LU and Cholesky are the applications showing more TLB request overhead.

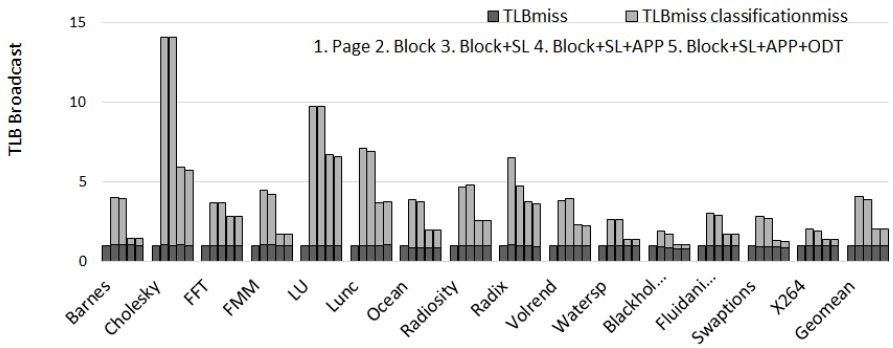

Figure 6: TLB miss broadcast overhead in Block grain

5.3. 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. As a consequence of flushing a single block instead of iterating the whole page, 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 7 shows the latency of the recovery mechanism both in the page-grain approach and in the block-grain approaches. The page (First bar)requires on average 97.7 clock cycles for the recovery operation. On the other hand, block-grain approaches require considerably less time: 24.2 cycles

on average for Block (Second bar), 24 cycles on average for Block+SL (Third bar), 24 cycles for Block+SL+APP (Fourth bar), and 26.3 cycles on average for Block+SL+APP+ODT (Fifth bar) scheme. The Block+SL+APP+ODT scheme merges TLB communication with the cache miss resolution which results in 2.4% extra recovery latency compared to the other block grain approaches. However, it is still lower than the page grain approach, since the page grain approach manages the recovery of up to 64 blocks.

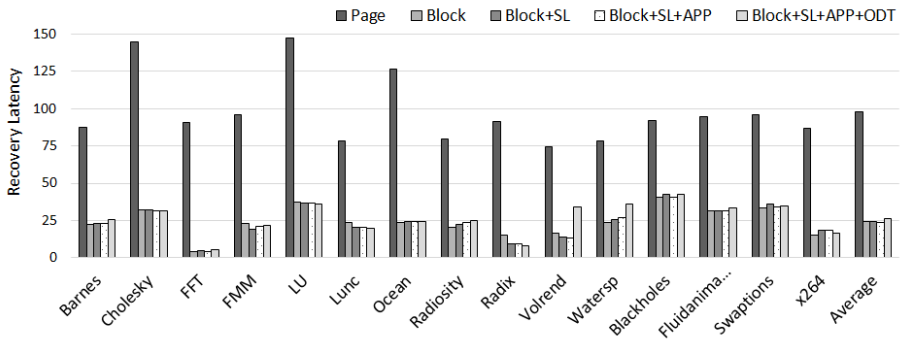

Figure 7: The latency of the recovery mechanism(cycles)

5.4. Average directory entries required (per cycle)

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 8 shows the normalized number of directory entries required with respect to Base, where all cached blocks are tracked by the directory (no deactivation is performed). When employing our fine-grain approach the required directory entries fall dramatically. Page(Second bar) avoids the storage of 34%, Block (Third bar) avoids the storage of 50.6% of the entries, Block+SL (Fourth bar) avoids the storage of 51.6% of the entries, Block+SL+APP (Fifth bar) avoids the storage of 63.1% of the entries and Block+SL+APP+ODT (Sixth bar) avoids the storage of the 62.3% compared to baseline approach. Additionally, when compared to the page-grain scheme, our block-grain schemes reduce the number of required entries by 25.3%, 26.3%, 43.8%, and 42.6%, respectively, for the Block, Block+SL, Block+SL+APP, and

Block+SL+APP+ODT. 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.

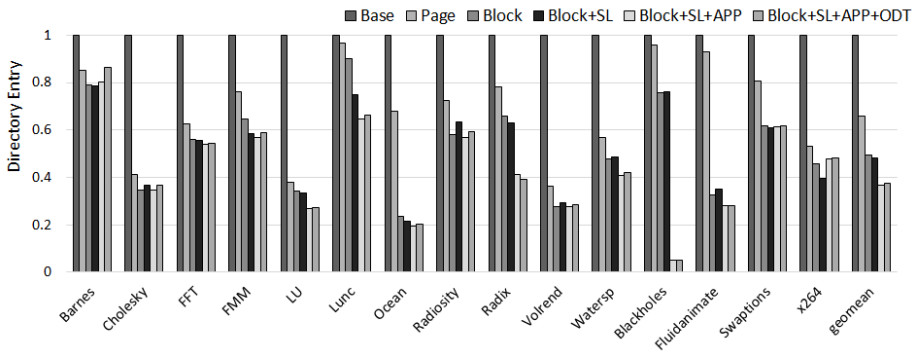

Figure 8: Average directory entries required (per cycle)

5.5. L1 cache misses

Thanks to the reduction in the directory cache requirements that cause fewer evictions in the directory cache and less coverage misses, thanks to having less flushed blocks from the L1 cache, a reduction in L1 cache misses is found in block-grain schemes. Additionally, the Block+SL+APP+ODT scheme sends the data block along with the TLB communication on recoveries. This helps to improve the hit rate of shared blocks and reduces the L1 cache misses. Figure 9 shows the L1 cache miss ratio normalized with reference to Base for the page approach (First bar), Block (Second bar), Block+SL (Third bar), Block+SL+APP (Fourth bar) and Block+SL+APP+ODT (Fifth bar). 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). Compared to baseline protocol, Block reduces L1 cache-miss of 50% of the entries, Block+SL avoids the L1 cache-miss of 53% of the entries, Block+SL+APP reduces L1cache-miss of the 53% and

Block+AL+APP+ODT reduces L1cache-miss of the 55%. Additionally, when compared to Page, our block-grain schemes reduce the L1 cache-miss by 17%, 24%, 24% and 25%, respectively, for the Block, Block+SL, Block+SL+APP, and Block+SL+APP+ODT. On average, the page has 31% of coverage misses compared to the total amount of misses while the Block has on average 10% 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. Our new Block+SL+APP+ODT scheme reduces the L1 cache misses by 2% (up to 5% in Radix) compared to the BLOCK+SL+APP scheme.

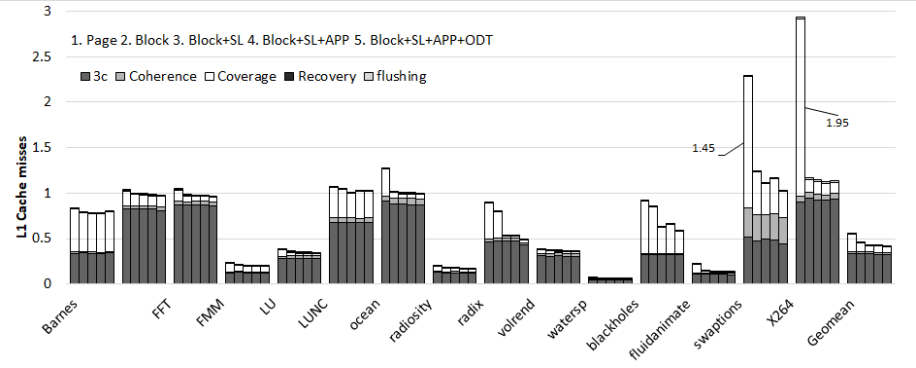

Figure 9: Normalized L1 cache miss rate with respect to baseline

5.6. Normalized network traffic under coherence deactivation

The network traffic is also reduced in the proposed block-grain approaches even with some TLB communication overhead. The reduction in messages occurs because of fewer invalidation messages from the directory cache and a lower L1 cache miss ratio. Figure 10 shows the normalized network traffic compared to baseline for the page approach (First bar), Block (Second bar), Block+SL (Third bar), Block+SL+APP (Fourth bar) and Block+SL+APP+ODT (Fifth bar). 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-grain approaches detect more private data and less cache misses,

which results in less traffic. On the other hand, the TLB-to-TLB communication network's overhead increases the amount of on-chip network traffic. These two opposite communication trends can roughly balance out and still proposed approaches to reduce the network traffic. Block, Block+SL, Block+SL+APP, and Block+SL+APP+ODT reduce the network traffic by 33.4%, 40.2%, 44.1%, and 45.9% respectively compared to the baseline setup. Block increases network traffic 11% compared to Page as having more TLB communication overhead. Block+SL has same network traffic as page and Block+SL+APP reduces 7% network traffic compared to Page with the help of a reduction in TLB broadcast. The Block+SL+APP+ODT scheme reduces the 10.1% reduction in network traffic compared to page scheme because of reduction in L1 cache misses. Watersp reduces around 93.0% 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.

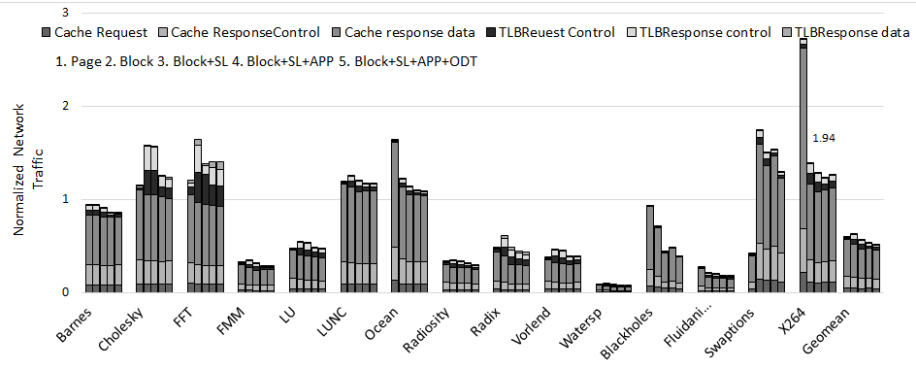

Figure 10: Normalized network traffic under coherence deactivation

5.7. Execution time for coherence deactivation

Given all the previous analysis (lower L1 cache miss rate, a larger number of private misses, less network traffic, and less directory occupancy) we can expect reductions in execution time with the block-grain approaches. Figure 11 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), Block+SL+APP_1/8

(Sixth bar), and Block+SL+APP+ODT (Seventh 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, Block+SL+APP, and Block+SL+APP+ODT reduce the execution time by 8.4%, 10.4%, 11.8% and, 13.9%considering 8 times smaller directory cache when compared to Base. Furthermore, when compared to Page, they reduce the execution time by 2.0%, 4.1%, 6.0% and 8.0% respectively.

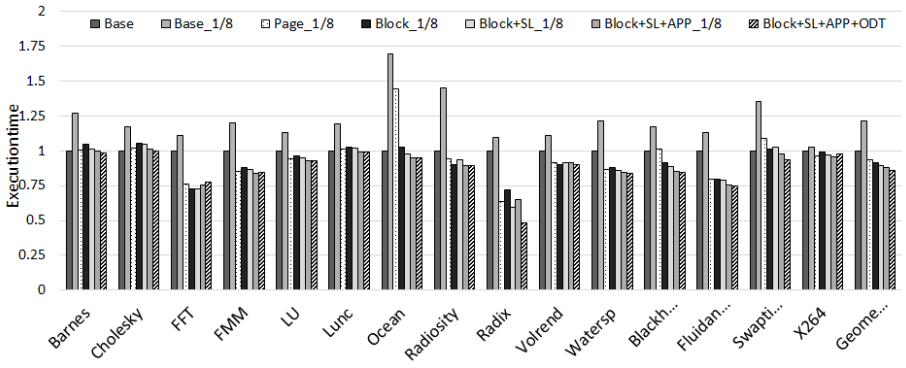

Figure 11: Execution time for coherence deactivation

5.8. Scalability analysis

This section shows how the different fine-grain schemes scale with the coherence deactivation. We just perform the scalability analysis for the applications that finished within 5 days for 32 cores for each application, that is, all benchmarks mentioned in Table 5 except x264 and swaptions. Figure 12 shows how the Block+SL+APP+ODT scheme scales better compared to the Page scheme. Block+SL+APP+ODT reduces the execution time by 13% for 8 cores, by 15% for 16 cores, and by 21% for 32 cores with respect to the baseline configuration. On the other hand, the page scheme reduces the execution time by 9% for 8 cores, by 8% for 16 cores, and by 13% for 32 cores over the baseline. This difference shows that how performing a more accurate classification provides better directory usage, and ultimately better system performance and scalability.

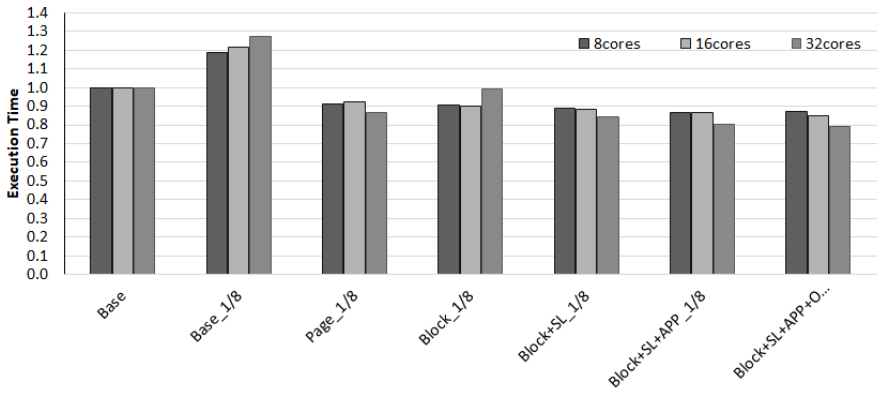

Figure 12: Execution time for coherence deactivation for different number of cores

5.9. Memory overhead

Our block-grain approaches trade off non-scalable directory entries with scalable TLB bit vectors. Therefore, as the number of cores grows in the system, more memory will be saved than 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.

6. 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 [14]. 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 [33].

6.1. Fine-grain approaches

Fine grain approaches include compiler- and directory-based approaches. In a compiler-based approach [7, 15] 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 [17] the L1 cache stores non-coherent blocks and the L2 cache stores coherent blocks. POPS [34] optimizes the protocol by combining private and shared memory blocks on various L2 cache slices in the NUCA architecture. Valls et al. [35] 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 [19] is a directory-based mechanism that temporarily allocates a single record to a private region rather than assigning 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.

6.2. Coarse-grain approaches

Coarse grain approaches perform a classification of memory pages with the page table's help and/or the TLB [6, 20, 36, 10]. Cuesta et al. [6] use the page

table to detect the memory pages' nature and deactivate coherence for blocks in private pages. Ros et al. [20] improve the classification accuracy thanks to TLB-to-TLB communication and a late discovery of TLB evictions. TokenTLB [21] is a token-based coherence mechanism [37], that reduces TLB communication. With a prediction of the use of pages [22] the accuracy of classification can be improved, as TLB evictions can be detected earlier. The forced-sharing predictor [36] 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. [11] have proposed a hardware-software co-design proposal using a parallel programming model to deactivate the cache coherence protocol, reducing the directory area energy consumption with the help of extra hardware. Studies at sub-page granularity have also been performed recently [38], by using an on-chip page table to find address translation. This proposal requires to modify the operating system and the page tables to store, among other information, a *keeper* for each of the sub-pages. The amount of storage required to hold all keepers, information that increases as the number of cores increases, for the vast virtual space can be prohibitive. In contrast, our proposal is able to work with finer granularity, and consequently achieving higher accuracy, with low and scalable storage requirements, at the same time that it does not require modifications to the operating system. In addition, a block-grain approach enables a much simpler recovery mechanism compared to page-level or coarser-grain since only a single cache line per TLB request is recovered. Finally, in order to reduce the traffic required to communicate TLBs, we propose three optimization techniques: spatial locality, access permission prefetch, and opportunistic data placement, which could be also applicable to sub-page grain schemes. Finally, a real time prototype of private/shared data classification has been recently implemented on LEON SPARC multiprocessor[39] using a page-grain approach.

7. Conclusions and future work

Categorizing memory accesses into private and shared helps to achieve scalability and efficiency in a multicore system. This paper proposes a novel approach to augment each TLB entry with bit vectors that denote each cache block's sharing and access status within the corresponding page. This way we can categorize data as private or shared at a finer granularity than existing approaches using the TLB structures. Without loss of generality this classification can used to avoid coherence for private data, freeing up directory entries.

The proposed scheme detects, on average, more accessed private miss blocks 18.0% than previous approaches and results in performance improvement of 8.0% 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 study classification mechanisms in a snooping based cache coherence scenario, eliminating the non-scalable directory structure and using only scalable TLB bit vectors, while reducing interconnection traffic.

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] D. E. Culler, J. P. Singh, A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann Publishers, Inc., 1999.

[2] N. Hardavellas, M. Ferdman, B. Falsafi, A. Ailamaki, Reactive nuca: near-optimal block placement and replication in distributed caches, in: ACM SIGARCH Computer Architecture News, Vol. 37, 2009, pp. 184–195.

[3] B. R. Upadhyay, 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.

[4] D. Kim, J. Ahn, J. Kim, 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.

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

[6] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, 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.

[7] Y. Li, R. Melhem, 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.

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

[9] S. Shukla, 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.

[10] A. Esteve, A. Ros, M. E. Gómez, A. Robles, J. Duato, Tlb-based temporality-aware classification in cmps with multilevel tlbs, IEEE Transactions on Parallel and Distributed Systems (TPDS) 28 (8) (2017) 2401–2413.

[11] P. Caheny, L. Alvarez, M. Valero, M. Moretó, 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.

[12] J. Dumas, E. Guthmullcr, 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.

[13] B. R. Upadhyay, A. Ros, N. S. Murty, Tlb-based block-grain classification of private data, in: 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2020, pp. 122–130.

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

[15] Y. Li, R. Melhem, A. Abousamra, 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.

[16] J. F. Cantin, M. H. Lipasti, J. E. Smith, Improving multiprocessor performance with coarse-grain coherence tracking, in: ACM SIGARCH Computer Architecture News, Vol. 33, 2005, pp. 246–257.

[17] S. H. Pugsley, J. B. Spjut, D. W. Nellans, 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.

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

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

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

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

[22] A. Esteve, A. Ros, A. Robles, M. E. Gómez, Tokentlb+cup: A token-based page classification with cooperative usage prediction, IEEE Transactions on Parallel and Distributed Systems (TPDS) 29 (5) (2018).

[23] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, 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.

[24] S. Srikantaiah, 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.

[25] D. Lustig, A. Bhattacharjee, M. Martonosi, Tlb improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level tlbs, ACM Transactions on Architecture and Code Optimization (TACO) 10 (1) (2013) 2.

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

[27] B. Cuesta, A. Ros, M. E. Gomez, A. Robles, J. Duato, Increasing the effectiveness of directory caches by avoiding the tracking of noncoherent memory blocks, IEEE Transactions on Computers 62 (3) (2013) 482–495.

[28] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, Simics: A full system simulation platform, Computer 35 (2) (2002) 50–58.

[29] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, D. A. Wood, Multifacet's general execution-driven multiprocessor simulator (gems) toolset, ACM SIGARCH Computer Architecture News 33 (4) (2005) 92–99.

[30] N. Agarwal, T. Krishna, L.-S. Peh, N. K. Jha, GARNET: A detailed on-chip network model inside a full-system simulator, 2009, pp. 33–42.

[31] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, The splash-2 programs: Characterization and methodological considerations, ACM SIGARCH computer architecture news 23 (2) (1995) 24–36.

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

[33] M. Davari, A. Ros, E. Hagersten, S. Kaxiras, The effects of granularity and adaptivity on private/shared classification for coherence, ACM Transactions on Architecture and Code Optimization (TACO) 12 (3) (2015) 26.

[34] H. Hossain, S. Dwarkadas, 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.

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

[36] A. Esteve, A. Ros, M. E. Gómez, A. Robles, J. Duato, Efficient tlb-based detection of private pages in chip multiprocessors, IEEE Transactions on Parallel and Distributed Systems (TPDS) 27 (3) (2016) 748–761.

[37] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. Martin, D. A. Wood, Improving multiple-cmp systems using token coherence, in: 11th International Symposium on High-Performance Computer Architecture, 2005, pp. 328–339.

[38] M. Soltaniyeh, I. Kadayif, 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 37 (4) (2018) 806–819.

[39] N. Ho, I. I. Ashraf, P. Kaufmann, 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.