Early Experiences with Separate Caches for Private and Shared Data

Juan M. Cebrián, Alberto Ros, Ricardo Fernández-Pascual and Manuel E. Acacio
University of Murcia
Murcia, Spain
{*jcebrian, aros, rfernandez, meacacio*}*@ditec.um.es*

***Abstract*—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 blindly keeping coherence for all memory accesses can be counterproductive, since it incurs in unnecessary overhead for data that will remain coherent after the access (i.e., private and read-only shared data).**

Having this in mind, in this paper we propose the use of dedicated caches for private (+shared read-only) and shared data. The private cache (L1P) will be independent for each core while the shared cache (L1S) will be logically shared but physically distributed for all cores. This separation should allow us to simplify the coherence protocol, reduce the on-chip area requirements and reduce invalidation time with minimal impact on performance. The dedicated cache design requires a classification mechanism to detect private and shared data. In our evaluation we will use a classification mechanism that operates at the operating system (OS) level (page granularity). Results show two drawbacks to this approach: first, the selected classification mechanism has too many false positives, thus becoming an important limiting factor. Second, a traditional interconnection network is not optimal for accessing the L1S, and a custom network design is needed. These drawbacks lead to important performance degradation due to the additional latency when accessing the shared data.

1. Introduction

Modern chip multi-processors feature a mix of latency and throughput oriented cores running both sequential, multiprogrammed and multithreaded applications. These architectures usually share a common memory space that is kept coherent at a hardware level. While coherence and memory consistency are key aspects to ensure the correct execution of the applications on the system, they usually incur in a performance overhead proportional to the size of the system (i.e., number of cores, interconnection characteristics, etc). In addition, the implementation details of the coherence protocol have a huge impact on the scalability of the system, not only in terms of network traffic, but also area and energy requirements.

Applications show a wide range of data sharing degrees[1], from constant data exchange to independent behavior along threads/tasks. This property is mainly influenced by the programming methodology and the nature of the application. However, coherence protocols maintain coherence in the same way for all data accesses, regardless of the nature of the data that is being accessed. Therefore, blindly doing the same for all memory accesses can be counterproductive. The coherence protocol incurs in unnecessary overhead generating coherence messages and data transferences for accesses to addresses that would remain coherent anyway (i.e., private and read-only shared data).

In this work we aim to simplify the coherence protocol by taking advantage of the nature of the data that is being accessed. We will rely on the usage of multiple dedicated caches: a private cache (L1P) and a shared cache (L1S). The L1S may include different hierarchy levels. The private cache will store only private and shared read-only data. Coherence messages for this data are only required when its nature changes from private to shared or vice versa. On the other hand, the shared cache will be logically shared but distributed for all cores. In this case, no coherence actions are needed because there is only one copy of each data item. We explore several layout designs both using a centralized L1S and a tiled design (each tile stores a piece of the L1S).

Our proposal requires a classification mechanism that detects private and shared data. There are many classification mechanisms in the literature that work at different levels: compiler, OS, coherence protocol, etc. We chose one mechanism used in several recent works [1], [2], [3] because of its simplicity. This mechanism operates at the OS level (at a page granularity). When a core accesses a page it is marked private in the page table. A second access to a page (marked as private) by a different core will force an interruption of the first core to update the table state to shared.

With the dedicated cache design we expect to simplify the coherence protocol and to reduce the overhead caused by the coherence protocol on the interconnection network, improving scalability with minimal performance degradation. We will guide the readers through the design process highlighting the benefits of the proposal as well

1. Fraction of shared data with regard to the total amount of data.

as the problems that lead us to negative results. The main contributions of this paper include:

- A dedicated cache design for chip multiprocessors that exploits the nature of the data accessed.
- Different layout implementations of the solution (including a tiled layout, a tiled layout using a point to point network and a custom layout design).
- Evaluation of the proposal in terms of performance, network traffic and cache miss ratio.
- Discussion on the design flaws of the proposal and specific scenarios where it works.

The benefits and drawbacks of this design are the following:

✓ Eliminates the need for a coherence directory, improving scalability of the multi-core architecture.
✓ Reduces the pressure on the L1P (also reduces overall combined L1D[2] cache misses).
✓ Reduces the amount of duplicated data in L1 caches.
✗ Increased latency when accessing the shared data (requires using the network). This network may require high bandwidth.
✗ Requires a classification mechanism to detect the nature of the data.

The rest of the paper is organized as follows: Section 2 describes other solutions that exploit the nature of the data that can be found in the literature. Section 3 describes the implementation details of our proposal. Section 4 explains the evaluation environment, and provides details of the analyzed processor configuration. Section 5 discusses the main results of our research, before concluding with Section 6 and suggesting directions for further research.

2. Related Work

There are several 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.

Snooping protocols offer great simplicity but have scalability problems with high number of cores. The scalability of such protocols can be improved by taking advantage of a private-shared classification, as proposed by Kim et al. [4]. This proposal eliminates the need of broadcasts for private blocks. Other papers use data classification to mitigate the growing access latency in NUCA[3] architectures as core count increases. Private blocks can be placed in cache banks near the requesting core, reducing access latency and improving overall performance [3], [5], [6], [7].

On the other hand, directory-based protocols can likewise benefit from data classification. Alisafaee et al. [8] try to find large private regions in order to compress the directory information and save space. Other works [2], [9] 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. SWEL [10] 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, but requiring a coherence protocol) while we classify at a OS level (page granularity).

Regarding the classification mechanisms, this paper relies on the approach proposed by Cuesta et al. [2], used in several works published in international conferences [1], [11], [12]. The mechanism uses an additional bit on the page table to keep the state of the page (private or shared). The P/S bit is also included in the TLB[4] entries to speed up the access to the classification information. OS-based classification mechanisms mark pages as private the first time they are accessed after a TLB miss. On each access, the classification mechanism looks for a match between the *keeper* field and the current requestor. If they differ, the page is marked as shared. The benefit of OS-based mechanisms is that they require minimal hardware support, relying mainly on the page table and TLBs to classify data. This mechanism reduces the area requirements when compared with other proposals that operate either at a block-level [8], [10], [13]. Alternative virtual memory level classification mechanisms have been proposed in the literature [3], [4], [14]. Finally, compiler-assisted approaches [6], [7], [9] have only limited knowledge on what data is going to be shared and where it will be processed (core), reducing the accuracy of the prediction, and thus being less attractive for our proposal.

3. Proposal

In this paper we propose a dedicated cache design that uses a private L1 cache (L1P) to store both private and shared read-only data and a logically shared but physically distributed L1 (L1S) to store shared data. The L1P works much as a regular private L1 data cache. On the other hand, the L1S works like a shared L2 in a tiled chip multiprocessor (CMP), where each tile provides a bank for the global L1S/L2. Coherence of shared addresses may require an access through the interconnection network to the home tile (L1S bank) that holds the data. The mapping from data to banks is based on the lowest bits of the memory address (like the shared L2). The selected classification mechanism requires an additional bit that is included in the TLB entries (and memory pages) to indicate the Private/Shared (PS bit) status of the page.

When a core executes a memory instruction (e.g., load or store), it reads the PS bit of the accessed page in the TLB and uses it to decide if the memory request should be sent to the L1P or forwarded to the L1S (Figure 1). The L1S access latency would consist of three components:

2. Data L1.
3. Non-uniform Cache Architecture
4. Translation Lookaside Buffer.

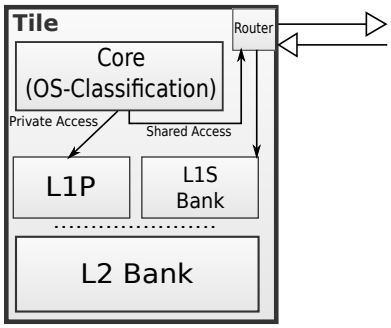

Figure 1. Dedicated cache design diagram.

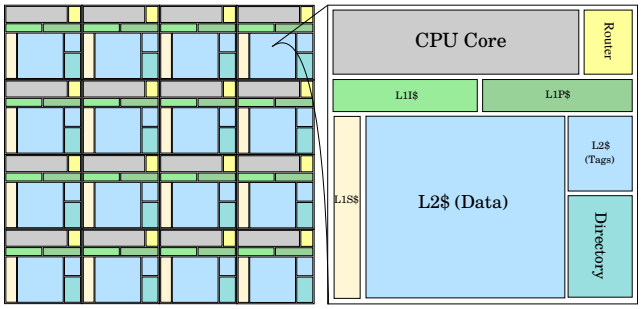

Figure 2. Layout example for a 16-core tiled design.

1) request/response operation to/from the network buffer of the tile, 2) transfer the request/data along the network and 3) the memory array access (e.g., *2 + Network + 4* cycles in our evaluation). An 8-byte request message is sent to the appropriate L1S bank and a 8+8-byte response message (control+data word) is sent back to the network controller of the requesting tile. The main benefits of this design include a reduction on protocol complexity and on-chip area requirements and saving waiting time for L1D cache invalidations of blocks with multiple sharers. The main drawback of this multi-cache design is the extra latency of accessing the L1S, which is not local to the core. Accessing the L1S requires data to be transmitted through the interconnect, posing a threat for the potential benefits of the proposal.

4. Evaluation Methodology

The proposed dedicated cache design has been evaluated using the PIN [15] toolset and GEMS 2.1 [16] simulator. Real system data accesses captured by PIN are used as input for the simulation environment based on GEMS (similarly to [17]). GEMS includes a detailed memory hierarchy model (Ruby) that allows us to estimate performance, access latencies, miss rates and other useful statistics. The interconnection network is modeled using the Garnet [18] simulator for the tiled and custom layout designs (Sections 5.1 and 5.2). The "simple" network model that comes with GEMS, an idealized point to point (P2P) network that does not model contention, is also used in our idealized evaluation (Section 5.3). The simulated architecture corresponds to a single chip multiprocessor (*tiled*-CMP) featuring 16 cores (depicted in Figure 2). The most relevant simulation parameters are shown in Table 1.

The evaluation of the dedicated cache design is performed against a traditional MESI directory-based coherence protocol implemented in GEMS (*MESI-Inclusive* in our figures). We evaluate our proposal using the applications from the SPLASH-2 benchmark suite with the recommended input sizes [19]. Variability in the applications is accounted by introducing random variations in each main memory access on different runs [20].

TABLE 1. SYSTEM PARAMETERS.

Memory parameters	
Block size	64 bytes
L1 cache (data & private & instr.)	32 KB, 4 ways
L1 access latency (data & private & instr.)	4 cycle
L1S (shared)	(32 KB, 4 ways) per tile
L1S access latency	4 + Network
L2 cache (shared)	512 KB/tile, 16 ways
L2 access latency	12 cycle
Cache organization	Inclusive
Directory information	Included in L2
Memory access time	160 cycles
Network parameters	
Topology	Base: 2-D mesh (4×4) Other: Custom/Simple Network
Routing method	X-Y determinist
Message size	5 flits (data), 1 flit (control)
Link time	1 cycle
Bandwidth	1 flit per cycle

5. Results

This section summarizes our main results for different implementations of the dedicated cache design.

5.1. Traditional 2D Mesh Tiled Layout

The first implementation to be evaluated is based on a 2D mesh tiled network design that models contention. This layout is one of the most realistic designs since it provides an scalable solution, reducing design and validation complexity. Figure 3 shows the normalized runtime of the tiled dedicated cache design against an standard MESI implementation. We expected minimal performance impact with the dedicated cache design (or even a slight speedup due to reduction on the time spent during invalidations). However, we can clearly see a huge 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. This is most likely because shared accesses are located in the critical path of the application. Additional hardware/software mechanisms could be used to detect and reorder critical shared accesses to minimize the latency penalty of the L1S.

In the end, despite the expected benefits of the dedicated cache design in complexity, area and invalidation time,

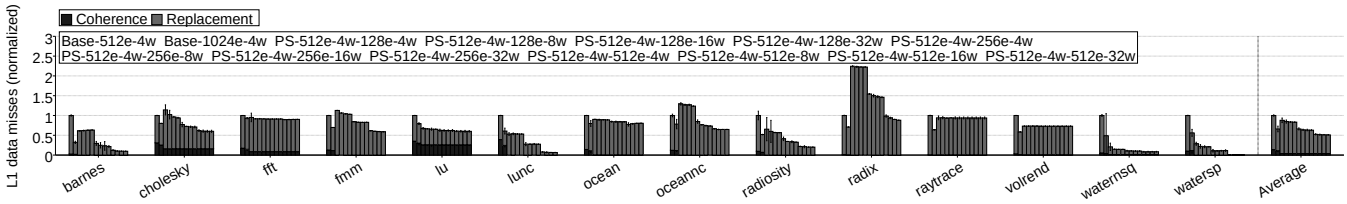

Figure 4. Normalized L1P+L1S cache misses (to unified/MESI L1D) for different cache configurations (sets and ways). 16 cores. Tiled design, 2D mesh garnet network. CLH locks. Entries (e) equal sets*ways (w). (e.g., *PS-512e-4w-256e-8w* represents the dedicated cache design with a private cache of 512 entries and 4 ways and a shared cache of 256 entries and 8 ways).

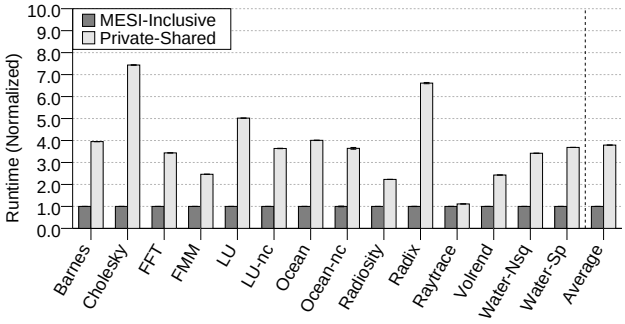

Figure 3. Normalized runtime (to unified/MESI L1D). 16 cores. Tiled design, 2D mesh garnet network. CLH locks.

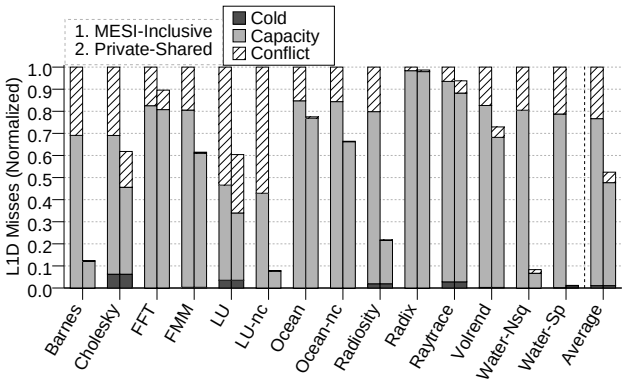

Figure 5. Normalized L1P+L1S cache misses (to unified/MESI L1D). 16 cores. Tiled design, 2D mesh garnet network. CLH locks.

the additional latency from the network interconnect along with other factors makes this design unfeasible for this benchmark set. However, this behavior differs from similar proposals that exploit the nature of the data being accessed, like SWEL [10], where the impact of avoiding the first levels of the memory hierarchy and accessing the network does not translate into any noticeable performance degradation. The next step in our quest was to discover why we experienced this behavior while others did not, and how we could solve this issue.

5.1.1. Effects of Dedicated Caches on Memory Accesses. We started by analyzing the effects on cache accesses when splitting data into two dedicated caches. This analysis was performed in terms of cache accesses and misses in order to discard other issues with the memory hierarchy. Since we are doubling the potential capacity of the L1D and reducing the number of replicated data in the L1S, we expected a significant reduction on the cache misses. This premise was validated by the results as shown in Figure 5. This figure shows the normalized misses of the dedicated cache design compared with the original design. In addition, while the slowdown of some of the applications (e.g., Cholesky) matches a high number of L1D cache misses (L1P+L1S), other applications (e.g., Barnes and Water-Sp) experienced an important reduction on the number of misses, but still performed worse than the original design in terms of performance. We can conclude that the extra combined capacity of the L1D (L1P+L1S) causes a significant reduction on the number of cache misses, but the extra latency of the L1S makes shared accesses to cost as much as misses on the private cache, so performance benefits are minimal. This leads us to believe that the major issues in our proposal should be elsewhere.

Furthermore, we also tested different configurations for the L1S varying the number of sets as well as the associativity of the cache (Figure 4). Results show barely any benefit from increasing the associativity of the L1S any further than four ways, but also that the number of entries (e = sets $\times$ ways) should remain above 256 (per tile). In addition, using a separate cache design outperforms the unified design when it doubles its capacity (*PS-512e-4w-512e-4w* vs *Base-1024e-4w*). 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. For the remaining of the article we will limit our study to a 4-way 512-entry L1S cache (per tile).

5.1.2. Network Traffic. Our next step was to check the network traffic of our implementation. Figure 6 shows the normalized number of flits sent along the interconnection network. We can clearly see a huge increment on the network usage as a result of the introduction of the shared dedicated cache (around 13$\times$ on average). It is important to note that the Water-Sp benchmark reaches an increment of 57 times more network traffic than the unified L1D

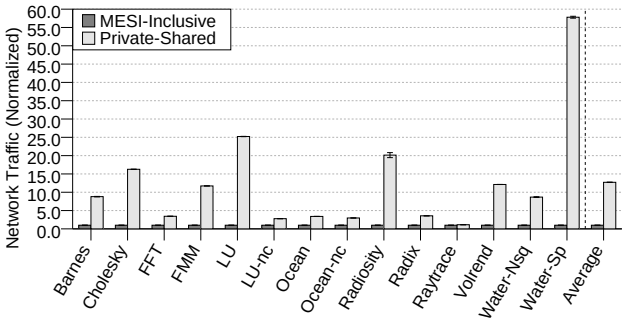

Figure 6. Normalized network traffic (to unified/MESI L1D). 16 cores. Tiled design, 2D mesh garnet network. CLH locks.

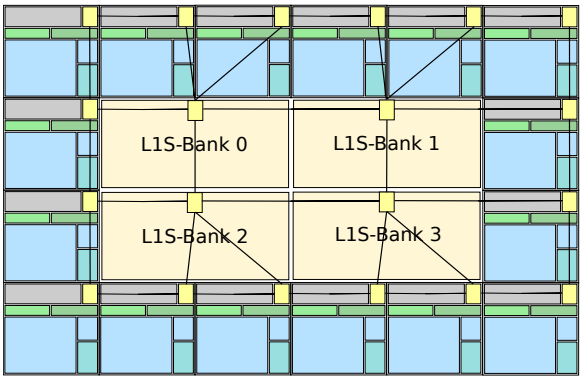

Figure 7. Four bank L1S layout design for a 16-core CMP.

design, while other benchmarks, like Ocean, experience an increment of around 4×, but both experience similar performance degradation. This supports our idea that the performance effects on network traffic are only critical for those applications that have shared accesses in their critical path of execution. The increment in network traffic translates into a noticeable performance degradation, given that any access to the L1S has an average latency of around 46 cycles[5] for our tiled 2D mesh configuration (as compared to the 6 cycle latency for the original unified L1D design, including misses). In addition, L1S misses for the dedicated cache design have an average of 120 cycle latency (same as the unified L1D design). So, even if we have less misses, we may not be able to hide this extra access latency by exploiting memory level parallelism (MLP) if the application does not provide enough outstanding misses.

5.2. Custom Layout (Centralized L1S)

At this point, we decided to check a different layout configuration, where all the L1S banks are placed equidistant to all tiles (right in the middle of the CMP processor). This design divides the L1S in four banks, each with a routing component. The interconnect is formed by connecting the router in each tile either neighbor tile router or to the L1S bank, as depicted in Figure 7. This configuration should reduce the average number of hops to reach the L1S, and we expected some improvement in L1S access latency.

Figures 8 and 9 show both the normalized runtime and flits transmitted assuming this new layout. With this design, network traffic is slightly decreased (as well as hops, as expected), going down from 13× higher in the tiled design to 10×. However, performance of the centralized design behaves 3.9 times slower than the original unified cache design, and slightly worse than the tiled design (3.7×). Miss latencies increase slightly as compared to the tiled design, rising from 120 to 123 for private accesses and reduced from 120 to 118 for shared accesses. Shared accesses latencies that miss and write increase from 145 to 155 for this

5. Empirically obtained from our simulation environment.

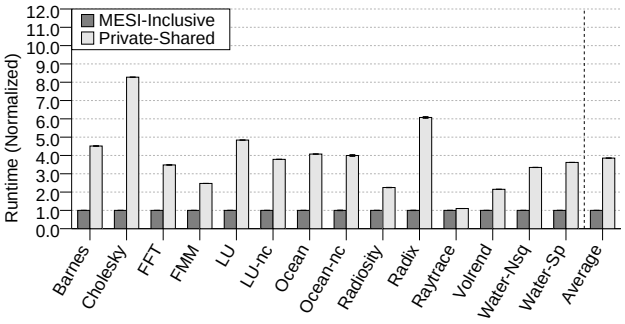

Figure 8. Normalized runtime (to unified/MESI L1D). 16 cores. Custom design, garnet network. CLH locks.

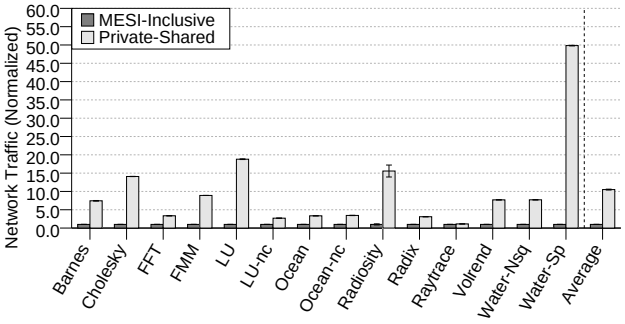

Figure 9. Normalized network traffic (to unified/MESI L1D). 16 cores. Custom design, garnet network. CLH locks.

design. This small latency variation happens because of the contention on the network when cores need to access the L1S. As a result, despite the decrease in network traffic, this design experiences slightly worse performance than the tiled design, and is definitely not worth implementing in real hardware. Increasing the number of banks and adding additional links may benefit this design, reducing the latency of both accesses and misses, but we did not explore this idea any further.

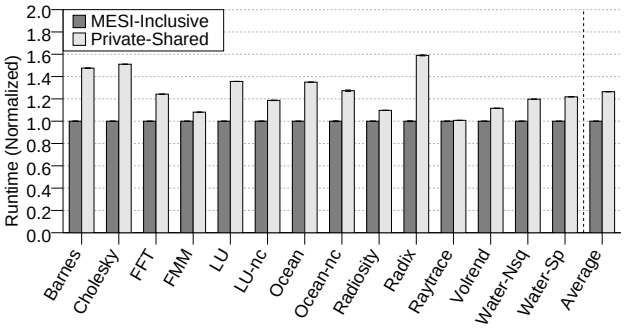

Figure 10. Normalized runtime (to unified/MESI L1D). 16 cores. Tiled design, simple P2P network. CLH locks.

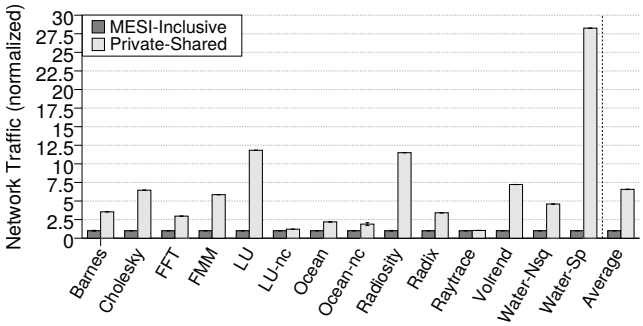

Figure 11. Normalized network traffic (to unified/MESI L1D). 16 cores. Tiled design, simple P2P network. CLH locks.

5.3. Idealized Network (Tiled - Point to Point)

Our next step was use the original tiled design assuming an incredibly fast network at our disposal, and analyze how the dedicated cache design would perform in this unrealistic scenario. This network, named *Simple* in our work, is a point to point network with one cycle latency. We modeled this network by using the default point to point network that comes with GEMS, where each core is connected to all L1S banks with a dedicated link.

By relying on a fast idealized network interconnect we expected barely no performance degradation for the dedicated cache design. However, as it can be seen in Figure 10, our design still suffers from 25% worse average performance than the unified cache design. Taking a look to the network traffic (Figure 11), we can observe similar trends to those of the previous interconnection networks that we have studied, though the absolute numbers differ.

By using this simple network, both the average hit and miss latencies are reduced considerably, dropping from from 45 to 8 and from 120 to 82 respectively when compared to the tiled design. Nevertheless, even for this optimistic configuration, the increase in network traffic for all the accesses to the L1S coupled with the additional latency of the interconnection network (compared with accessing a local cache) outweigh whatever savings we can achieve by simplifying the coherence protocol.

5.4. Discussion

Through Section 5 we have tried to improve our design by altering the characteristics of the network interconnect, without thinking about the reason for this huge increment in network traffic. Figure 12 shows the distribution of accesses to the L1P (loads and stores), served locally, and to the L1S, sent along the network interconnect (forwards).

The first thing that caught our attention was an increase in the total number of accesses. Some of the analyzed applications rely on locks and barriers to access critical sections of the code. These lock variables are stored in shared memory addresses that need to be stored on the L1S. Unfortunately, this means that the spinning process is no longer done locally, and each core needs to access the L1S through the network to try to acquire the lock. This generates additional network accesses that are not present in the original design. This problem could be ameliorated by using better locking mechanisms that reduce busy waiting (e.g., G-locks [21]).

Figure 13 shows the same distribution of accesses but using *functional* locks, that is, without simulating the memory accesses that perform the active spinning. This eliminates the extra accesses shown in Figure 12. We can now clearly see what was causing this huge increment on the network traffic: the large fraction of shared accesses due to the lack of precision on the classification mechanism. Our current implementation of the OS/TLB-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 was a surprising result since previous studies, such as SWEL [10], report that less than 10% of the memory accesses correspond to shared addresses (using a block-level classification).

Nevertheless, this low accuracy of the classification mechanism has been corroborated by Esteve et al. [22]. The main reason being that pages marked as shared never return to their private state. In this work, the authors discuss an enhancement to the classification mechanism that restores the private nature of a memory page after a certain period of non-usage (decay time). This new method boosts the number of private accesses from 43% to 79% in their research study.

However, we are uncertain if this boost in accuracy would be enough for a dedicated cache design to work. In order to perform a study of the potential of our design, regardless of the private access rate, we performed a linear extrapolation (Figure 14) based on the access and miss latencies obtained empirically from our simulation environment, and shown in Table 2. This extrapolation is based on the following formula:

$$((Hits^{P} \times Lat.^{P.Hit}) + (Misses^{P} \times Lat.^{P.Miss})) + ((Hits^{S} \times Lat.^{S.Hit}) + (Misses^{S} \times Lat.^{S.Miss})) - (Miss^{Multiple.Sharers} \times Lat.^{Miss+Write})$$

In other words, the application will experience a delay based on the number of hits/misses of the private (P) and shared (S) caches and their associated latency (if they cannot be hidden by MLP/outstanding misses). On the other hand, there is going to be a speedup proportional to the number of invalidations that are no longer required in the dedicated cache design, since this design eliminates the need for the coherence protocol. This speedup would depend on the number of misses that happen to have several sharers multiplied by the time it takes to perform the invalidation and writing the data back to the first level of coherent unified memory (L2 in our case).

Figure 14 shows that, even for ratios of L1S accesses around 10% (e.g., achieved by using a block-level classification like the one in [10]), there is going to be a considerable performance degradation of the application (around 60%) if the extra latency cannot be hidden by MLP or reordering critical path accesses when using both custom and tiled designs. This performance degradation can go down to 4% using a P2P simple network. For a ratio of 50% shared accesses the model predicts a performance degradation of 4×, the same we obtained empirically.

Nevertheless, 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[6] 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 (64-96KB) 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. Finally, GPUs are throughput oriented architectures that can hide memory latency exchanging execution threads, while we have evaluated our approach on latency oriented architectures where we have seen that if suffers from a substantial performance degradation.

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:

6. Streaming Multiprocessor.

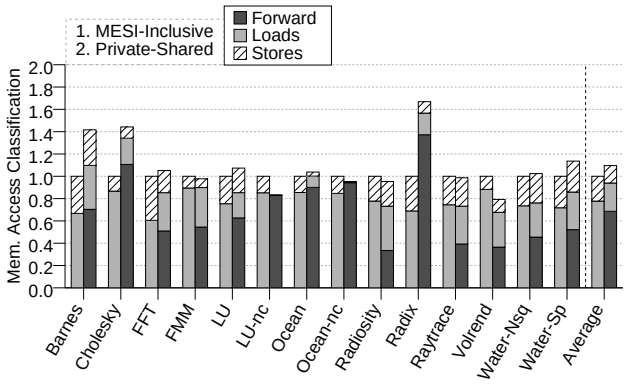

Figure 12. Private/Shared ratios. 16 core, garnet network. CLH locks.

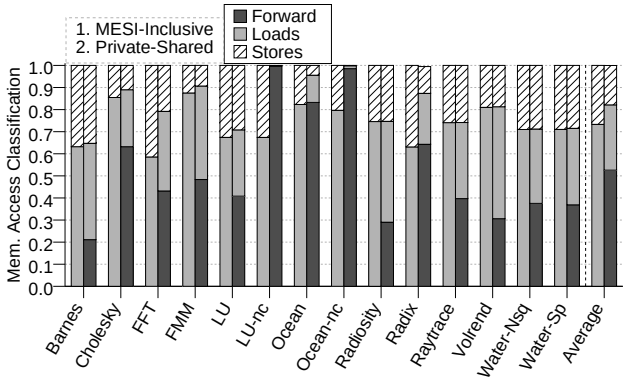

Figure 13. Private/Shared ratios. 16 core, garnet network. Functional locks.

a) private/shared read-only (sent to a local private caches) and b) shared (sent to a logically shared but distributed cache). This design would allow to simplify coherence protocols and core design, easing the validation process and improving scalability.

Our results show two main drawbacks that limit the usability of our implementation: a) low accuracy on the classification mechanism and b) huge increase of the latency of shared accesses (due to the interconnection network). Improved OS-Based mechanisms, like [22] or [**?**], can improve accuracy to 80+%. Even so, it still can be further improved, since other classification mechanisms that work at a virtual-

TABLE 2. L1S/P ACCESS LATENCIES FOR DIFFERENT NETWORK INTERCONNECTS.

	MESI-Tiled	**Tiled**	**Custom**	**Simple**
Private Hit Latency:	4	4	4	4
Private Miss Latency:	120	120	123	75
Shared Hit Latency:	-	45	45	8
Shared Miss Latency:	-	120	118	62
Shared Miss+Write Latency:	-	145	155	87

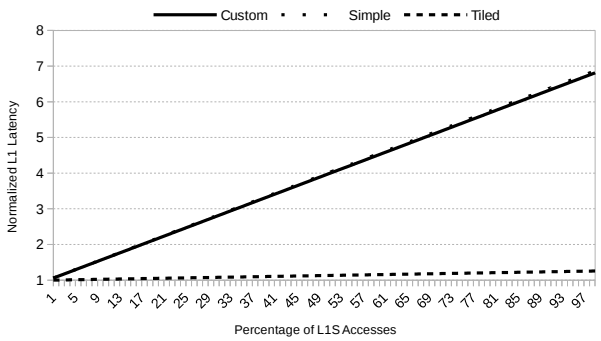

Figure 14. Private/Shared ratio extrapolation. 16 core. CLH locks.

memory level show less than 10% shared accesses. On the other hand, there is a huge gap in the average access latency between our idealized peer to peer simple network and a real world 2D mesh (from 6.75 to 23 cycles latency). We believe 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 clasisfication and reduced network latency, we believe our approach can become useful as the number of cores increases and coherence protocols face more scalability issues. This is true as long as the additional cores can balance for the performance degradation due to the dedicated caches. 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.

Acknowledgments

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] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 241–252.

[2] B. Cuesta *et al.*, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *38th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2011, pp. 93–103.

[3] N. Hardavellas *et al.*, "Reactive NUCA: Near-optimal block placement and replication in distributed caches," in *36th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2009, pp. 184–195.

[4] D. Kim, J. A. J. Kim, and J. Huh, "Subspace snooping: Filtering snoops with operating system support," in *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2010, pp. 111–122.

[5] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-level page allocation," in *39th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2006, pp. 455–465.

[6] Y. Li *et al.*, "Compiler-assisted data distribution for chip multiprocessors," in *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2010, pp. 501–512.

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

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

[9] J. Meng and K. Skadron, "Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling," in *Int'l Conf. on Computer Design (ICCD)*, Oct. 2009, pp. 282–288.

[10] S. H. Pugsley *et al.*, "SWEL: Hardware cache coherence protocols to map shared data onto shared caches," in *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2010, pp. 465–476.

[11] A. Ros *et al.*, "Temporal-aware mechanism to detect private data in chip multiprocessors," in *42nd Int'l Conf. on Parallel Processing (ICPP)*, Oct. 2013, pp. 562–571.

[12] S. Kaxiras and A. Ros, "A new perspective for efficient virtual-cache coherence," in *40th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2013, pp. 535–547.

[13] H. Hossain, S. Dwarkadas, and M. C. Huang, "POPS: Coherence protocol optimization for both private and shared data," in *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2011, pp. 45–55.

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

[15] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun. 2005, pp. 190–200.

[16] M. M. Martin *et al.*, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[17] M. Monchiero *et al.*, "How to simulate 1000 cores," *Computer Architecture News*, vol. 37, no. 2, pp. 10–19, Jul. 2009.

[18] N. Agarwal *et al.*, "GARNET: A detailed on-chip network model inside a full-system simulator," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.

[19] S. C. Woo *et al.*, "The SPLASH-2 programs: Characterization and methodological considerations," in *22nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1995, pp. 24–36.

[20] A. R. Alameldeen and D. A. Wood, "Variability in architectural simulations of multi-threaded workloads," in *9th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2003, pp. 7–18.

[21] J. Abellan, J. Fernandez, and M. Acacio, "Glocks: Efficient support for highly-contended locks in many-core cmps," in *24nd Int'l Parallel and Distributed Processing Symp. (IPDPS)*, May 2011, pp. 893–905.

[22] A. Esteve *et al.*, "Efficient tlb-based detection of private pages in chip multiprocessors," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2015.