CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **00**:1–12
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

A dedicated private-shared cache design for scalable multiprocessors

Juan. M. Cebrián[1,*] Ricardo Fernández-Pascual[1], Alexandra Jimborean[2], Manuel E. Acacio[1] and Alberto Ros[1]

[1] *Computer Engineering Department, Universidad de Murcia, Murcia, Spain*
[2] *Department of Information Technology, Uppsala Universitet, Uppsala, Sweden*

SUMMARY

Shared-memory architectures have become predominant in modern multi-core microprocessors in all market segments, from embedded to high performance computing. Correctness of these architectures is ensured by means of coherence protocols and consistency models. Performance and scalability of shared-memory systems is usually limited by the amount and size of the messages used to keep the memory subsystem coherent. Moreover, we believe that using the same mechanism to keep coherence for all memory accesses can be counterproductive, since it incurs unnecessary overhead for private data and read-only shared data. Having this in mind, in this paper we propose the use of dedicated caches for two kinds of data. The private cache (L1P) will be independent for each core and will keep data that can be accessed without contacting other nodes (i.e., private data and read-only shared data), while the shared cache (L1S) will be logically shared but physically distributed for all cores and will keep modifiable shared data. This separation should allow us to simplify the coherence protocol, reduce the on-chip area requirements and reduce invalidation time. This dedicated cache design requires a classification mechanism to detect private and shared data. Results show two drawbacks to this approach: first, the accuracy of the classification mechanism has a huge impact on performance. Second, a traditional interconnection network is not optimal for accessing the L1S, increasing register-to-cache latency when accessing shared data.
Copyright © 2015 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Coherence protocols, private data, shared data, separate caches

1. INTRODUCTION

Modern chip multi-processors feature a mix of latency and throughput oriented cores running both sequential and multithreaded applications and mixed workloads. These architectures usually share a common memory space that is kept coherent through hardware mechanisms. While coherence and memory consistency are key aspects to ensure the correct execution of the applications on the system, they usually incur performance overheads proportional to the size of the system (i.e., number of cores, interconnection characteristics, etc). In addition, the implementation choice of the cache coherence protocol has a considerable impact on scalability in terms of area requirements, application execution time and energy expenditure.

Data accessed by applications show a wide range of data sharing degrees, from thread local data to data shared by multiple cores. The sharing degree is mainly influenced by the programming methodology and the nature of the application. However, traditional coherence protocols maintain coherence in the same way for all data accesses, regardless of the nature of the data that is being

[*]Correspondence to: jcebrian@ditec.um.es

accessed. In other words, blindly handling all memory accesses in the same way can be suboptimal, since the coherence protocol incurs in unnecessary overhead for accesses targeting addresses that would remain coherent anyway, such as private data [1] and read-only data [2].

Recent literature provides plenty of hardware optimizations that make use of the private or shared nature of data or accesses. For example, Hardavellas *et al.* [3] and Li *et al.* [4, 5] employ the classification to optimize distributed shared caches (NUCA: Non-uniform Cache Architecture), Kim *et al.* [6] improve the efficiency of token-based protocols, and Cuesta *et al.* [1, 2] reduces the pressure in the directory caches (see Section 5 for more examples).

The hardware optimizations based on the knowledge of the private or shared nature of the data require mechanisms to detect and classify such properties. There are many classification mechanisms in the literature that work at different levels: compiler [7, 4, 5, 8], operating system (OS) [1, 3, 6], translation look-aside buffer (TLB) [9, 10], or at the coherence protocol level [11, 12, 13].

This work explores a novel architecture design that takes advantage of the nature of the accessed data. Our main objective is to provide simplicity and scalability in a multi-core design. In particular, we rely on the usage of two dedicated caches: a private cache (L1P) and a shared cache (L1S).[†] The L1P cache stores only private and shared-read-only data. Coherence actions for these data blocks are only required when their nature changes from private or read-only to shared-read-write or vice versa. These coherence actions are performed locally, without the need to communicate with other cores, but with the next cache level. On the other hand, the shared cache is logically shared but distributed across all cores, i.e., a NUCA cache. In this case, no coherence actions are needed because there is only one copy of each data block at a particular level in the cache hierarchy. This way, the memory coherence of the system is maintained without needing neither a directory, nor snooping requests.

With this dedicated private and shared cache design we expect to simplify the coherence protocol, since the coherence actions are local to the core executing them, and to eliminate completely the area overhead entailed by the coherence protocol, since no information about the data cached in other cores is required. Our goal is therefore to improve scalability with none or negligible performance degradation. We will guide the reader through the design process highlighting the benefits of the proposal, the problems that led us to negative results, and the future research lines that derive from this work. Our contributions include:

- A dedicated cache design for chip multiprocessors that exploits the nature of the accessed data.
- Analysis of different implementations of the concept, including using a two-dimensional mesh and a point-to-point network.
- Analysis of different private-shared classification techniques, explaining the benefits and drawbacks of each approach.
- Evaluation of the proposal in terms of performance, network traffic and cache miss ratio using several interconnects and classification techniques.
- Discussion on the design flaws of the proposal, conditions required by the architecture to be efficient, and suggestions about future research lines.

The rest of the paper is organized as follows: Section 2 describes the implementation details of our proposal. Section 3 explains the evaluation environment, and provides details of the analyzed processor configuration. Section 4 discusses the main results of our research. Section 5 describes state-of-the-art solutions that exploit the nature of accessed data and compares them to our proposal. Finally, we conclude with Section 6 and suggest directions for further research.

[†]Although in this work we focus on a single level of local cache (per core), the proposed concepts are extensible to several levels of local caches.

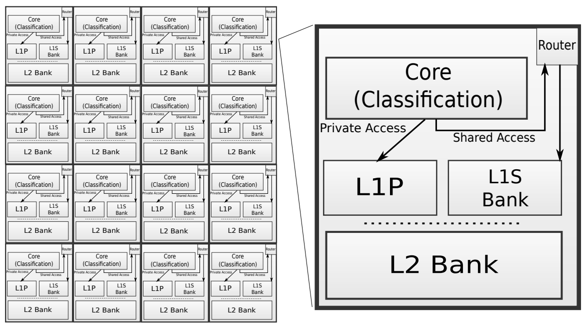


Figure 1. Dedicated cache design diagram.

2. SEPARATE CACHES FOR PRIVATE AND SHARED DATA

In this paper we propose a novel cache design that uses two separate caches to store two types of data. A private L1 cache (called L1P) for each core is used to store both private and shared read-only data, while a logically shared but physically distributed L1 (called L1S) is used to store modifiable shared data. This design avoids having more than one copy of any modifiable data with the intention of simplifying the task of guaranteeing memory coherence.

The L1P works much as a regular private L1 data cache, but it does not have to deal with external requests from other cores. This significantly simplifies the implementation of the cache controller. On the other hand, the L1S works like a NUCA cache in a tiled chip multiprocessor (CMP). Each tile provides a bank for the global L1S and accesses to shared data may perform the look-up in the L1S in the local bank or in a remote bank. The distributed nature of the L1S cache requires an interconnection network to direct the request to the home tile (L1S bank) where the requested data block maps. The L2 cache in our proposal has also a NUCA architecture, but it can store both private and shared blocks as in most other designs.

In the proposed architecture, when a core executes a memory instruction (e.g., load or store), it has to know the type of the access (see Section 2.1) before it is initiated in order to decide if the memory request should be sent to the L1P or to the L1S (Figure 1). The access to the L1P is performed as in standard implementations. However, the access to the L1S may require extra latency. In particular, when the access is performed in a remote bank, an 8-byte (control) request message is sent to the corresponding bank, and an 8+8-byte response message (control+requested word) is sent back to the requesting core through the interconnect.

The main benefits of this design are the following:

- ✓ It eliminates the need for core-to-core communication (both directly or indirectly) to keep cache coherence. No invalidation messages are issued upon write misses, because shared read-write data always reside in a unique place in the shared cache. On the other hand, no forwarding messages are required to read a copy cached by a different core, since the up-to-date copy of the data is guaranteed to be in the shared cache. This considerably reduces the complexity of cache controllers and the number of protocol transient states.
- ✓ It completely eliminates the need for a directory, since no invalidation or forwarding messages are required. Provided that, typically, the area required by the directory structure is one of the main limitations for building large-scale multi-core architectures, the scalability of multi-core processors is largely improved by our proposal.
- ✓ It removes the indirection to the directory, thus resolving all cache misses in two hops, without the need of waiting for invalidation acknowledgements.
- ✓ It reduces the pressure in the L1P, and therefore the number of cache misses, since private data will not compete with shared data, which have completely different associativity requirements [14, 15].

- ✓ It reduces the amount of memory required for the L1 caches, since shared blocks are not duplicated in several L1 caches. In general, the L1 caches are better utilized.
- ✓ It allows specialized cache designs for private and shared data blocks.

However, this design does not come without drawbacks. The main disadvantages are the following:

- ✗ It entails extra latency when accessing shared data, even when the access hits in the L1. A low latency interconnect is necessary in order to overcome this drawback.
- ✗ The accesses to shared data are transmitted through the interconnect, thus increasing network traffic. A high-bandwidth interconnect is necessary if the number of shared accesses becomes predominant.
- ✗ It requires an effective classification mechanism to detect the nature of the accesses. This mechanism should provide the information a-priori, that is, before accessing the L1.

The proposed design exhibits numerous advantages, nevertheless, the design of the interconnect and the classification mechanisms are essential for the overall performance. A poor classification, for instance, may cancel the benefits of the L1 split caches. In this work we analyze these important design factors.

2.1. *Classification mechanisms*

Directory-based classification [11, 12, 13, 16, 17], despite providing high accuracy, is not suitable for our proposal, since our architecture requires that the nature of the memory access is known before the access is initiated. More importantly, using such techniques would require maintaining hardware directories, which represent the main bottleneck in traditional coherence protocols, and therefore were eliminated in our architecture design.

Similarly, TLB-based classification [9, 10] is able to detect a large amount of private pages, but at the cost of extra complexity in the design of the TLBs. Since the main goal of this work is to simplify the hardware required to maintain coherence, we discard the TLB-based approach. Next subsections analyze the most promising classification mechanisms.

2.1.1. OS-based classification. The OS-based classification mechanism [1, 3, 6] operates at the OS-level, hence at memory page granularity. This mechanism uses an additional bit in the page table to mark the state of the page as private or shared. This P/S bit is also included in the TLB entries to facilitate the retrieval of the classification information when performing a memory access to guide the core accessing a block towards the cache that contains it.

OS-based classification mechanisms mark pages as private the first time they are accessed after a page fault. The core ID of the node that requests the page is marked as the owner. Next, upon each access, the classification mechanism first attempts to match the core ID of the owner and of the core currently requesting the page. If they differ, an interruption of the first core is triggered, which results in an update of the page table changing the state of the accessed page to shared.

The disadvantage of the OS-based classification is that it is highly conservative: (1) it operates on a very coarse granularity marking as shared accesses that target distinct locations which reside in the same memory page; and (2) it does not account for temporality and merely accumulates information regarding accesses to the same memory page in different stages of execution. Thus, long running programs yield a high degree of sharing, since eventually most of the pages will be accessed by multiple cores.

2.1.2. Compiler-assisted classification. Compiler-assisted approaches inspect the code at compile-time and conservatively classify accesses based on the nature of the target data [4, 5, 7] or based on the nature of the code regions initiating the access [8].

Classifying *data* statically [4, 5, 7] poses monumental challenges for compilers due to dynamic memory allocation, pointer chasing and other statically unknown events, yielding such classifications either inaccurate, and thus not suitable for our proposal which must guarantee

Table I. System parameters.

Memory parameters	
Block size & page size	64 bytes & 4 KB
L1 (or L1P) cache	32 KB, 4 ways, 4 cycles access latency
L1S cache	(32 KB, 4 ways) per tile, 4 cycles + Network access latency
L2 cache (shared)	512 KB/tile, 16 ways, 12 cycles access latency
Memory access time	160 cycles
Network parameters	
Topology	2-D mesh (4×4) & Point-to-point
Routing method	X-Y deterministic
Message size	5 flits (data), 1 flit (control)
Switch-to-switch time	6 cycles (Garnet) & 1 cycle (Simple)
Bandwidth	1 flit per cycle

correctness, or highly conservative, which provides correctness guarantees but cancels the optimization opportunities.

In contrast, compile-time classifications which exploit information regarding the nature of code regions [8], namely data-race-free (DRF) and non-data-race-free (nonDRF) regions, are highly accurate and do not resort to conservative decisions. Code region classification straightforwardly stems: (1) from the programming paradigm, e.g. OpenMP parallel programming clearly delineates DRF and non-DRF regions based on programmer's annotations; (2) from the programming language semantics, e.g. C++11 standards require DRF semantics; or (3) from compile-time analysis, e.g. in automatically parallelized applications, the compiler has full knowledge of data sharing and of the work distribution among the threads. Based on the code region classification, the compiler marks accesses performed during the execution of DRF regions as private, while accesses within non-DRF regions are shared. Hence, accesses targeting the *same memory location* at different stages of execution may be marked *differently*.

3. EVALUATION METHODOLOGY

The proposed dedicated cache design has been evaluated using the cycle-accurate GEMS [18] simulator. Our simulation infrastructure employs real system data accesses captured by a PIN [19] tool, which are used as input for the GEMS simulator, similarly to other simulation tools [20, 21]. GEMS includes a detailed memory hierarchy model (Ruby) that enables us to estimate performance, access latency, miss ratio, etc. The interconnection network is modeled using the two network simulators included in GEMS. The *Garnet* simulator, which models accurately the network contention, is used to simulate a tiled 2D-mesh design. The *Simple* network model, which does not model contention, is used to model a more optimistic scenario that implements a point to point (P2P) network. The simulated architecture corresponds to a chip multiprocessor (*tiled*-CMP) featuring 16 cores, as depicted in Figure 1. The most relevant simulation parameters are shown in Table I. The evaluated multi-core systems implement two levels of on-chip caches.

The evaluation of the dedicated cache design is compared to a traditional MESI directory-based coherence protocol that stores the directory information in the L2 cache (*MESI-Inclusive* in our figures). We evaluate our proposal using the applications from the Splash-2 benchmark suite with the recommended input sizes [22], and also using applications that can benefit from a compiler-assisted classification, such as OpenMP codes from SpecOMP 2012 [23] (352.nab, 359.botsspar, and 367.imagick —test input—) and Rodinia [24] (bfs —graph1MW_6.txt—, btree —mil.txt, command.txt—, hotspot —1024 × 1024—, particlefilter —128 × 128 × 10, 10000 particles—, and pathfinder —width 50000—).

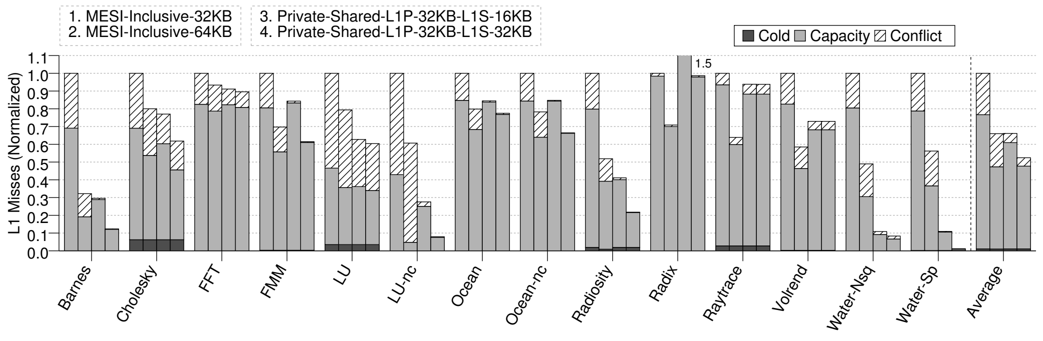


Figure 2. Normalized L1P+L1S cache misses (to unified/MESI L1).

4. RESULTS

This section analyzes the pros and cons of the proposed dedicated cache design, focusing on the two key factors that can affect negatively the performance of the proposed architecture: the interconnection network and the classification of accesses. Our analysis assumes a classification of accesses performed by the operating system, unless otherwise stated.

4.1. Impact of dedicated caches on cache misses.

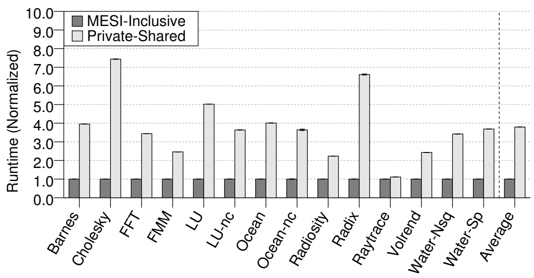
We start our study by analyzing the effects on cache misses when splitting data into two dedicated caches. Since we reduce the amount of replicated data in the L1S, we expect a significant reduction on the cache misses. The results are shown in Figure 2, which plots the number of misses classified in cold, capacity, and conflict, normalized with respect to a MESI configuration with 32 KB L1 caches (all caches employed in this study are 4-way associative). As a second configuration, it plots the misses that would take place in a MESI configuration with 64 KB L1 caches. We can see that, on average, the number of misses are reduced by 35% after doubling the cache size. Then, it shows two configurations with dedicated L1P and L1S caches: the first one (third bar) has a 32 KB L1P cache and a 16 KB L1S cache; and the second one (fourth bar) has a 32 KB L1P cache and a 32 KB L1S cache, that is, the same capacity as the second configuration.

The most important result of this study is that a combined L1P+L1S with a capacity of only 48 KB yields a number of cache misses comparable to a unified cache of 64 KB. Additionally, when moving to an aggregate L1 cache capacity of 64 KB, the number of cache misses is reduced considerably. We attribute this behavior to changes on the reuse distance of the data when using dedicated caches and the reduction of replicated data in the L1S. This effect can improve the performance of the applications if the accesses to the L1S cache can be performed with low latency. Note that an L1P cache with the same capacity as the L1 cache in MESI can be accessed with the same latency.

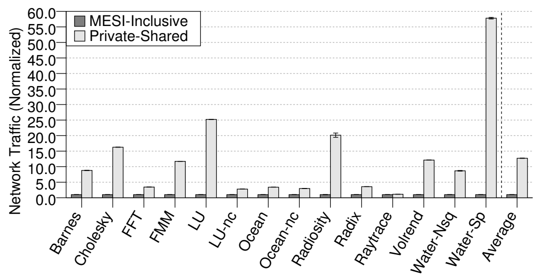
4.2. Impact of the interconnection network

Since all references to shared data have to travel through the interconnection network before accessing the L1S, the design of the interconnection network plays an important role in our architecture. We analyze both a common 2D-mesh topology, which is scalable, and a point-to-point network, which is optimal in terms of performance although not scalable.

4.2.1. 2D-mesh interconnect. Figure 3 presents performance results for an organization with a 2D-mesh interconnect normalized to a MESI implementation. Figure 3a shows the applications' execution time. Results show a considerable slowdown on the overall performance of the applications, reaching 3.7× on average for all Splash-2 benchmarks (7.5× at worst for Cholesky and 1.2× at best for Raytrace). Notice that there is great variability in the results, meaning that some applications can hide this additional memory latency better than others, where, most likely, shared accesses are located on the critical path. However, programs should not be sensitive to a larger latency for shared data when using dedicated caches, since real shared data entails communication

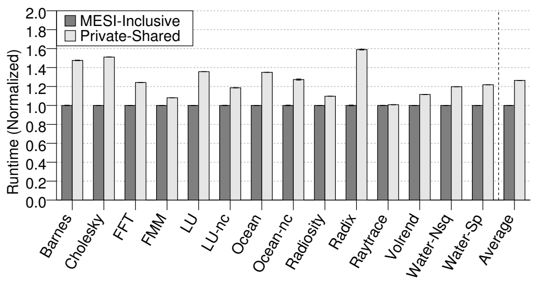


(a) Execution time

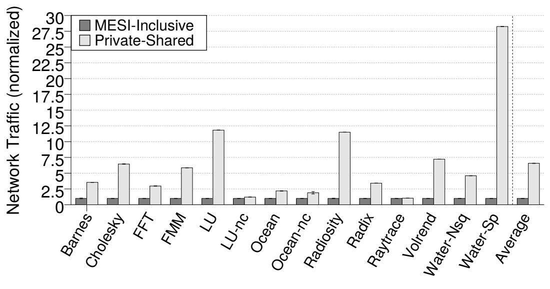


(b) Network traffic

Figure 3. Performance results for a 2D-mesh network.



(a) Execution time



(b) Network traffic

Figure 4. Performance results for a point-to-point network.

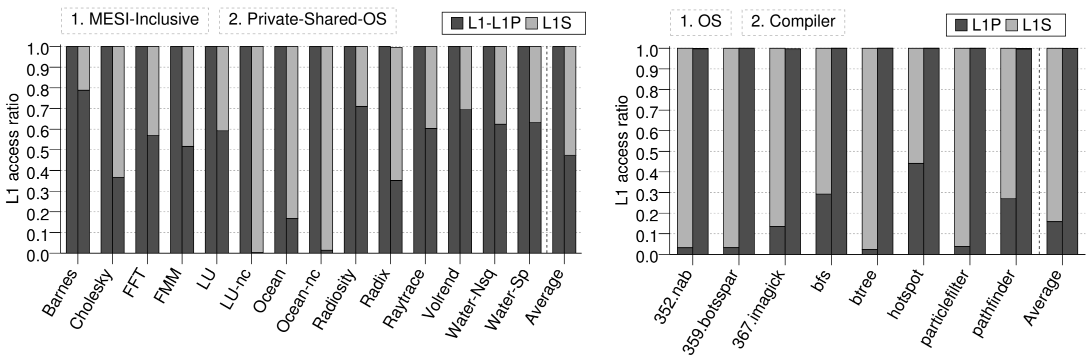
with other cores also in a MESI implementation. The reason for this slowdown is the imprecise OS-based classification, marking as shared accesses that are actually private.

Figure 3b shows the normalized number of flits sent along the interconnection network. There is also a significant increase in the network usage due to the introduction of the shared dedicated cache (around 13$\times$, on average). Water-Sp reaches an increment of 57$\times$ more network traffic than the unified L1 design, while other benchmarks, like Ocean, experience an increment of around 4$\times$, but both result in similar performance degradation. This supports our idea that the performance effects caused by network traffic are only critical for those applications that have shared accesses on their critical path of execution, at least when the shared nature of the accesses is detected by an OS-based classification.

4.2.2. Point-to-point interconnect. Figure 4 offers performance numbers for an idealized organization with a point-to-point network with one-cycle link latency normalized to MESI. Although this network provides low latency and high-bandwidth, this topology does not scale with the number of nodes in the system. Figure 4a shows a performance degradation of 25%, on average, compared to an unified cache design. Regarding the network traffic (Figure 4b), we observe similar trends as with the 2D-mesh network, although the absolute numbers differ. Despite the memory latency for shared blocks is reduced with this network topology, it is still increased for a large number of accesses, some of which, perhaps, have been misclassified as shared. In an attempt to mitigate this problem, next sections analyze the private-shared classification ratio of the OS-based mechanism and compare it to more accurate classification techniques, such as compiler-assisted ones.

4.3. *Impact of the classification mechanism*

In the previous sections, we have analyzed the impact of the interconnection network on the performance of our proposal considering an OS-based classification. The next step in our analysis is



(a) No classification vs. OS-based (b) OS-based vs. Compiler-assisted

Figure 5. Private/Shared ratio for L1 cache accesses.

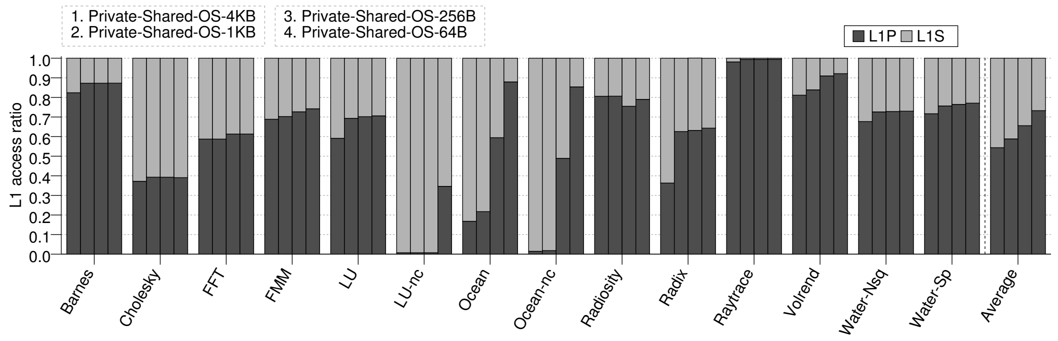


Figure 6. OS-based private-readonly with different page sizes.

to compare the ratio of private and shared accesses of this classification mechanism and a compiler-assisted one.

4.3.1. OS-based classification. In a MESI implementation, all memory accesses perform a look-up in the L1 cache. In the dedicated private-shared architecture accesses go to the corresponding L1 cache (L1P or L1S). Figure 5a shows the distribution of accesses to the L1P (only private data), served locally, and to the L1S, served through the interconnection network. It is clear that the cause of the large increase in execution time and network traffic for the dedicated cache architecture is the large fraction of shared accesses due to the lack of precision of the OS-based classification. Our current implementation of the OS-based classification marks more than half of the memory accesses as shared, thus requiring an access to the L1S through the interconnection network. In fact, for applications as LU-nc and Ocean-nc more than 98% of the accesses target the L1S. This is a surprising result since previous studies, such as SWEL [11], report that less than 10% of the memory accesses correspond to shared addresses, although that study was performed using a block-level classification, which is not suitable for our approach, as previously mentioned. To emulate its behavior, we performed a study with different granularities for the page-based classificaton, from 4 KB to 64 bytes (i.e., the block size) depicted in Figure 6. The precision of the OS-based classification improves with smaller page sizes, but even with page sizes equal to the block size and classifying both private and shared-read-only as private, it only achieves to classify approximately 74% of the accesses as private.

For OpenMP applications (Figure 5b), we can observe that the accuracy of the operating system in classifying accesses is even lower. The main reason for this low accuracy is the way OpenMP applications are parallelized. The iterations of a parallel `for` loop by default can be executed in parallel on any core. Furthermore, the workload distribution can vary from one loop to another. Namely, future parallel loops iterating over the same array might distribute differently the iterations to threads. Consequently, the iterated array becomes shared, and all future accesses to the array will be served through the L1S cache.

The conclusion of this analysis is that the OS-based classification is not precise enough for our proposal. The reason why this technique works for other proposals is that they are commonly targeting optimizations that get advantage of a large number of private blocks, but not private accesses. It is common in parallel applications to find a small amount of highly accessed (OS-)shared blocks. Although adaptive TLB-techniques [10] can improve the fraction of private accesses detected, it can only rise it up to 79%. Perhaps with a combination of adaptivity and low granularity, the accuracy in detecting private accesses can be improved up to the point that makes our architecture effective. We plan to analyze how to achieve higher accuracy with a low-overhead mechanism as part of our future work.

4.3.2. Compiler-assisted classification. The second classification technique that we evaluate is a compiler-assisted approach. We focus in a recent proposal that targets OpenMP codes, SPEL [8]. The major advantage of this proposal is that it takes advantage of knowledge about the programming model. There are, however, two disadvantages. The first one is that it is limited to data race free codes (e.g. OpenMP). The second one is that a certain data block can be accessed as private at some code region and as shared in another. This entails extra (local) coherence actions, that may require evicting blocks from the L1P when changing from a private (DRF) to a shared (non-DRF) region, or evicting blocks from the L1S when transitioning from a shared region to a private one. This overhead is not noticeable in an OS-based classification, since pages change at most once during their life in physical memory.

Figure 5b shows, however, that the advantages of the compiler-assisted classification can offset its drawbacks. Using this classification, most accesses are performed in the faster L1P. Only synchronization accesses, which inherently require communication, are performed through the L1S.

As a result, we can conclude that this classification, can be a good candidate for our architecture, because it detects a large number of private blocks, and performs this detection before the access to L1 (a-priori). There are, however, some important challenges that must be addressed in order to adapt this technique to our proposal. First, the dedicated cache architecture requires modifications to guarantee that stale data is not accessed. Second, the classification should be extended to more general programming paradigms, in order to obtain performance benefits for a large range of applications.

5. RELATED WORK

As mentioned in the introduction, there are a number of proposals in the recent literature that exploit the nature of data being accessed with the goal of reducing both complexity and on-chip network traffic. In this section we extend this description, and point out the differences with respect to our work.

Some authors use data classification to mitigate the growing access latency in NUCA architectures as core count increases. Private blocks can be placed in cache banks near the requesting core, thus reducing access latency and improving overall performance [25, 3, 4, 5]. Our architecture inherently retain this property of holding private data in the cache banks of the core that reference them. The use of several levels of private caches (L1P, L2P, etc.) would allow reducing even more the latency for private data.

Snooping protocols offer great simplicity but have scalability problems with high number of cores due to the enormous energy consumption required. The reason for this energy consumption is the issue of broadcast messages to all cores in the system on every write miss. The scalability of such protocols, in particular token-based protocols, was improved in the *Subspace Snooping* approach [6] by taking advantage of a private-shared classification and eliminating the need of broadcasts for private blocks. We chose a radically different approach, by completely eliminating the invalidation messages in the protocol due to write misses.

Directory-based protocols can likewise benefit from data classification. Spatio-Temporal Coherence [13] tries to find large private regions in order to compress the directory information and save space. Other works [7, 1, 2] propose to deactivate coherence for data not requiring it, which prevents directory caches from tracking private blocks, thus reducing both directory occupancy and access latency. Our architecture completely removes the need of a directory structure, thus eliminating the scalability problem of the directory.

SWEL [11] performs a private-shared classification at a block level (directory) and maintains shared read-write blocks at the shared last level cache. However, it incurs in some performance penalty due to the additional latency when accessing shared read-write blocks, since they skip the first levels of the cache hierarchy (similarly to our proposal). The main difference with our approach is that they handle the classification at a directory level (with block granularity) while we classify at a OS level (page granularity). Therefore, they require a directory cache to classify data, and coherence messages to invalidate or forward locally cached data.

VIPS [26] achieve both simplicity and efficiency by employing a write-back policy for private store operations and a write-through policy for shared store operations. While VIPS still employs a directory cache, the VIPS-M version is able, like us, to completely remove the directory. However, this is achieved at the cost of allowing incoherent data between synchronization points, resulting in a weaker consistency model (SC-for-DRF). Our memory hierarchy is fully coherent, so the memory model implemented by the processors is respected.

End-to-end SC [27] improves the efficiency of multi-cores by not enforcing sequential consistency for thread-local (private) or read-only accesses. Our proposal is orthogonal to this technique.

SPEL [8] employs a compiler-assisted classification of regions of code. The classification is used to implement a coherence protocol with two different modes: one for (extended-)data-race-free accesses, which enforces coherence only at the end of (extended-)data-race-free regions, the other for racy accesses which maintain coherence with a directory-based protocol. Again, the main difference with respect to SPEL is that our proposal removes the directory and the traffic related to coherence.

Finally, there are working architectures that implement dedicated caches/memories that are handled by either the compiler or the programmer (e.g., scratchpad memories or NVIDIA Private/Shared L1). To the best of our knowledge, latest NVIDIA architectures implement three memory spaces: a thread-private (usually write-through) and a block-shared memory space per SM[‡] plus a global address space for all SMs. PTX 2.0 and onwards uses a unified address space that combines all of the aforementioned memories, but global coherency is not maintained by the hardware (Pascal architecture may include this feature). The Kepler architecture provides a reconfigurable shared memory + L1 cache (16+48, 32+32 or 48+16 KB) per streaming processor. On the other hand, the Maxwell architecture provides physically separated shared memory (64–96 KB) and L1 cache (12–48 KB). The main differences between our dedicated design with the NVIDIA architecture is that the programmer has handle private (p_local) and shared (p_shared for sharing within an SM and p_global for among SMs) accesses, while ours is transparent to the programmer. In addition, coherency is not ensured by the hardware (at least with current NVIDIA architectures), while our design ensures coherency.

6. CONCLUSION

In this work we analyze a dedicated cache design that takes into consideration the nature of the data being accessed. Our proposal relies on the information provided by a classification mechanism that divides memory accesses into: a) private or read-only shared (sent to a local private caches) and b) modifiable shared (sent to a logically shared but distributed cache). This design would

[‡]Streaming Multiprocessor.

allow to simplify coherence protocols and core design, easing the validation process and improving scalability.

Our results show two main drawbacks that may limit the usability of our implementation: a) low accuracy on the classification mechanism and b) high latency in the network connecting the first-level shared caches. Improved classification mechanisms, like adaptive TLB-based approaches [10] or compiler-assisted approaches [8], can improve the accuracy of the classification, leading to fewer shared accesses. Still, these mechanisms have limitations, such as increasing complexity and traffic, or not being applicable to all programming models. Further research to mitigate this limitations seems a promising path to improve the results of proposals like ours.

On the other hand, there is a huge gap in the average access latency between our non-scalable point-to-point network and a more scalable but with larger latency 2D-mesh. We believe that there is also room for improvement by using a specialized network since our design only requires 8 bytes to be sent as control/request plus 16 bytes to be returned (control + data word).

With improved accuracy for access classification and reduced network latency, we believe that our approach can become useful as the number of cores increases and coherence protocols face more scalability issues. Additional research is needed to discover the number of cores required to make the dedicated design feasible in terms of performance. Throughput oriented cores (GPUs or accelerators like the Xeon Phi) can also benefit from our design, since additional memory latency is usually hidden in those systems by swapping execution threads. This encourages us to continue this line of research and propose alternative solutions that take into account the nature of the data that is being accessed in order to avoid unnecessary coherence operations.

ACKNOWLEDGEMENT

This work was supported by *Fundación Séneca-Agencia de Ciencia y Tecnología de la Región de Murcia* under the project *Jóvenes Líderes en Investigación "18956/JLI/13"*.

REFERENCES

1. Cuesta B, Ros A, Gómez ME, Robles A, Duato J. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. *38th Int'l Symp. on Computer Architecture (ISCA)*, 2011; 93–103.
2. Cuesta B, Ros A, Gómez ME, Robles A, Duato J. Increasing the effectiveness of directory caches by avoiding the tracking of non-coherent memory blocks. *IEEE Transactions on Computers (TC)* Mar 2013; **62**(3):482–495.
3. Hardavellas N, Ferdman M, Falsafi B, Ailamaki A. Reactive NUCA: Near-optimal block placement and replication in distributed caches. *36th Int'l Symp. on Computer Architecture (ISCA)*, 2009; 184–195.
4. Li Y, Abousamra A, Melhem R, Jones AK. Compiler-assisted data distribution for chip multiprocessors. *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2010; 501–512.
5. Li Y, Melhem RG, Jones AK. Practically private: Enabling high performance cmps through compiler-assisted data classification. *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2012; 231–240.
6. Kim D, Ahn J, Kim J, Huh J. Subspace snooping: Filtering snoops with operating system support. *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2010; 111–122.
7. Meng J, Skadron K. Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling. *Int'l Conf. on Computer Design (ICCD)*, 2009; 282–288.
8. Ros A, Jimborean A. A dual-consistency cache coherence protocol. *29th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2015; 1119–1128.
9. Ros A, Cuesta B, Gómez ME, Robles A, Duato J. Temporal-aware mechanism to detect private data in chip multiprocessors. *42nd Int'l Conf. on Parallel Processing (ICPP)*, 2013; 562–571.
10. Esteve A, Ros A, Gómez ME, Robles A, Duato J. Efficient tlb-based detection of private pages in chip multiprocessors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* Mar 2015; .
11. Pugsley SH, Spjut JB, Nellans DW, Balasubramonian R. SWEL: Hardware cache coherence protocols to map shared data onto shared caches. *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2010; 465–476.
12. Hossain H, Dwarkadas S, Huang MC. POPS: Coherence protocol optimization for both private and shared data. *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2011; 45–55.
13. Zebchuk J, Falsafi B, Moshovos A. Multi-grain coherence directories. *46th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, 2013; 359–370.
14. Valls JJ, Ros A, Sahuquillo J, Gómez ME, Duato J. PS-Dir: A scalable two-level directory cache. *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2012; 451–452.
15. Valls JJ, Ros A, Sahuquillo J, Gómez ME. PS directory: A scalable multilevel directory cache for CMPs Aug 2015; **71**(8):2847–2876.

16. Davari M, Ros A, Hagersten E, Kaxiras S. The effects of granularity and adaptivity on private/shared classification for coherence. *ACM Transactions on Architecture and Code Optimization (TACO)* Aug 2015; **12**(3):26:1–26:21.
17. Davari M, Ros A, Hagersten E, Kaxiras S. An efficient, self-contained, on-chip, directory: DIR$_1$-SISD. *24th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2015; 317–330.
18. Martin MM, Sorin DJ, Beckmann BM, Marty MR, Xu M, Alameldeen AR, Moore KE, Hill MD, Wood DA. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News* Sep 2005; **33**(4):92–99.
19. Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: Building customized program analysis tools with dynamic instrumentation. *2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2005; 190–200.
20. Monchiero M, Ahn JH, Falcón A, Ortega D, Faraboschi P. How to simulate 1000 cores. *Computer Architecture News* Jul 2009; **37**(2):10–19.
21. Carlson TE, Heirman W, Eeckhout L. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. *ACM/IEEE Conf. on Supercomputing (SC)*, 2011; 52:1–52:12.
22. Woo SC, Ohara M, Torrie E, Singh JP, Gupta A. The SPLASH-2 programs: Characterization and methodological considerations. *22nd Int'l Symp. on Computer Architecture (ISCA)*, 1995; 24–36.
23. Standard Performance Evaluation Corporation. SPEC OMP2012. http://www.spec.org/omp2012. URL http://www.spec.org/omp2012.
24. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee SH, Skadron K. Rodinia: A benchmark suite for heterogeneous computing. *Int'l Symp. on Workload Characterization (IISWC)*, 2009; 44–54.
25. Cho S, Jin L. Managing distributed, shared L2 caches through OS-level page allocation. *39th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, 2006; 455–465.
26. Ros A, Kaxiras S. Complexity-effective multicore coherence. *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2012; 241–252.
27. Singh A, Narayanasamy S, Marino D, Millstein T, Musuvathi M. End-to-end sequential consistency. *39th Int'l Symp. on Computer Architecture (ISCA)*, 2012; 524–535.