UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

PS-Architecture: A scalable and energy-efficient architecture for CMP NUCAs

Tesina del Máster en Ingeniería de Computadores
Curso 2012/2013

Escuela Técnica Superior de Ingeniería Informática

Joan Josep Valls Mompó

Directores:
Julio Sahuquillo Borrás
María Engracia Gómez Requena

Julio de 2013

Acknowledgments

A mi familia, por todo el apoyo recibido y soportarme durante todos estos años.

A mis compañeros y amigos, por los buenos momentos que hemos compartido.

A Julio y María Engracia, por haber confiado en mí y haberme dado esta gran oportunidad.

A Alberto, por toda esa ayuda recibida desde el principio y, en general, por estar siempre al otro lado del correo y ayudarme a solucionar todos los problemas que han ido surgiendo.

A todos vosotros, gracias.

Contents

1 Introduction **9**

1.1 Problem Description 9

1.2 Non Uniform Cache Access (NUCA) 11

1.3 Coherence Protocols 12

1.3.1 MOESI Protocol 12

1.3.2 Update and Invalidation Protocols 13

1.3.3 Directory-based and Snoopy Protocols 15

1.4 Dealing with Scalability in Future CMPs 17

1.5 Work Structure 18

2 Related Work **19**

2.1 Directory Caches 19

2.1.1 Duplicate-tag and directories 19

2.1.2 Sparse directories 20

2.2 Cache Memories 21

2.2.1 Energy-efficient cache designs 21

2.2.2 Private-shared optimizations 22

3 PS Directory **24**

3.1 Base Architecture 24

3.2 Analyzing the Behavior of Private and Shared Blocks from the Directory Point of View 25

3.3 The PS Directory Scheme 28

4 PS Cache **32**

4.1 Accessing the Cache Hierarchy 32

4.2 The PS Cache Scheme 35

4.2.1 The PS Page Classification Mechanism 36

4.2.2 The PS Cache Architecture . . . 37

5 Experimental Framework 39

5.1 Simulation tools . . . 39

5.1.1 Simics-GEMS . . . 39

5.1.2 CACTI . . . 40

5.1.3 Simulated System . . . 40

5.2 Metrics and Methodology . . . 42

5.3 Benchmarks . . . 44

5.3.1 Barnes . . . 44

5.3.2 FFT . . . 45

5.3.3 Ocean . . . 45

5.3.4 Radiosity . . . 45

5.3.5 Radix . . . 46

5.3.6 Raytrace . . . 46

5.3.7 Volrend . . . 46

5.3.8 Water-Nsq . . . 47

5.3.9 Blackscholes . . . 47

5.3.10 Swaptions . . . 48

6 Experimental Evaluation 49

6.1 PS Directory Results . . . 49

6.1.1 Performance . . . 49

6.1.2 Area and Energy Analysis . . . 52

6.1.3 Directory Coverage Ratio Analysis . . . 53

6.2 PS Cache Results . . . 57

6.2.1 Private-Shared Blocks Behaviour Analysis . . . 57

6.2.2 Power Consumption . . . 60

7 Conclusions and Future Work 63

7.1 Conclusions . . . 63

7.2 Future Work . . . 64

7.3 Thesis-related Publications . . . 65

List of Figures

1.1 Evolution in the access times. 11
1.2 Average access time depending of the memory configuration. 12
1.3 State transition diagram for a MOESI protocol. 14
1.4 Example of how an update protocol works. 14
1.5 Example of how an invalidation protocol works. 15

3.1 Organization of the tile assumed in this work and a 4×4 tiled CMP. . . 25
3.2 Number of accesses to private and shared entries per kilo instruction in a conventional directory. 26
3.3 Number of evictions of private and shared entries per kiloinstruction in a conventional directory and its effect on performance. 26
3.4 Private-Shared directory organization. 29
3.5 Parallel access of the Shared cache and the NUCA cache. Private cache is only accessed on a miss in the Shared cache. 30
3.6 Directory controller flow-diagram. 30

4.1 Example of private-shared blocks distribution in a 8-way set cache. . . 33
4.2 Fraction of memory instructions. 34
4.3 Distribution of accesses to the L1 cache and to the L2 cache. 34
4.4 L1 Coherence lookups 35
4.5 The PS cache architecture. 36

6.1 Normalized misses with respect to a conventional single-cache directory. 50
6.2 Normalized execution time with respect to a perfect directory. 50
6.3 Normalized energy consumed by the directory with respect to a single-cache directory. eDRAM technology is used in the Private directory cache. 52
6.4 Scalability analysis in terms of area. 53

6.5 Normalized performance with respect to a conventional single-cache directory. . . . 54
6.6 Normalized energy consumed by the directory with respect to a single-cache directory. . . . 55
6.7 Scalability analysis in terms of area. . . . 57
6.8 Average number of blocks of each type in the L1 cache for both studied protocols. . . . 58
6.9 Distribution of the number of ways searched in the L1 cache normalized with respect to the snoopy protocol. . . . 59
6.10 Average state of ways during lookups in the L2. . . . 60
6.11 Static and dynamic energy comparison. . . . 60
6.12 Comparison of dynamic energy consumption in L1 and L2. . . . 61
6.13 Reduction of the dynamic energy consumption. . . . 61

List of Tables

1.1 Comparing Technological Features of SRAM versus eDRAM. 18

5.1 PS Cache System parameters 41

5.2 Directory latencies 42

6.1 Area (in $mm^2 * 1000$) of the different PS configurations for 16 cores compared with the Single cache directory. 52

6.2 Area (in $mm^2 * 1000$) of the different PS configurations for 16 cores compared with the $1\times$ Single cache directory. 54

6.3 Static and dynamic energy consumption of the different PS configurations for 16 cores compared with the $1\times$ Single cache directory. 55

Chapter 1

Introduction

Current and incoming chip–multiprocessor (CMP) present important constraints of power and area that must be addressed to sustain the performance. To this end, critical components must be identified and their design must be revisited in order to devise new efficient power-aware structures with no or minimal performance penalties, which is the focus and goal of this work.

To achieve this goal this work considers the different types (i.e. private and shared) of blocks in parallel workloads to devise i) an energy and area efficient directory cache, and ii) a power-aware first-level processor cache.

1.1 Problem Description

As the number of cores increases in both incoming and future shared-memory CMP generations, the cache hierarchy prevents processor performance to scale due to power and area design concerns. In this context, coherence protocols and first-level data caches are among the critical components in the cache hierarchy, whose design should be revisited to provide new memory structures to enable scalability.

From the memory coherence point of view, directory–based coherence is the commonly preferred approach over snoop–based coherence, since the former keeps track of cached blocks to avoid the use of broadcast messages. Two main design choices have been used in both research proposals [MH07, CRG+11] and commercial processors [CKD+10, BGM00, SBB07] to implement CMP directories: *Duplicate Tags* and *Sparse Directories*. Both approaches present different design concerns. Duplicate Tags do not scale for large systems due to the high energy consumed by the highly associative lookups, required to build the *sharer vector*. Sparse directories use a directory cache, that is implemented as a typical processor cache to keep track of the cached blocks.

This approach presents two main disadvantages: i) implements a large sharer vector that is expected to introduce important on–chip area and leakage overheads in future CMPs [ZSQM09], and ii) blocks in the processor cache must be invalidated when the associated directory entry is evicted. However, due to power reasons directory caches is the preferred approach and the focus of this work. This work proposes the PS directory that deals with the mentioned drawbacks of directory caches. For this purpose, a new design is presented, namely the PS directory, that reduces the area occupied by the sharer vector field, as well as the consumed energy. Moreover, the proposed design combines different RAM technologies (SRAM and eDRAM) to deal with both area and energy.

On the other hand, processor caches, typically organized in a two- or three-level cache hierarchy, occupy a significant percentage of the overall CMP die area [BJ11] and consume an important fraction of the overall energy budget. Dissipated energy has two main components: dynamic energy and static energy or leakage energy. The former is consumed when transistors switch their value while the latter is due to leakage currents.

Dynamic energy mainly concentrates in the first-level cache due to this level is the mostly accessed one. An important fraction of this energy is due tag comparisons on each cache access, that are typically accessed in parallel for performance in set-associative caches. Because of this reason, many research has been performed in the past concentrating in monolithic processors to tackle this problem. However, this research cannot be directly applied to CMPs.

This work presents a new cache scheme, namely the PS cache, to reduce the highly energy consumed due to tag comparisons. The proposed approach only looks up a subset of ways of the cache set. To this end, blocks in the different ways are tagged according to a given non-speculative criterion, and only those tags of the blocks matching the target criterion are compared.

The two main contributions of this work discussed above lay on the fact that private (P) and shared (S) blocks in parallel workloads exhibit different behavior. Based on this fact, the proposals take advantage of it, by performing different actions according to the type of block that is being accessed. The remainder of this chapter is organized as follows. Section 1.2 describes the basics of NUCA (non uniform cache access) architectures that are the assumed in the baseline CMP system. Section 1.3 presents the different types of coherence protocols and summarizes their pros and cons. Section 1.4 depicts the current scalability issues in current and future CMPs. Section 1.5 lists how the rest of this thesis is structured.

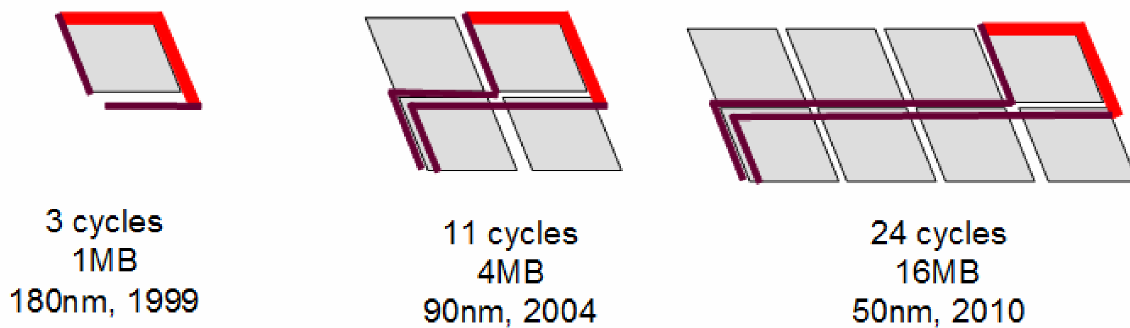


Figure 1.1: Evolution in the access times.

1.2 Non Uniform Cache Access (NUCA)

The improvements in the integration scale have brought an increase in memory cache's capacity and deeper memory hierarchy (L1, L2, L3, etc.), that, in general, occupies a large percentage of the chip's area. The problem lies that the bigger de memory size, the higher the access latency is. Figure 1.1 shows an example. In a conventional memory design the worst case scenario is taken into account, which in this case is defined as the time required to access the memory position furthest away. When sizes were not as big this was not a serious problem, but nowadays it is a considerable penalization in memory access.

A common solution to this problem is the utilization of NUCA (Non Uniform Cache Architecture) type caches, in which the big-sized memory is divided in many of lesser size banks, resulting in different access latency depending on the distance. In other words, an organization of a NUCA consists of a number of columns, a number of banks and the individual capacity of each one of them. Then, for a NUCA with m columns and n banks of c capacity, we achieve a total capacity of $C = m \times n \times c$.

In Figure 1.2 we can observe several different configurations and their respective average access times. In the the upper ones a traditional UCA along an inclusive multilevel UCA are presented. With this additional level in the memory hierarchy we can reduce the high latencies of the UCA. Then, several desgin variants of a NUCA are presented. In the S-NUCA-1 (Static-NUCA-1) the less significant bits are used to decide in which bank the block can be found or should be stored, each one of them with a different acess latency. All banks are connected through data and address buses, which imposes a *wire overhead* of 20.9%, which is a serious performance problem. In the S-NUCA-2 said overhead is reduced up to 5.9% through the use of a 2D interconnection

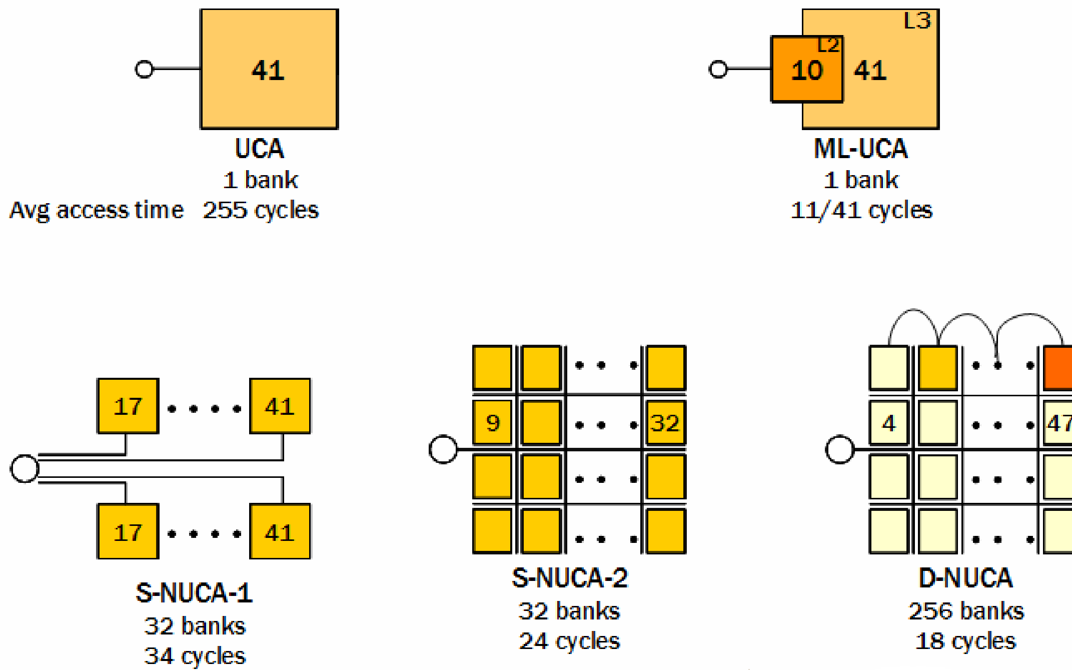


Figure 1.2: Average access time depending of the memory configuration.

network. Lastly, we have the D-NUCA (Dynamic-NUCA) which shares a design similar to the S-NUCA-2. In this configuration a block does not have a fixed bank assigned, but regarding of its utilization rate it will be found on the nearest or furthest banks. That is, the dynamic migration of data between the banks of the cache is allowed. With this we can achieve an improvement in the average access latency.

1.3 Coherence Protocols

In order to maintain the coherence between all processors of a CMP the use of cache coherence protocols is required. In this section a brief summary about the design options of these protocols will be made.

1.3.1 MOESI Protocol

There are many alternatives in the design of coherence protocols depending of the possible states of the blocks stored in their private caches (typically L1). This alternatives are often named by the states they make use of: MOESI, MOSI, MESI, MSI, etc. Each state represents somre read and write permissions for the block stored in the private cache. In this work a MOESI protocol has been used, which has a bigger

number of states (other protocol employ a subgroup of them), and which states are the following:

- M (Modified): A block in modified state has the only valid copy of the data. The core maintains this copy in its cache and has write and read permission over the block. The other private caches can't have a copy. The copy in the shared L2 (if present) is obsolete. When another core requests this block, the cache with the block in modified state must provide it.
- O (Owner): A block in owner state has a valid copy of the data, but in this case, other copies in a shared state may coexist. There can be only a block in owner state. The core maintaining a copy of this block has read permission, but cannot modify it. When a core tries to modify it, coherence actions are needed to invalidate the other copies. In this way, the owner state is similar to the shared one. The difference resides in the fact that the owner is responsible for providing a copy of the block after a cache miss, since the copy in the shared L2 (if present) is obsolete. Furthermore, the evictions of blocks in owner state always need writeback operations.
- E (Exclusive): A block in exclusive state has a valid copy of the data. The other private caches cannot have a copy of this block. The core with this copy has read and write permissions. The shared L2 may also have a valid copy of the block's data.
- S (Shared): A block in shared state has a valid copy of the data. Other cores may also have a copy in shared state and one of them in owner state. If no private cache has the block in owner state, the shared L2 also has a valid copy of the block and is responsible for providing it if requested.
- I (Invalid): A block in invalid state has no valid copy of the data. The valid copies may be found either in the shared L2 cache or in some other private cache.

1.3.2 Update and Invalidation Protocols

When a block that was already stored in cache by another core is writte, it is necessary to take specific actions in order to keep the coherence of the memory system. There exist two main options: update protocols and invalidation protocols.

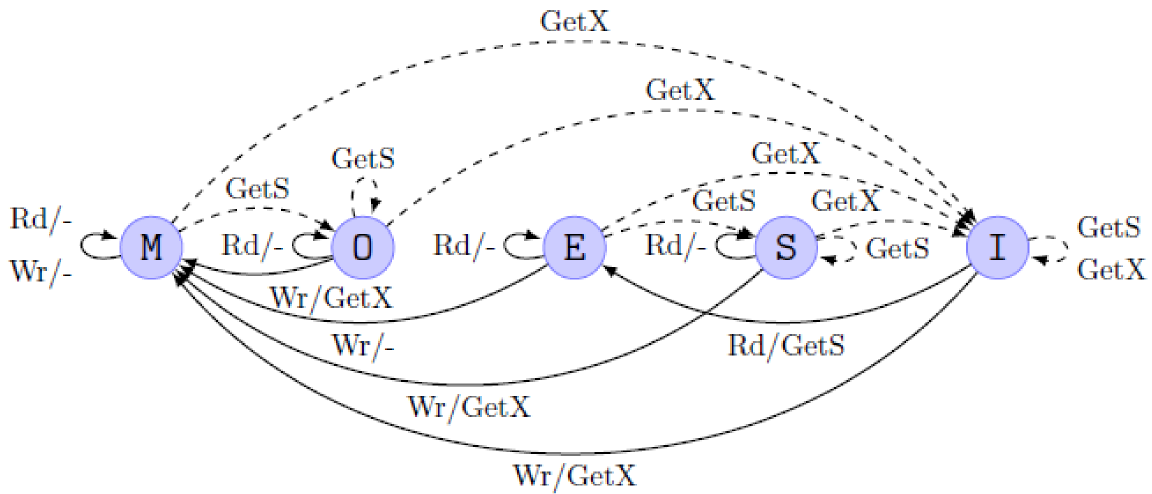


Figure 1.3: State transition diagram for a MOESI protocol.

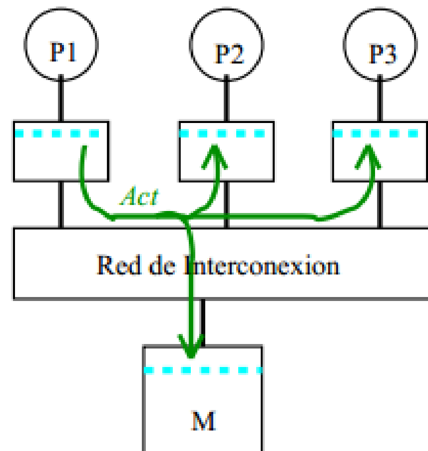


Figure 1.4: Example of how an update protocol works.

In an update protocol, when a core writes on a block, all other copies in the rest of the CMP caches are updated. Subsequent access to that block will have an updated value of it. In order to do this, it is necessary that this update is notified to the rest of the caches, indicating the number of the implicated block and updated value. Once the block is updated, the read operations of other processors don't provoke a cache miss. This is specially good when there are writings made by a processor followed by a sequence of readings of other processors.

On the other hand, we have the invalidation protocols, in which when a core writes on a block, all other copies in the other caches are invalidated. In this case, subsequent accesses will provoke a cache miss. After the resolution of this miss, the cache will obtain an updated copy. With this method, once a block is invalidated, new modifications

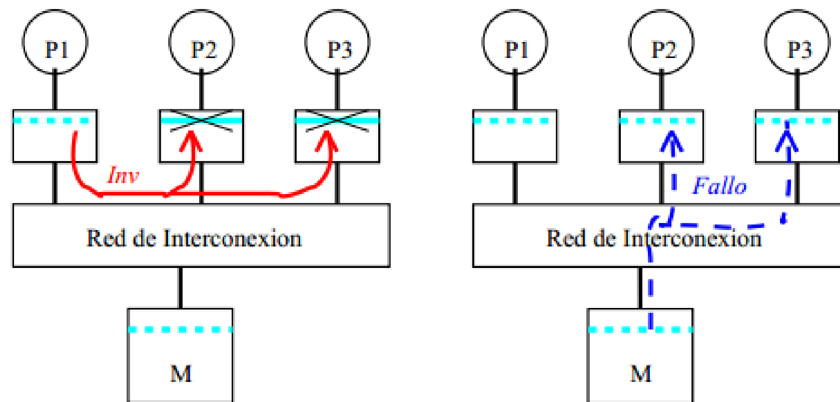


Figure 1.5: Example of how an invalidation protocol works.

by the same core will not trigger new invalidations. Therefore, the best case scenario for an invalidation protocol is that in which there are multiple write operations by a same core.

In this work an invalidation protocol has been chosen, because it is faster and simpler to implement, since we only need to indicate the block number to the remaining caches, than the update one.

1.3.3 Directory-based and Snoopy Protocols

When implementing the coherence mechanisms in the cache system, for example the invalidation or update orders explained in the previous section, we have two options: snoopy and directory-based protocols.

In snoopy protocols this orders must be sent to all caches of all cores, even if they do not have a copy of the block, and they are responsible of finding if that same order affects them in any way or not. For this to be a viable options, an interconnection network that simplifies broadcasting (MPs with shared centralized memory, a common bus) is needed. Furthermore, the system needs to implement a monitorizing mechanism of the bus in order to intercept the invalidation/update orders and add specific lines to the bus in order for it to support the specific protocol being used. The main disadvantage of this kind of protocols is that the higher the number of cores of the CMP, the bigger the traffic induced in the network.

Directory protocols aim to solve this scalability and interconnection problems of snoopy protocols. The systems employing these protocols are the recommended ones

when scalability (in the number of processors) is an important design factor.

One of the main objectives of directory-based protocols is the avoidance of broadcasting. In its place, communications only takes place between those processors which will likely have a copy in its caches. For that, we need a structure that stores if a memory line is in a private cache, which proessors have a copy and if said line is clean or dirty. In a full directory scheme, all information of all memory lines is kept. For example, for a system with n cores, each one with its private cache, a boolean array of size n+1 is needed. If a bit i (i=1,...,n) is set to true, that means that processor i has a valid copy of the line. The (i=0) bit indicates if the line is clean or dirty. An array in which all elements are zero means that the lines is found exclusively in main memory. If the (i=0) bit is active, the line is dirty and only one of the other bits can be active as well. With this we achieve that memory traffic grows linearly with the number of cores, while this grow in snoopy protocols is quadratic. In this point of view we achieve improvements in scalability.

In order to store all information of all memory lines of a full directory, an important amount of area is needed, with brings again scalability restrictions. That is why directory caches, which work similarly to a full directory, are used. Main difference between the two of them is that, when a directory cache is forced to evict a line because of space reasons (something that cannot happen in a full directory), the blocks in the private caches must be invalidated, even if they are still being used, in order to keep the coherence. This way a new kind of miss arises and adds itself to the list of misses that are inherent to private caches (cold, conflict and capacity) and to those inherent to shared memory systems (coherence). They are the so called coverage misses, misses that are the result of a block access that has been evicted because there was no space left in the directory cache.

In this work a study of this structure will be made, proposing a new one that improves in terms of energy and area, without compromising performance. For this purpose, we will focus on the behaviour of private and shared blocks (explained in section 3.2).

1.4 Dealing with Scalability in Future CMPs

CMP systems must be designed to accommodate specific area and power budgets. Both technological constraints represent major design concerns since they prevent future manycore CMPs from scalability with future increasing core counts. Power consumption is mainly distributed between cores and large on-chip cache memories in current designs. Caches occupy a large percentage of the on-chip area to mitigate the huge penalties of accessing the off-chip main memory. Giving more silicon area and power to the cache hierarchy and related structures (e.g. directory caches) leaves less space and power for cores, which could force CMP designs with simpler cores so yielding to lower performance, especially for single-threaded applications [MH08].

Many efforts have been carried out in either the industry and academy to deal with power and area focusing on the cache subsystem, including processor caches, off-chip caches and directory structures. Regarding the processor caches they are the responsible of a significant percentage of the area and power consumption of processors. Regarding the directory caches, they have been proven to provide effectiveness and scalability, both in terms of power and area, for a small to medium number of cores. However, these design issues must be properly faced for future systems since the pressure on achieving good cache performance increases with the core counts. There are two main ways to tackle these issues: i) architectural solutions to achieve a good tradeoff among performance, area and power (as in the PS cache), and ii) combining disparate technologies in a power and/or area aware design. Both ways can be applied independently (as in the PS cache) or together, as in the PS directory cache.

This work presents architectural innovations to track separately private and shared blocks in two independent directory caches. The main aim of this two-cache approach is to tailor each cache structure to the requirements of each type of blocks with different architectural solutions. In addition, distinct technologies can be used for implementing the different caches. For instance, a power aware technology can be used for one cache while a fast technology can be employed for the other one. In the case of the PS cache such innovations aim to reduce the number of ways looked up.

The cache hierarchy has been typically implemented with SRAM technology (6 transistors per cell) which consumes important amounts of power and area. A few years ago, technology advances have allowed to embed DRAM (eDRAM) cells in CMOS technology [MS05]. An eDRAM cell integrates a trench DRAM storage into a logic circuit technology. Table 1.1 highlights the main properties of these technologies regarding the design issues addressed in this work. Compared to SRAM, eDRAM cells have both

Table 1.1: Comparing Technological Features of SRAM versus eDRAM.

Technology	**Density**	**Speed**	**Power**
SRAM	low	fast	high
eDRAM	high	slow	low

less power consumption and higher density but lower speed. Because of the reduced speed, eDRAM cells have not been used in manufactured first-level high-performance processor caches. In short, both technologies present diverse features regarding density, speed, and power.

These CMOS compatible technologies have been used both in the industry and the academia to implement processor caches. For instance, in some modern microprocessors [TDF+02, SKT+05, KSSF10] SRAM technology is employed in L1 processor caches while eDRAM cells are used to allow huge storage capacity in last level caches. Regarding academia, some recent works [VSP+09, WLZ+09] mingle these technologies in different cache levels. In short, both technologies properly combined at different (or even the same) cache structures can be used to address speed, area, and power in the cache subsystem.

In this work, and to the best of our knowledge, this is the first time that both technologies are combined to implement the directory cache. We use SRAM for speed in the frequently accessed Shared cache while eDRAM is employed for power and area savings in the much larger Private cache. Therefore, scalability and performance are provided by design thanks to the joined use of architectural techniques and the choice of the appropriate technology for each cache structure.

1.5 Work Structure

The rest of this work is organized as follows. Chapter 2 discusses the related work. Chapter 3 presents the PS directory proposal. Chapter 4 presents the PS cache proposal. Chapter 5 describes the simulation environment. Chapter 6 analyzes the performance, area and energy consumption of the proposals. Finally, Chapter 7 presents some concluding remarks.

Chapter 2

Related Work

This work focuses on two main memory structures of the cache hierarchy: directory caches and processor caches. This chapter summarizes related research to this thesis, broken down according to the memory structure being studied.

2.1 Directory Caches

Cache coherence is needed in shared memory systems where multiple cores have copies of the same memory blocks. Part of this work focuses on directory-based protocols, which are the commonly adopted solution for a medium to large core count. These protocols use a coherence directory to track which private (e.g. L1) processor caches share each block. The directory structure is accessed to carry out coherence actions such as sending invalidation requests to serialize write operations, or asking a copy of the block to the owner (e.g. the last processor that wrote it).

Traditional directory schemes does not scale with the core count, which is the current trend in the microprocessor industry. Thus, implementing directories that scale to hundreds of cores in terms of power and area is a major design concern. Directory implementations, both in academia and industry, follow two main approaches: duplicate-tag directories and sparse directories.

2.1.1 Duplicate-tag and directories

Duplicate-tag directories maintain a copy of the tags of all tracked blocks in the lower cache level (e.g. the L1 core cache). Therefore, this approach does not raise directory induced invalidations. The sharer vector is obtained by accessing the highly associative directory structure. This approach has been implemented in modern small

CMP systems [BGM00, SBB07] and is the focus of recent research works [RAG10, ZSQM09]. The main drawback of this approach is the required associativity of the directory structure, which must be equal to the product of the number core caches by the associativity of such caches. This means that a directory access requires a 512 associative search for 64 8-way L1 caches. Duplicate-tag directories are area–efficient, however, the highly associative structures yield to a non-scalable quadratic growth of the aggregated energy consumption [FLKBF11a], so this approach becomes prohibitive for a medium to large core count.

The high power consumption incurred by duplicate-tag directories has led some research to focus on providing high associativity with a small number of ways. Cuckoo Directory [FLKBF11a] uses a different hash function to index each directory way, like skew–associative caches. Hits require a single lookup but replacements require from multiple hash functions to provide multiple candidates, so giving the illusion of a cache with higher associativity but at the expense of higher consumption and latency.

2.1.2 Sparse directories

Sparse cache directories [GWM90] are organized as a set-associative cache like structure indexed by the block address. Reducing the directory associativity makes this approach more power–efficient than duplicate–tag directories. Each cache directory entry encodes the set of sharers of the associated tracked block. Conventional approaches use a bit vector, that is, a bit per-core cache, to encode the sharers. In this scheme, the per-core area grows linearly with the core count and the aggregated directory area grows quadratically, since the number of directory structures increases with the number of cores. Previous research works have focused on reducing directory area by focusing on the entry size.

To shorten the entry size some approaches use compression [AGGD01, CKA91, Che93, ON90]. In [AGGD01, AGGD05] a two-level cache directory is proposed. The first-level stores the typical sharer vector while the second-level uses a compressed code. When using compression, area is saved at expenses of using an inexact representation of the sharer vector, thus yielding to performance losses. Hierarchical [GWX+10] representation of the sharer vector has been also used for entry size reduction purposes. However, hierarchical organizations impose additional lookups on the critical path so hurting latency. Sparse directories may reduce area by reducing the number of directory entries but at the expense of performance since directory evictions force invalidations at the core caches of the blocks being tracked.

Unlike typical sparse directories, a recent scheme [SK12] uses different entry formats of the same length. Lines with one or a few sharers use a single directory entry while widely shared lines employ several cache lines (multi-tag format) using hierarchical bit vectors. This scheme requires extra complexity and accesses for managing dynamic changes (expanding/contracting) in the format.

Finally, other proposals [CRG+11] focus on reducing the number of entries implemented in the cache directory instead of focusing on the sharer vector. While this approach does not affect the performance, it requires modifying the OS, the Page Table, the processor TLBs and the coherence protocol.

2.2 Cache Memories

The second proposal of this work is an energy-efficient cache design that takes advantage of a private-shared detection of the blocks referenced by applications. Hence, this section first reviews some related work about energy-efficient cache designs, and then, it discusses some other optimizations based on a private-shared detection.

2.2.1 Energy-efficient cache designs

Caches consumption comes from both leakage (or static) and dynamic consumption. Regarding leakage reductions, Powell *et al.* [PhYF+00] proposed a Gated-Vdd technique that aims to reduce leakage for instruction caches by reconfiguring them and turning off unused lines. Kaxiras *et al.* [KHM01] proposed the Cache Decay, an approach that reduces the leakage power of processor caches by turning off those cache lines that are predicted to be dead, i.e., not referenced by the processor before they are evicted. Alternatively, Flautner *et al.* [FKM+02] exploited the fact that in a particular period of time only a subset of the cache lines are accessed to propose the Drowsy Caches. Different from the previous proposals, the voltage is reduced, but not cut off, for those cache lines that are not being accessed. In this way, the content of the cache line is not erased.

While techniques that aim to save leakage focus on reducing (or cutting off) voltage, dynamic energy saving techniques try to minimize the number of data read and written on every cache access. For example, Albonesi [Alb99] proposed Selective Cache Ways, a cache design able to enable only a subset of the cache ways when the cache activity is not high. The prediction of ways was previously proposed by Calder *et al.* [CG96] to reduce the access time of set-associative caches. Ghosh et al. [GÖF+09] proposed

Way Guard, a mechanism for large set-associative caches that employs bloom filters to reduce dynamic energy by skipping the lookup of cache lines that do not contain the requested data according to the bloom filter.

Other techniques focus on reducing both leakage and dynamic consumption, for example, by reducing the area of the cache tags, like in the TLB Index-Based Tagging [LHK11], or by performing run-time partitioning, like in the Cooperative Caching scheme [SPJ+12] or in the ReCaC scheme [KCG+10].

Our proposal will allow the reduction of both leakage and dynamic consumption without arising the need for a reconfiguration.

2.2.2 Private-shared optimizations

The detection of private and shared data can be employed to optimize performance and power. Kim *et al.* [KKH10] detect the sharing degree of memory pages to reduce the fraction of snoops in a token-based protocol. In this way, they can replace broadcast messages with multicast ones, thus reducing the energy consumption of the interconnection. The R-NUCA approach by Hardavellas *et al.* [HFFA09] detects private and read-only pages and maps them efficiently in a distributed NUCA cache. By mapping private pages to the closest NUCA bank to the core accessing them, the access latency is reduced, but also the amount of traffic generated, which will impact in the energy consumed by the network. Cuesta *et al.* [CRG+11] deactivate coherence for private pages, thus avoiding their tracking by directory caches. This enables to reduce the directory size up to eight times while still maintaining performance. Reductions in directory area also leads to reduce both dynamic and leakage consumptions.

Some previous works rely on the compiler and/or memory allocator to classify memory pages in order to either remove coherence for private pages [MS09] or improve data placement [LAMJ10, LMJ12]. In [LAMJ10], a data ownership analysis of memory regions is performed at compilation time. This information is transferred to the page table by modifying the behavior of the memory allocator by means of hooks. This proposal is further improved in [LMJ12] by considering a new class of data, named as practically-private, which is mapped to the NUCA cache according to a first-touch policy. In [MS09], private data is not stored at the LLC with the aim of avoiding cache thrashing for private blocks. Different from our approach, these works mark statically data as private either by the memory allocator or at compile time, when privacy of some data cannot be guaranteed.

SWEL [PSNB10] is a novel hardware-based coherence protocol that uses a private-

shared block classification at the directory to allocate shared read-write blocks only at the shared LLC, so removing the need of coherence maintenance for them. The main drawback of that proposal is the latency penalization of accessing shared read-write blocks, which must be served by the LLC cache. POPS [HDH11] decouples data and coherence information in the shared LLC to reduce access latency to this information and to improve the aggregate NUCA capacity. It also employs a directory private-shared classification (this time with the help of a predictor table). Spatio-temporal Coherence Tracking [Ali12] also classifies private and shared data at the directory, accounting for temporal private data. It tries to find large private regions to merge them in the directory to save directory space. Finally, Ros and Kaxiras [RK12] proposed VIPS, a complexity-efficient coherence protocol that employs write-back caches for private data for efficiency reasons and write-through caches for shared data for protocol simplicity.

Chapter 3

PS Directory

This chapter analyzes the behavior of private and shared blocks, and presents the PS directory proposal. Also, the baseline CMP that will be considered in this work is introduced.

3.1 Base Architecture

A tiled CMP architecture consists of a number of replicated *tiles* connected by a switched direct network. Different tile organizations are possible so, to focus the research, this work assumes that each tile contains a processing core with primary caches (both instruction and data caches), a slice of the L2 cache, and a connection to the on-chip network. Cache coherence is maintained at the L1 caches. In particular, a directory-based cache coherence protocol is employed with a directory cache storing coherence information. Both the L2 cache and the directory cache are shared among the different processing cores but they are physically distributed among them, that is, it is implemented as a NUCA architecture [KBK02]. Therefore, a fraction of accesses to the L2 NUCA cache is sent to the local slice while the rest is serviced by remote L2 slices. In addition, L1 and L2 caches are non-inclusive, that is, some blocks stored in the L1 caches may not have an entry in the L2 cache (but in the directory). Figure 3.1 shows the organization of a tile (left side) and a 16-tile CMP (right side), which is used as baseline for experimental purposes.

This work pursues to take advantage of the parallel workload behavior to design more efficient directory and data caches.

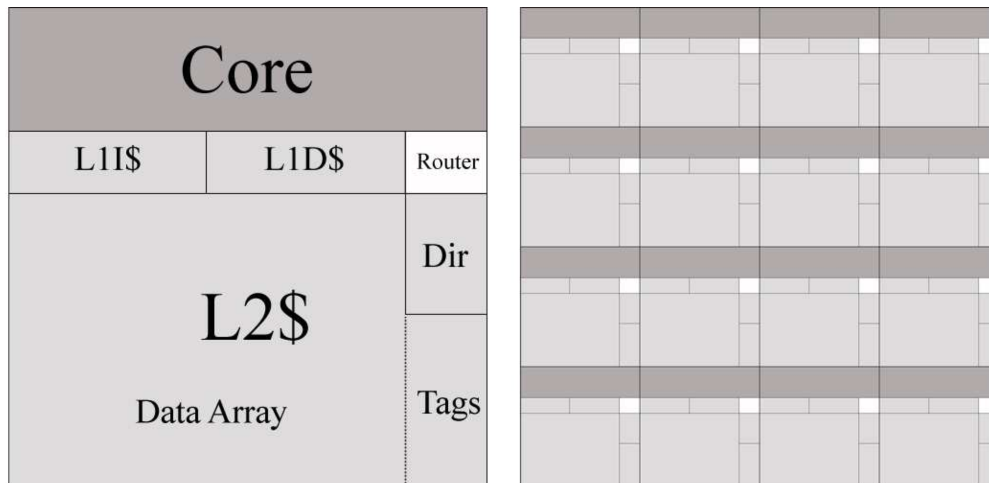


Figure 3.1: Organization of the tile assumed in this work and a 4×4 tiled CMP.

3.2 Analyzing the Behavior of Private and Shared Blocks from the Directory Point of View

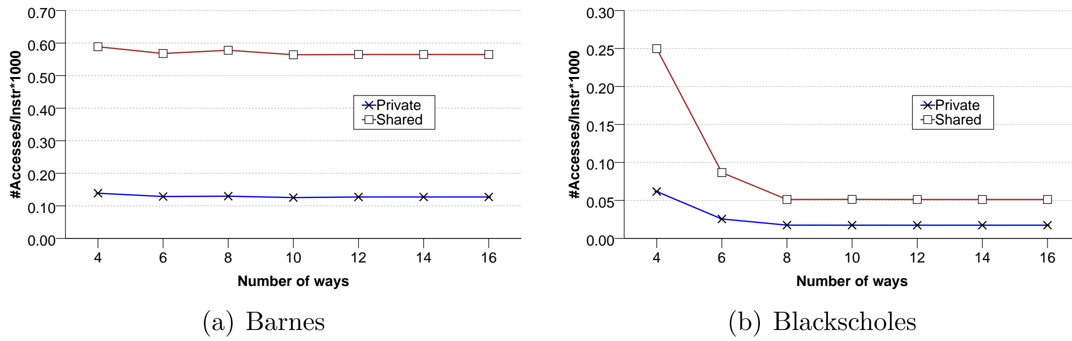
The PS directory relies on the fact that private and shared blocks present different behavior from the directory point of view, which can be outlined in four key observations and one finding. As explained below, these five key points advocate to organize directory caches in two independent structures, one for tracking private blocks and the other for shared blocks.

- **Observation 1:** *Directory entries keeping track of private blocks do not require the shared vector field.*

- **Observation 2:** *Most data blocks in parallel workloads are private.*

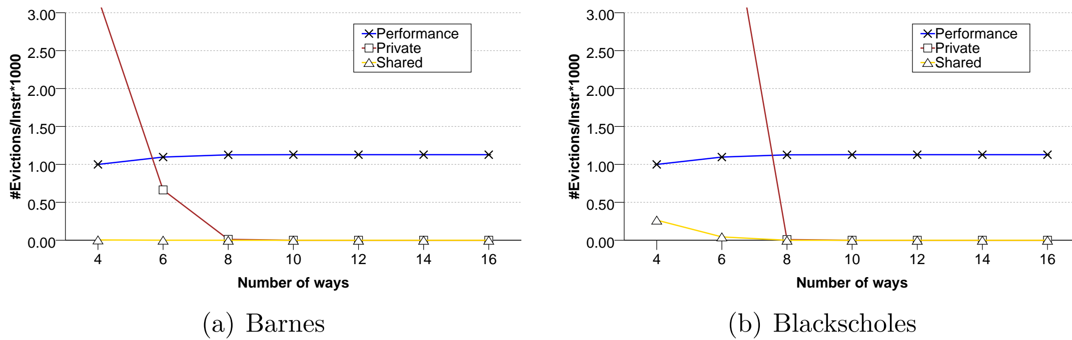
This observation suggests to store entries of shared and private blocks in two different structures. According to these two observations, the Private cache should be designed narrower and taller than the Shared cache, respectively, that is, with shorter entries but with a higher number of them. Due to the smaller entry size in the Private cache important area savings can be achieved, especially for systems with a large number of cores, thus offering scalability. Notice that the larger the Private cache is in comparison with the Shared cache, the more area savings can be obtained, thanks to the missing sharer vector field.

- **Observation 3:** *Most directory look–ups concentrate on shared entries.*



(a) Barnes (b) Blackscholes

Figure 3.2: Number of accesses to private and shared entries per kilo instruction in a conventional directory.



(a) Barnes (b) Blackscholes

Figure 3.3: Number of evictions of private and shared entries per kiloinstruction in a conventional directory and its effect on performance.

- **Observation 4:** *Almost all directory entries for private blocks are accessed only once.*

These observations emphasize that private blocks access the directory either when they are not stored in the processor cache (e.g., first access or invalidations due to directory evictions) or when a write-back is performed (e.g., due to space constraints in the processor cache). The first case will cause a directory miss, while the second case will hit in the private directory cache and will invalidate the corresponding entry. On the other hand, shared entries are accessed more times due to the different cores sharing the same block. Thus, most directory accesses are due to shared blocks. According to this reasoning, the PS directory accesses the Shared cache first so preventing likely useless accesses to the Private cache, which will result in energy savings.

Figure 3.2 depicts the number of directory entries looked-ups (differentiating between shared and private) per kilo instructions committed, varying the number of ways

in the directory cache and keeping constant the number of sets[1]. Two benchmarks, *Barnes* from the SPLASH-2 benchmark suite [WOT+95] and *Blackscholes* from PARSEC [BKSL08], have been used to illustrate these observations.

As can be seen, the number of accesses to entries tracking shared blocks is about 5× larger than that to entries tracking private blocks in Barnes. Entries of private blocks are only looked up again in case a block is replaced either from the directory or from the processor cache, and then asked again by the processor. In both cases, the directory entry is removed, thus when the corresponding private block is looked up in the directory, a miss will occur. Number of accesses in the figure is different from zero because a given block can walk the directory several times. Consequently, private entries are scarcely accessed in spite of being the number of them much larger than that of shared entries. Results for Blackscholes show minor differences for a higher number of ways because the number of directory evictions is noticeably reduced in this benchmark as the directory capacity increases. These results suggest that while shared blocks should have a reduced directory access time for performance, this time is not so critical for private blocks. Keeping this observation in mind, we study the potential benefits of using a power and area aware technology to implement the private cache.

- **Finding 1:** *Shared directory entries have much less associativity requirements than private directory entries.*

To quantify the proper associativity degree, we ran experiments with a conventional (or single–cache) directory varying the number of ways. We identified and quantified the number of evicted directory entries that cause subsequent misses in the processor caches, and classified them into private and shared according to the type of the block that was being tracked. Then, the effect of both types of blocks on performance was measured. Misses in the processor caches that occur due to an eviction of directory entries will be referred to as *coverage misses* as also done is some recent works [CRG+11, RCFP+12].

We found that private and shared entries have different associativity requirements. Figure 3.3 illustrates the results for two different workloads. Results reveal that the number of evicted shared blocks provoking coverage misses slightly varies with the number of ways, while the number of private blocks drops dramatically. The number of evicted private blocks is really high for a low associativity degree, which translates to significant performance degradation.

[1]Experimental conditions are defined in Chapter 6

Assuming a typical LRU replacement policy and tacking into account that entries in the directory tracking private blocks are not accessed again, the time an entry is busy tracking a given block works out like a FIFO policy, that is, in absence of locality, the impact of private blocks on performance mainly depends on the number of ways available to them. If it is too low, it is likely that the block will be forced to be evicted from the processor cache being stored in, even though it is still being used, thus increasing the number of coverage misses. On the other hand, with a higher number of ways, we give them more chances before eviction. It can be observed that around 8 ways is enough to stabilize the number of evictions of private blocks as well as the system performance.

Based on these results, the Private cache will be designed with around 8 ways (see Chapter 5 for further details), whereas the Shared cache will implement a lower number of ways (i.e., 2 ways), since the impact of shared blocks on performance mainly depends on their access locality and are not benefited from such a large complexity.

3.3 The PS Directory Scheme

The main goal of this proposed approach is to take advantage of the different behavior exhibited by shared and private directory entries to design scalable directory caches while, at the same time, improving their performance. Figure 3.4 depicts the proposed two-level organization consisting of the Private cache and the Shared cache.

As previously exposed, the Private cache is designed with narrower entries since they do not require the sharer vector and with a larger number of entries because of the expected high number of private blocks. On the other hand, the Shared cache has a reduced number of entries, thus the sharer vector is only implemented in a small fraction of directory entries.

When an access to a memory block misses in the processor cache, it is looked up on the directory for coherence maintenance. Then, if the access results on a directory miss, the block is provided by the NUCA slice (or by the main memory) to the processor cache, and an entry is allocated in the directory cache to track that block. In the PS directory this entry is allocated in the Private cache since the block is held at this point of time by a single cache. Then, the core identifier is stored in the entry owner field.

On subsequent accesses to that memory block by the same processor, it will find the block in its L1 cache, so that no additional access to the directory cache will be done. On the other hand, when that block is evicted from the processor cache two main actions are carried out: i) the data block is written back in the NUCA cache,

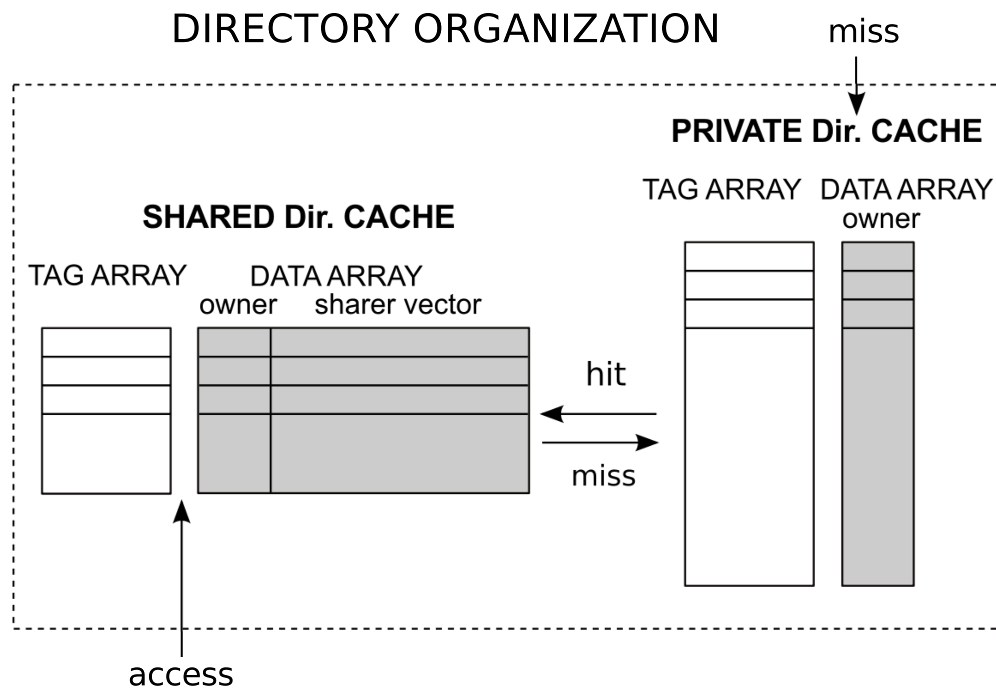


Figure 3.4: Private-Shared directory organization.

and ii) the directory cache is notified in order to invalidate the entry of that block (stored in the Private cache). Thus, a subsequent access to that block will result in a directory cache miss. This means that the Private cache access time does not affect the performance of private blocks since these blocks are provided directly to cores by the NUCA cache or main memory.

If a block tracked by the Private cache is accessed by a core other than the owner, the block becomes shared and its entry is moved to the Shared cache updating the sharer vector accordingly. From then until eviction, coherence of this block is tracked in the Shared cache. That is, the proposal allows only unidirectional movements from the Private to the Shared cache. Bidirectional transfers of entries among both caches have been also explored but the extra hardware cost does not justify the scarce benefits.

Regarding timing, directory caches are typically accessed in parallel with the NUCA cache. On a directory hit, the data block can be provided either by the NUCA cache or by a remote processor cache (i.e. the owner). In case that the data block must be provided by a remote processor cache, the NUCA access is canceled. Analogously, the PS directory could access both directory cache structures, however since most directory accesses concentrate on shared blocks, the PS scheme only accesses the Shared cache in parallel with the NUCA slice. This way provides major energy savings with minimal performance penalty. Figure 3.5 depicts this design choice. Depending on the protocol,

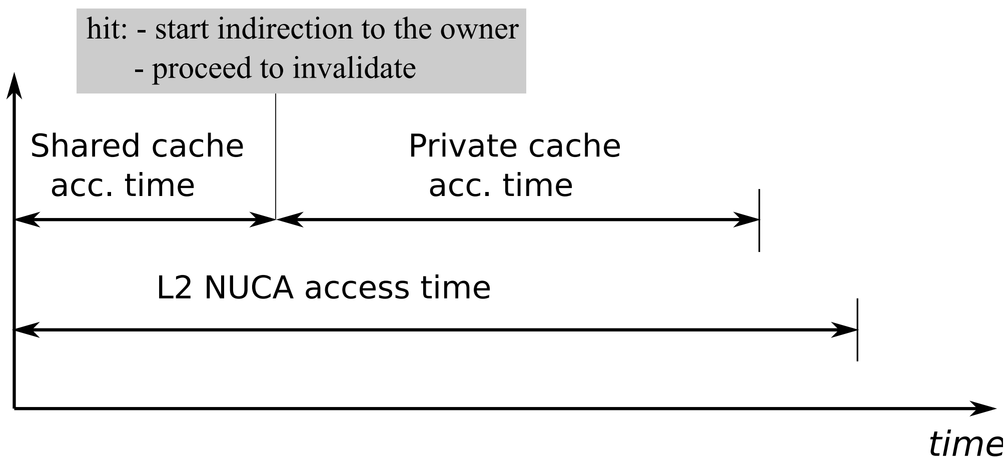


Figure 3.5: Parallel access of the Shared cache and the NUCA cache. Private cache is only accessed on a miss in the Shared cache.

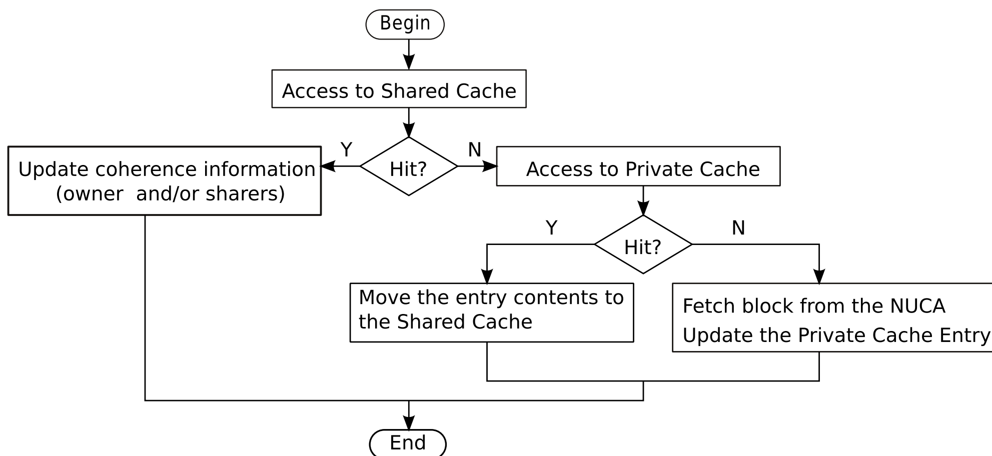


Figure 3.6: Directory controller flow-diagram.

specific coherence actions can start as soon as a hit rises in the Shared cache; for instance, read requests can be forwarded to the owner of the block, or invalidation requests can be issued to the caches sharing the block in case of write requests. On a miss in the Shared cache, the Private cache is accessed. This access could be also performed in parallel with the Shared cache but at expenses of power while bringing minimal benefits on performance.

Figure 3.6 summarizes the actions carried out by the directory controller on a coherence access, which works as follows:

- When a coherence request reaches the directory, the directory controller looks up first the Shared cache since it is more likely that the access results in a hit in this cache due to the higher fraction of accesses to shared entries. On a hit, the con-

troller updates (if needed) the sharer vector, performs the associated coherence actions, and cancels the NUCA access (depending on the block state). On a miss in the Shared cache, the controller looks up the Private cache. This sequential timing has, on average, negligible impact on performance since most directory accesses are to shared blocks, and most accesses to private blocks provide the block from the NUCA cache.

- A hit in the Private cache means that the block is shared because another core already has a copy of it in its cache. The processor that accessed it the first time will not access the directory again because its cache already holds the block, unless a data cache or directory eviction occurs and them the entry will miss in the directory again. Hence, the directory entry is moved to the Shared cache. This way ensures that entries for private blocks are retained in the Private cache while shared entries are filtered and moved to the Shared cache.

- On a directory miss, an entry is allocated in the Private cache to keep track of the missing block. As there is no coherence information stored for that block in the two directory caches, then the block is not being currently cached by any processor. Thus, the block is assumed to be private to the core accessing it and the owner information (requesting processor) is updated with the core identifier.

- In the proposed implementation, when an entry is replaced from any of both directory caches it moves away from the directory after performing the corresponding invalidations, and no movement to the other cache is allowed.

The proposal reduces area by design with respect to conventional caches implemented with the same number of entries since directory entries in the Private cache are much narrower. In addition, power is also reduced by accessing smaller cache structures sequentially. Nevertheless, the use of two independent organizations with different design goals, speed for the Shared and capacity for the Private, suggests that using specific technologies addressing these design issues could provide the proposal further energy and area savings.

Low-leakage technologies or transistors with low leakage currents could be used in the Private cache, whose number of entries is much higher and its access time is not critical for performance. This work explores the use of eDRAM technology in the Private cache which provides, as experimental results show, important area and leakage savings.

Chapter 4

PS Cache

This chapter studies how memory reference instructions access the different structures of the cache hierarchy. The study discern among writes (stores) and reads (loads). Then, the PS cache scheme is proposed, which represents one of the major contributions of this work.

4.1 Accessing the Cache Hierarchy

Memory reference instructions represent a significant percentage of the executed instructions. Moreover, when running multithreaded workloads, in addition to access the local cache, remote caches (e.g. remote caches) can be accessed for coherence purposes.

Unlike multiprogrammed workloads, the accessed blocks in these workloads can be classified in two different categories: private blocks and shared blocks. The former are accessed only by a core while the latter are accessed by several cores. Figure 4.1 illustrates a possible distribution of shared and private blocks in a cache with 8-way sets. Some blocks can be in an invalid state, so they are neither classified as private nor shared.

Cache memories, especially first-level caches, are frequently accessed. Low level caches are much less frequently accessed since L1 caches filter many of the memory instructions issued by the processor. For instance, assuming that a 4-way superscalar out-of-order core executing a given benchmark reaches an IPC equal to 2, and memory reference instructions represent a fifth of its executed instructions, then the L1 data cache of the core is accessed 40% of the processor cycles. However, if the L1 data cache filters by 80% of the processor requests, the L2 data cache will be only accessed by the remaining 20% requests. This means that L1 caches consume most of the dynamic

power of the cache hierarchy which is our focus in the PS cache design.

We launched experiments to quantify these values in the studied workloads. Figure 4.2 shows the percentage of memory reference instructions in each individual benchmark executed on a 16-core CMP system[1]. This value is roughly the same, around 20%, across the different benchmarks.

Only about a fifth of accesses issued to the L1 cache go on to the L2 cache. This means that most of the dynamic energy consumed by caches comes from first-level caches. Consequently, it makes sense to focus on saving dynamic energy at that level. Even more, if we consider that both the tag and data arrays are accessed in parallel in L1 caches and that some recently processors such as the Intel Core i7 [HP11] has been deployed with high associativity in the L1 cache (i.e. 8 ways). To provide further insights in these values and which fraction is filtered by the L1, we classify requests to L1 in four main categories according to the result of the access as: load hit, store hit to a private block, store hit to a shared block and L1 misses. Those accesses falling into the two former categories are filtered by L1 and do not access to the second-cache level, while the remaining memory requests access both to the L1 (since this cache is accessed by every memory instruction) and to the L2. Figure 4.3 shows these results. As observed, a very small fraction of L1 requests (below 5%) access then to the L2. Notice that the percentage of writes to shared blocks can hardly be appreciated in the plot since this value is negligible.

When running parallel workloads, the number of accesses to the cache increases with respect to monolithic processors because of coherence requests issued by other

[1]Experimental environment, system parameters, protocols and cache hierarchy are described in Chapter 5.

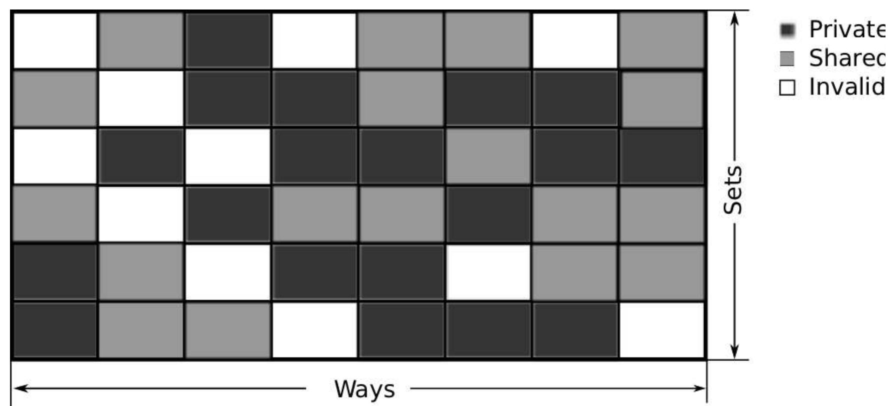


Figure 4.1: Example of private-shared blocks distribution in a 8-way set cache.

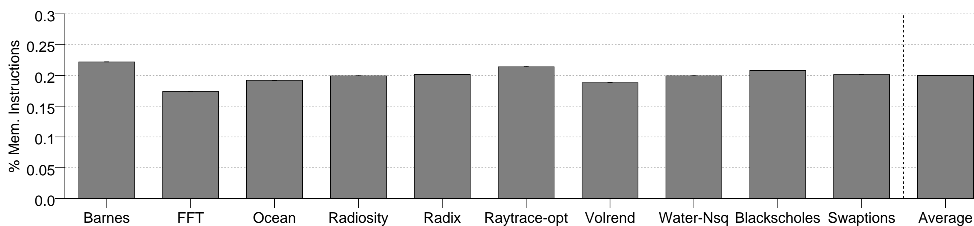


Figure 4.2: Fraction of memory instructions.

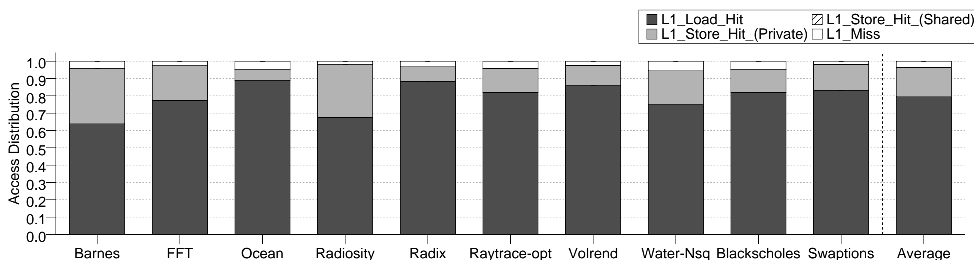


Figure 4.3: Distribution of accesses to the L1 cache and to the L2 cache.

cores. In other words, the cache is not only accessed from the processor side but also from the interconnection network (NoC) side, therefore increasing the dynamic power consumption. In this context, the number of accesses coming from the NoC strongly depends on the type, snoop-based or directory-based, of the underlying coherence protocol.

Snoop-based protocols are based on broadcasting coherence requests to all the cores, which requires high bandwidth and energy consumption at the network but also at the cache since all caches in the system are accessed on a coherence request. Thus they are only appropriate at small system scales [APJ09, CSL+06, KKH10]. Directory-based protocols keep track of the various copies of cached blocks in a directory structure between the private and the shared cache levels [CRG+11, FLKBF11b, ZSQM09]. This allows processors to easily identify replicas of a block. Coherence requests are only sent to cores storing a replica, so only a subset of caches are looked up. This makes them more suitable for large-scale CMPs.

Figure 4.4 shows the percentage of cache accesses coming from the bus side in both types of protocols in a 16-core system[1]. As observed, this value is noticeable in snoopy protocols and it represents around one fifth of the total accesses, moreover, in some workloads this value is as high as 45%. In contrast, this value presents a scarce interest in directory-based protocols.

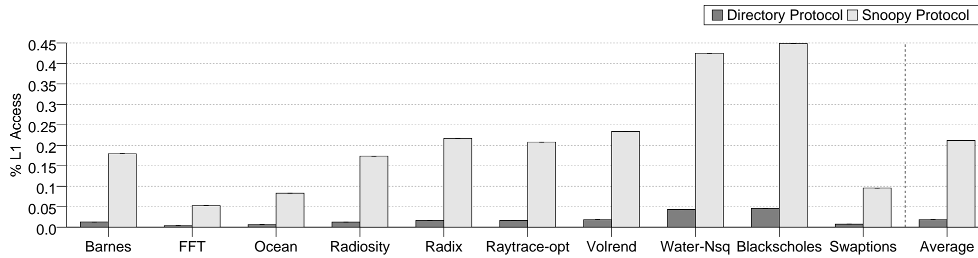


Figure 4.4: L1 Coherence lookups

The previous discussion illustrates the importance of reducing dynamic power consumption in caches of CMP systems, mainly in L1 caches and in snoop based protocols as well, where more coherence requests are issued by the cache controllers. To deal with this problem, this work proposes the PS cache, an architectural approach with the aim of taking advantage of the behavior exhibited in parallel workloads.

In particular, previous research has observed that the percentages of private (accessed only by a core) and shared cache blocks in parallel workloads running in CMP systems widely differ among them. The aim of this thesis is to save energy by reducing the number of lookups on each cache access. More precisely, the proposal saves significant energy consumption by looking up only those blocks having the same type (shared or private) as the requested one either by the processor or by the coherence access.

4.2 The PS Cache Scheme

The main goal of this proposal is to take advantage of the classification between private and shared blocks to design a more power-efficient cache architecture that reduces the number of ways looked up on each cache access. Instead of accessing all the ways in a set, only a subset of them are searched. Our baseline architecture is the same as the one presented in section 3.1.

In order for the proposal to work we need i) to keep blocks tagged as private or shared in the cache, and ii) a private-shared classification mechanism to indicate the type of the block to be searched. Blocks are tagged in the cache using a bit (the PS bit) attached to each cache line. This bit indicates the type of each stored block. In addition, although our proposal can work with any of the private-shared classification mechanisms, in this work we assume the TLB-based private-shared mechanism used in [CRG+11] and previously proposed in [HFFA09] to find out the block type before

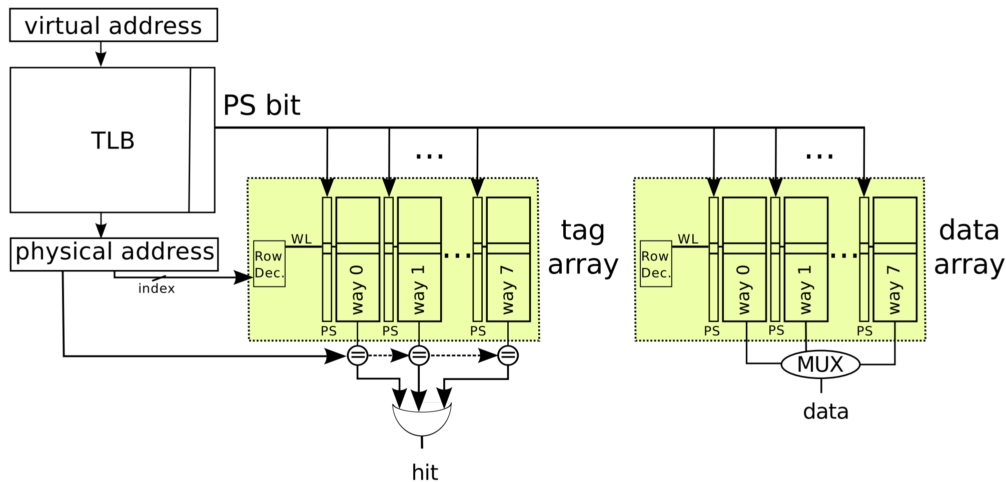


Figure 4.5: The PS cache architecture.

accessing the data and tag arrays, and in this way only access those ways whose type (PS bit value) matches it. In other words, only those ways of the set whose type mathes the TLB bit of the searched block page are accessed.

4.2.1 The PS Page Classification Mechanism

On a memory reference, the core obtains the block type of the reference from the TLB when it is accessed with the purpose of getting the address translation. The classification mechanism is OS-based and stores sharing information both in the page table and in the core TLB. The sharing information comprises the PS bit and the *keeper* (this is only stored in the page table), which is the core that first accessed the page. On a TLB miss, the page table is accessed and the sharing information is updated comparing the keeper with the core requesting the access and the PS bit is copied in the TLB of the requestor core. This TLB bit is the one used by the core on a memory reference to know the type of the requested block.

The classification mechanism assumes that by default a page is on a private state so all the blocks of that page stored in the first-level caches are also private. If the OS detects that a second core is trying to access any block within a private page (that is, the keeper field in the page table is different to the core identifier accessing the page), the page must become shared. The page table entry is updated but as in [CRG+11, HFFA09], a private-shared update mechanism is triggered to maintain coherence between the page table and the keeper's TLB.

The private-shared update mechanism has to ensure that all the PS bits in the

cached blocks of a given page as well as in the TLBs keep the same type as their entry in the page table. For doing this, the core requestor, i.e. the core that wants to access a block in the page, issues a *recovery request* to the page keeper (obtained from the page table entry). On the arrival of such a request, the keeper updates its TLB PS bit and also the PS bits of the cached blocks of the given page.

This way we keep the PS bit, of the different ways in the cache, coherent with the state of the page in the TLBs and the page table. As commented above, the PS bit allows us to discern the group of ways in which the requested block can be found and, consequently, only search in those ways for it.

4.2.2 The PS Cache Architecture

Figure 4.5 depicts an overview of the proposed cache architecture. On a memory reference instruction, the cache controller first looks in the TLB to get the physical address. As can be seen, the TLB includes a bit per entry, the PS (Private-Shared) bit, that indicates how the page is classified. The value of this bit is read from the TLB entry jointly with the physical address. With this information, the cache controller proceeds searching the block in the L1 cache. The key difference is that only those ways which share the same type as the one indicated by the TLB are searched.

As observed, each cache line has attached a PS (Private-Shared) bit which indicates the type of each block (according to the page table and the TLBs). The PS bit provided by the TLB is compared with the PS bits of all the ways in the set. A simple logic is included to select the wordline (WL) of those ways whose PS bit matches the value of that obtained from the TLB. This means, that in the tag array, only a subset of the tags are read and then compared with the tag of the physical address and, in the data array, only a subset of data blocks are read. This allows the proposal to reduce significant dynamic power consumption across the memory accesses since in general, as experimental results will show, only a small fraction of ways is required to be searched in many memory operations.

Apart from reducing the number of searched ways when accessing the cache from the core side, our proposal also allows to reduce the dynamic consumption when accessing the cache from the NoC side (i.e. coherence requests). When accessing from the NoC side, only shared ways are looked up, since the classification mechanism ensures that before arriving the coherence request, the block is shared or it has been reclassified as shared by the private-shared update mechanism.

In addition, in order to reduce the static power consumption our mechanism takes

also advantage of the invalid bit. The power of all ways in an invalid state is turned off and are also excluded from the process of searching the block. This allows not only reducing the number of possible ways for the block (the lower the number, the less dynamic consumption), but also reducing the static energy consumption since power supply to these ways is cut off while they are in the invalid state.

Regarding hardware complexity, the proposal requires minimal complexity. On one hand, no extra information must be added to the TLB except a single bit (the PS bit) to each entry. On the other hand, the proposal can be easily adapted to current caches. In fact, using a single wordline for all the ways in the set presents several problems due to, among others, many transistors are connected to the row's wordlines and the column's bitlines increasing the total capacitance and resulting in an increase in delay and power dissipation. Thus, to deal with this problem, current SRAM cache designs employ the divided wordline approach (DWL), which divides the wordline into a fixed number of blocks, for instance, one WL per cache way [JNW07]. Notice, that our proposal takes benefit of this wordline scheme already working in current caches. In addition, the proposal can be easily integrated in current L1 caches which are pipelined in three or four stages, so we could get the PS bit in a first stage and then adapt the following cache stages.

The previous discussion focused on typical physical tag and physical index caches. However, the proposal could be also applied to other types of caches; for instance, virtually indexed but physically tagged. These caches access the tag array in parallel with the TLB, and then the physical address is used to compare tags. Notice that our scheme would not carry out any tag comparison since the PS bit of the TLB is obtained at the same time as the physical tag. Therefore, energy consumption due to tag comparison, which dominates the dynamic energy, would be avoided.

Chapter 5

Experimental Framework

This chapter describes the experimental framework used to carry out this work. This framework includes the simulation packages, the followed methodology and the used workloads.

5.1 Simulation tools

In this section, we describe the simulation tools employed through this thesis. We use the Simics-GEMS simulator in order to perform the performance evaluation. The area requirements of our proposals and the latencies of cache accesses used in our simulations have been calculated using the CACTI tool.

5.1.1 Simics-GEMS

Simics [MCE02] is a functional full-system simulator capable of simulating several types of hardware including multiprocessor systems. Full-system simulation enables us to evaluate our ideas running realistic workloads on top of actual operating systems. In this way, we also simulate the behavior of the operating system. Differently from trace-driven simulators, Simics allows dynamic change of instructions to be executed depending on different input data.

GEMS (General Execution-driven Multiprocessor Simulator) [MSB05] is a simulation environment which extends Virtutech Simics. GEMS is comprised of a set of modules implemented in C++ that plug into Simics and add timing abilities to the simulator. GEMS provides several modules for modeling different aspects of the architecture. For example, Ruby models memory hierarchies, Opal models the timing of an

out-of-order SPARC processor, and Tourmaline is a functional transactional memory simulator. Since we assume simple in-order processing cores we only use Ruby for the evaluations carried out in this thesis.

Ruby provides an event-driven framework to simulate a memory hierarchy that is able to measure the effects of changes to the coherence protocols. Particularly, Ruby includes a domain-specific language to specify cache coherence protocols called SLICC (Specification Language for Implementing Cache Coherence). SLICC allows us to easily develop different cache coherence protocols and it has been used to implement the protocols evaluated in this thesis.

The memory model provided by Ruby is made of a number of components that model the L1 caches, L2 caches, memory controllers and directory controllers. These components model the timing by calculating the delay since a request is received until a response is generated and injected into the network. All the components are connected using a simple network model that calculates the delay required to deliver a message from one component to another.

5.1.2 CACTI

CACTI (Cache Access and Cycle Time Information) [MBJ09] provides an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. By integrating all these models together, users can have confidence that trade-offs between time, power, and area are all based on the same assumptions and, hence, are mutually consistent.

CACTI is continually being upgraded due to the incessant improvements in semiconductor technologies. Particularly, we employ the version 5.3 for the results presented in this thesis. We are mainly interested in getting the access latencies and area requirements of both cache and directory structures that are necessary for implementing our proposals, assuming a 32nm process technology.

5.1.3 Simulated System

The values of the main system parameters used for the base system in the evaluation are shown in Table 5.1. We used the CACTI 6.5 tool [MBJ09] to estimate access time, area requirements and power consumption of the different cache structures for a 32nm technology node.

Table 5.1: PS Cache System parameters

Memory Parameters	
Cache hierarchy	Non-inclusive
Cache block size	64 bytes
Split L1 I & D caches	64KB, variable number of ways
L1 cache access time	2 cycles
Shared single L2 cache	512KB/tile, variable number of ways
L2 cache access time	6 cycles (2 if only tag accessed)
Memory access time	160 cycles
Network Parameters	
Topology	2-dimensional mesh (4x4)
Routing technique	Deterministic X-Y
Flit size	16 bytes
Data and control message size	5 flits and 1 flit
Routing, switch, and link time	2, 2, and 2 cycles

Different configurations for the PS directory have been evaluated, and results compared with the ones obtained by a conventional directory (*single cache* configuration) with a 1× *coverage ratio* if not stated a different ratio. This ratio indicates the number of directory entries per processor cache entry. So when the ratio is 1× each directory cache slice has the same number of entries as an L1 cache (i.e. 1×). Two PS directory configurations have been evaluated varying its shared-to-private ratio (1:3 and 1:7), that is, the number of entries in the Private cache is three and seven times greater than that of the Shared cache. These two directory configurations have been chosen for comparison purposes, because they have the same number of entries (computed as the sum of entries in both directory caches). Additionally, we perform a sensitivity study with lower coverage ratios for our PS directory in order to show the significant reduction in directory area and power that it can achieve without degrading application performance (Section 6.1.3).

Table 5.2 shows the access time and characteristics for each directory structure. Values for the Private cache were calculated both for SRAM and eDRAM technologies and for different coverage ratios. CACTI provides latencies in *ns* thus we rounded these values to obtain an integer number of processor cycles. The L2 cache access time was assumed to be 6 cycles, and the remaining access times were scaled accordingly. Notice that eDRAM latency is much longer than SRAM latency.

A key finding discussed in Section 3.2 is that private blocks are more benefited from a higher number of ways than shared blocks. This finding suggests that the Private cache should implement higher associativity. Based on this finding, the baseline design

Table 5.2: Directory latencies

Directory cache	# Ways	1× # Sets	Latency 1×	0.5×	0.25×	0.125×
Single cache	4	256	2	2	2	-
Shared dir 1:3	2	128	2	2	2	2
Private dir 1:3 SRAM / eDRAM	6	128	2 / 4	2 / 4	2 / 3	2 / 3
Shared dir 1:7	2	64	2	2	2	2
Private dir 1:7 SRAM / eDRAM	7	128	2 / 4	2 / 4	2 / 3	2 / 3

propose a 6-way Private cache (two ways over the conventional cache). However, for a fair comparison, the associativity of the Shared cache is reduced down to only 2-ways (i.e. two ways less than the conventional one). Notice that, considering both cache structures, the average number of ways per set matches that of the conventional cache, but skewed to the Private cache. In addition, since most of the accesses to the directory are solved in the Shared cache, which is accessed first, power consumption is reduced compared to the 4-way single cache, as studied in Section 6.1.2.

The PS cache evaluation analyzes two different cache coherence protocols: a directory-based protocol and a snoop-based protocol. Both protocols store the blocks in the private caches considering MOESI states, and implement a non-inclusive policy between the L1 and the LLC. Also, invalidation acknowledgements are directly sent to the requester. The main difference between the two protocols is the trade-off between area and power. The directory protocol implements an on-chip directory cache, which increases its area overhead, while the snoopy protocol performs a broadcast on every write, and on every load in case the data is not found in the LLC. Snoopy protocols induce a higher number of coherence requests to the L1 caches, therefore it benefits more from a reduction in the number of ways searched during those requests.

5.2 Metrics and Methodology

The proposals presented in this work have been evaluated in terms of performance, on-chip area required and energy consumed. For evaluating the performance, we measure the total number of cycles employed for each application during its parallel phase, i.e., the execution time of the parallel phase. Although the IPC (instructions per cycle) constitutes a common metric for evaluating performance improvements, it is not appropriate for multithreaded applications running on multiprocessor systems [AW06].

This is due to the spinning performed during the synchronization phase of the different threads. For example, a thread can be repeatedly checking the value of a lock until it becomes available, which increases the number of completed instructions (and maybe the IPC) but actually the program is not making progress.

In order to show why execution time improvements are achieved, we also show L1 cache misses, since a L1 cache miss corresponds to an access to the directory structure, which is the memory hierarchy element which this proposal focus on. This misses are divided in three types of misses (3C, coherence and coverage) to betterunderstand how the PS directory affects each of them. Of special interest are the coverage misses, since one of our main reasons in the directory design is to reduce them. Finally, this L1 misses have also been divided in directory hits and misses, so we can study why we improve o worsen the execution time. A directory miss, if it is already at full capacity, will mean that a directory entry will have to be evicted and, as a consquence, it is possible that a block has to be evicted from one or many L1 caches, which if referenced again will have a negative impact on performance.

Energy consumption is a design issue in actual CMPs, since it influences the encapsulation costs (package) and refrigeration. For this reason, processor desgins must adjust to some specific power restrictions (target power budget). This means that when a processor is designed, its performance cannot be evaluated by itself, but energetic cost must also be taken into account.

In this work area and energy consumption of the proposals will also be evaluated.

All cache coherence protocols evaluated in this thesis have been implemented using the SLICC language included in GEMS. Other protocols, like Token, are already provided by the simulator. All the implemented protocols have been exhaustively checked using a tester program provided by GEMS. The tester program stresses corner cases of cache coherence protocols to raise any incoherence by issuing requests that simulate very contended accesses to a few memory blocks.

All the experimental results reported in this thesis correspond to the parallel phase of each program. We have created benchmark checkpoints in which each application has been previously executed to ensure that memory is warmed up and, hence, avoiding the effects of page faults. Then, we run each application again up to the parallel phase,

where each thread is bound to a particular core. The application is then run with full detail during the initialization of each thread before starting the actual measurements. In this way, we warm up caches to avoid cold misses

5.3 Benchmarks

The proposed schemes have been evaluated with a wide range of scientific applications. *Barnes* (16K particles), *FFT* (64K complex doubles), *Ocean* (514×514 ocean), *Radiosity* (room, -ae 5000.0 -en 0.050 -bf 0.10), *Radix* (512K keys, 1024 radix), *Raytrace* (teapot –optimized by removing locks for unused ray ids–), *Volrend* (head), and *Water-Nsq* (512 molecules) are from the SPLASH-2 benchmark suite [WOT+95]. *Blackscholes* (simmedium) and *Swaptions* (simmedium) belong to PARSEC suite [BKSL08]. The experimental results reported in this work correspond to the parallel phase of the evaluated benchmarks.

5.3.1 Barnes

The *Barnes* application simulates the interaction of a system of bodies (galaxies or particles, for example) in three dimensions over a number of time steps, using the Barnes-Hut hierarchical N-body method. Each body is modeled as a point mass and exerts forces on all other bodies in the system. To speed up the interbody force calculations, groups of bodies that are sufficiently far away are abstracted as point masses. In order to facilitate this clustering, physical space is divided recursively, forming an octree. The tree representation of space has to be traversed once for each body and rebuilt after each time step to account for the movement of bodies.

The main data structure in Barnes is the tree itself, which is implemented as an array of bodies and an array of space cells that are linked together. Bodies are assigned to processors at the beginning of each time step in a partitioning phase. Each processor calculates the forces exerted on its own subset of bodies. The bodies are then moved under the influence of those forces. Finally, the tree is regenerated for the next time step. There are several barriers for separating different phases of the computation and successive time steps. Some phases require exclusive access to tree cells and a set of distributed locks is used for this purpose. The communication patterns are dependent on the particle distribution and are quite irregular. No attempt is made at intelligent distribution of body data in main memory, since this is difficult at page granularity

and not very important to performance.

5.3.2 FFT

The *FFT* kernel is a complex one-dimensional version of the radix-$\sqrt{n}$ six-step FFT algorithm, which is optimized to minimize interprocessor communication. The data set consists of the n complex data points to be transformed, and another n complex data points referred to as the *roots of unity*. Both sets of data are organized $\sqrt{n}$ x$\sqrt{n}$ matrices partitioned so that every processor is assigned a contiguous set of rows which are allocated in its local memory. Synchronization in this application is accomplished by using barriers.

5.3.3 Ocean

The *Ocean* application studies large-scale ocean movements based on eddy and boundary currents. The algorithm simulates a cuboidal basin using discretized circulation model that takes into account wind stress from atmospheric effects and the friction with ocean floor and walls. The algorithm performs the simulation for many time steps until the eddies and mean ocean flow attain a mutual balance. The work performed every time step essentially involves setting up and solving a set of spatial partial differential equations. For this purpose, the algorithm discretizes the continuous functions by second-order finite-differencing, sets up the resulting difference equations on two-dimensional fixed-size grids representing horizontal cross-sections of the ocean basin, and solves these equations using a red-back Gauss-Seidel multigrid equation solver. Each task performs the computational steps on the section of the grids that it owns, regularly communicating with other processes. Synchronization is performed by using both locks and barriers.

5.3.4 Radiosity

Radiosity is an application of the finite element method to solving the rendering equation for scenes with surfaces that reflect light diffusely. Unlike rendering methods that use Monte Carlo algorithms (such as path tracing), which handle all types of light paths, typical radiosity methods only account for paths which leave a light source and are reflected diffusely some number of times (possibly zero) before hitting the eye; such paths are represented by the code "LD*E". Radiosity is a global illumination algorithm in the sense that the illumination arriving at the eye comes not just from the

light sources, but all the scene surfaces interacting with each other as well. Radiosity calculations are viewpoint independent which increases the computations involved, but makes them useful for all viewpoints.

5.3.5 Radix

The *Radix* program sorts a series of integers, called *keys*, using the popular radix sorting method. The algorithm is iterative, performing one iteration for each radix r digit of the keys. In each iteration, a processor passes over its assigned keys and generates a local histogram. The local histograms are then accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. This permutation step requires all-to-all communication. The permutation is inherently a senderdetermined one, so keys are communicated through writes rather than reads. Synchronization in this application is accomplished by using barriers.

5.3.6 Raytrace

This application renders a three-dimensional scene using ray tracing. A hierarchical uniform grid is used to represent the scene, and early ray termination is implemented. A ray is traced through each pixel in the image plane and it produces other rays as it strikes the objects of the scene, resulting in a tree of rays per pixel. The image is partitioned among processors in contiguous blocks of pixel groups, and distributed task queues are used with task stealing. The data accesses are highly unpredictable in this application. Synchronization in Raytrace is done by using locks. This benchmark is characterised for having very short critical sections and very high contention. Barriers are not used for the *Raytrace* application.

5.3.7 Volrend

The *Volrend* application renders a three-dimensional volume using a ray casting technique. The volume is represented as a cube of voxels (volume elements), and an octree data structure is used to traverse the volume quickly. The program renders several frames from changing viewpoints, and early ray termination is implemented. A ray is shot through each pixel in every frame, but rays do not reflect. Instead, rays are sampled along their linear paths using interpolation to compute a color for the corresponding pixel. The partitioning and task queues are similar to those in Raytrace.

Data accesses are input-dependent and irregular, and no attempt is made at intelligent data distribution. Synchronization in this application is mainly accomplished by using locks, but some barriers are also included.

5.3.8 Water-Nsq

The *Water-Nsq* application performs an N-body molecular dynamics simulation of the forces and potentials in a system of water molecules. It is used to predict some of the physical properties of water in the liquid state.

Molecules are statically split among the processors and the main data structure in Water-Nsq is a large array of records that is used to store the state of each molecule. At each time step, the processors calculate the interaction of the atoms within each molecule and the interaction of the molecules with one another. For each molecule, the owning processor calculates the interactions with only half of the molecules ahead of it in the array. Since the forces between the molecules are symmetric, each pair-wise interaction between molecules is thus considered only once. The state associated with the molecules is then updated.

Although some portions of the molecule state are modified at each interaction, others are only changed between time steps. Most synchronization is done using barriers, although there are also several variables holding global properties that are updated continuously and are protected using locks.

5.3.9 Blackscholes

The *Blackscholes* application is an Intel RMS benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE).

Blackscholes stores the portfolio with numOptions derivatives in array OptionData. The program divides the portfolio into a number of work units equal to the number of threads and processes them concurrently. Each thread iterates through all derivatives in its contingent and calls function BlkSchlsEqEuroNoDiv for each of them to compute its price.

5.3.10 Swaptions

The *Swaptions* application is an Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. The HJM framework describes how interest rates evolve for risk management and asset liability management for a class of models. Its central insight is that there is an explicit relationship between the drift and volatility parameters of the forward-rate dynamics in a noarbitrage market. Because HJM models are non-Markovian the analytic approach of solving the PDE to price a derivative cannot be used. Swaptions therefore employs Monte Carlo (MC) simulation to compute the prices.

The program stores the portfolio in the swaptions array. Each entry corresponds to one derivative. Swaptions partitions the array into a number of blocks equal to the number of threads and assigns one block to every thread. Each thread iterates through all swaptions in the work unit it was assigned and calls the function HJM Swaption Blocking for every entry in order to compute the price. This function invokes HJM SimPath Forward Blocking to generate a random HJM path for each MC run. Based on the generated path the value of the swaption is computed.

Chapter 6

Experimental Evaluation

This chapter evaluates performance and energy of the PS directory and the PS cache proposed in this work. In addition, area for the PS directory is also analyzed.

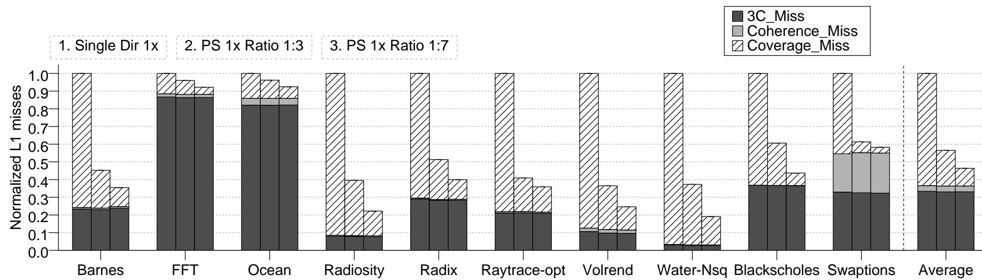
6.1 PS Directory Results

During simulations a L1 cache with 4 ways, a L2 cache with 8 ways and a single directory cache witch 256 sets and 4 ways, same number of entries as an L1 cache, are assumed.

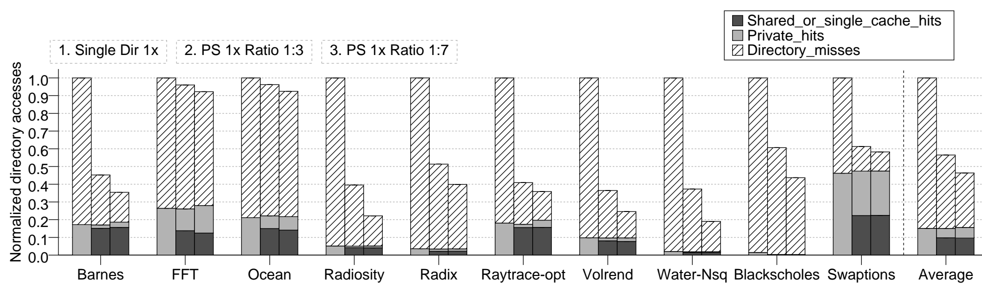
6.1.1 Performance

This section analyzes the performance of the proposed PS directory for a 16-tile CMP compared to a traditional *single cache* directory with the same number of entries. The performance of the directory cache may significantly affect system performance. Effectively, every time a directory entry is evicted, invalidation messages are sent to the corresponding processor caches for coherence purposes. These invalidations will cause *coverage misses* upon a subsequent memory request to those blocks, therefore impacting on the final performance.

Figure 6.1(a) shows the L1 cache misses classified in *3C* (i.e., cold or compulsory, capacity and conflict), *Coherence*, and *Coverage*. As observed, the PS directory cache is able to remove most coverage misses caused by a *single cache* approach with the same number of entries (by 84.2% and 68.2% for 1:7 and 1:3 private-to-shared ratios, respectively). Essentially, this reduction in coverage misses comes from removing conflict misses in the directory cache. These conflict misses are mainly caused by private directory entries, as shown in Section 3.2. Therefore, by adding two additional ways to



(a) Normalized L1 cache misses.



(b) Normalized directory accesses.

Figure 6.1: Normalized misses with respect to a conventional single-cache directory.

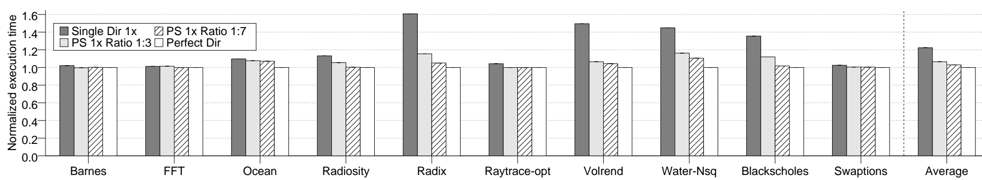


Figure 6.2: Normalized execution time with respect to a perfect directory.

the Private cache (at the cost of reducing the number of sets, so the number of entries remains the same) most directory conflict misses can be reduced.

To illustrate where benefits come from, lets study the 1:3 ratio. This ratio provides the same number of sets to the Shared and to the Private cache, with 2-way and 6-way associativity, respectively. In other words, this PS organization has exactly the same number of sets as the 4-way single cache, and on average, the same number of ways per set. Thus this scenario clearly shows that critical *private* sets are efficiently handled by the PS scheme. To sum up, performance benefits mainly come from identifying that the private entries suffer from conflict misses and assigning associativity in a smarter way to different structures depending on the requirements of the type of the entries.

Performance of a multilevel directory cache can be defined as the number of coherence requests that find the required coherence information in the directory, that is, as the overall directory hit ratio regardless of the directory structure that provides such information. Figure 6.1(b) presents the accesses to each cache organization and from those which miss and which hit and in the case of a hit the directory structure that has the entry (Private or Shared caches).

Notice that, as expected by design, the Private cache shows on average a poor hit ratio despite the much higher number of entries (3× and 7× times the entries of the Shared cache), and most accesses hit in the Shared cache, which corresponds to the smaller and faster directory structure. Remember that each hit in the Private cache refers to a private block that becomes shared. Although the 1:7 ratio could seem to have a Shared cache too small, it provides on average better results than the 1:3 ratio.

Reducing both the number of coverage misses in the processor caches and the access latency to the directory cache translate into improvements in execution time as shown in Figure 6.2. This figure compares the performance of the studied schemes with that of a perfect directory cache. A directory cache is referred to be perfect when it does not incur in performance degradation, that is, there are no coverage misses. Therefore, a perfect directory cache provides the same performance as a duplicate tags approach but more scalability. Nevertheless, unlike the proposed scheme, there is no realizable implementation of a duplicate tag approach. Compared to the single directory cache, the PS directory reduces execution time on average by 13.6% and 11.1% for the 1:7 and 1:3 shared-to-private ratios, respectively. Compared to the perfect cache, the Single cache increases the execution time on average by 22.3%, yielding in some case to unacceptable performance (e.g. by 60% in Radix). However, performance drops of the proposal with respect to the perfect cache are by 6.4% and 2.9% for the ratios 1:3 and 1:7 respectively. In short, results probe that the PS directory is a simple and effective

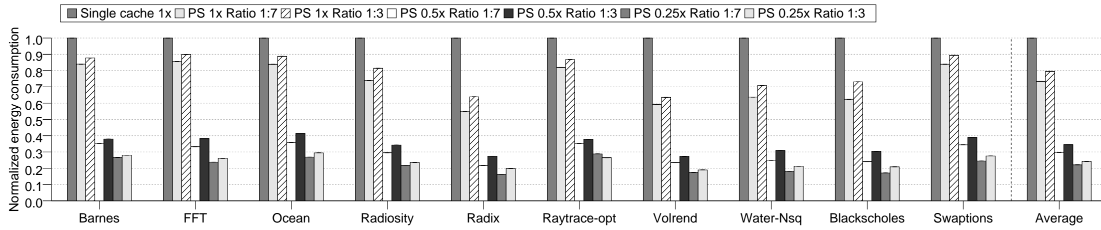


Figure 6.3: Normalized energy consumed by the directory with respect to a single-cache directory. eDRAM technology is used in the Private directory cache.

design since it is able to reach performance close to a perfect directory.

6.1.2 Area and Energy Analysis

This section shows how the PS directory is able to reduce area and energy consumption while increasing performance.

Table 6.1 shows the area required for different PS schemes and the single directory cache. As expected, all the PS configurations are able to reduce area, even those entirely implemented with SRAM technology. In particular, compared to the single cache, the PS configurations with SRAM Private caches save 18.51% and 25.48% of area for 1:3 and 1:7 shared-to-private ratios, respectively. These savings come because the Private cache does not include the sharer vector field. In addition, when eDRAM technology is considered, these reductions increase up to 25.02% and 33.12% for 1:3 and 1:7 shared-to-private ratios, respectively.

On the other hand, the PS directory attacks energy consumption by design, especially leakage, since it uses two low hardware complexity structures with less storage capacity than a single conventional directory cache.

In Figure 6.3 we show the total energy consumed during the benchmarks execution,

Table 6.1: Area (in $mm^2 * 1000$) of the different PS configurations for 16 cores compared with the Single cache directory.

Directory	Private	Shared	Private	Total	Area (%)
Single		19.51	–	19.51	100.00%
PS 1:3	SRAM	6.40	9.50	15.90	81.49%
	eDRAM	6.40	8.22	14.63	74.98%
PS 1:7	SRAM	3.45	11.08	14.54	74.52%
	eDRAM	3.45	9.60	13.05	66.88%

normalized with respect to a single-cache directory using SRAM technology in the Private directory cache. We can observe that a PS directory with the same number of entries as a single cache directory can save around 27% and 20.5% of the energy consumption of the single cache directory for the 1:7 and the 1:3 ratios, respectively. When taking eDRAM technology in the Private directory cache into consideration, the savings are as high as 87.3% and 81.3% for the 1:7 and the 1:3 ratios, respectively.

Figure 6.4 depicts the required silicon area per-core for the studied directory configurations. As observed, the conventional directory requires more area than any of the PS configurations, which grows exponentially with the number of cores. The PS directory is able to reduce by 84.3% (ratio 1:7) and 73.3% (ratio 1:3) the area required by the conventional directory cache for a 1024–core system in spite of implementing both the same number of entries. Thus, the PS directory overcomes one of the biggest problems that sparse directories present, namely, their scalability.

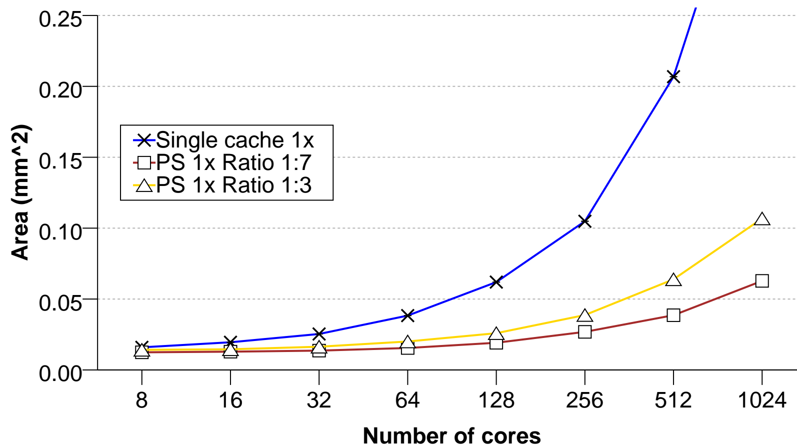
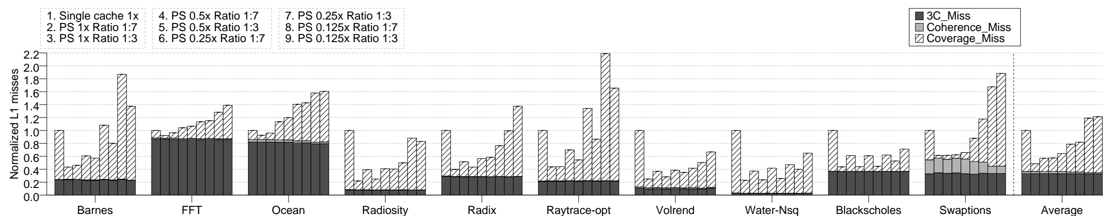


Figure 6.4: Scalability analysis in terms of area.

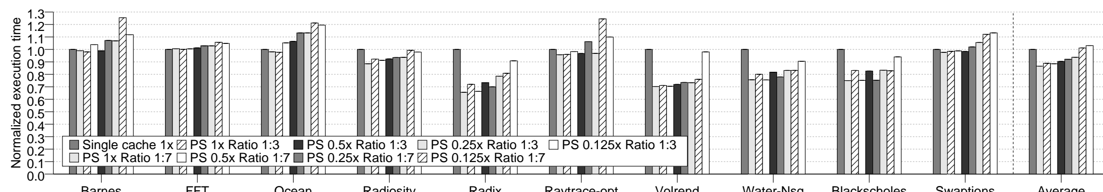
6.1.3 Directory Coverage Ratio Analysis

In this section we evaluate the impact on performance of reducing the directory coverage ratio, that is, the number of entries in the PS directory cache. As the number of entries is reduced in the directory cache, a degradation in performance is expected, but at the same time area and energy consumption will benefit. The ideal directory cache size is the one that entails negligible impact on performance while at the same time allows reductions in area requirement and power consumption.

Figure 6.5(a) shows the L1 cache misses classified in *3C*, *Coherence*, and *Coverage*.



(a) Normalized L1 cache misses.



(b) Normalized execution time.

Figure 6.5: Normalized performance with respect to a conventional single-cache directory.

This figure is similar to Figure 6.1(a) but for this study we show different coverage ratios for the directory cache. As shown, with the only exception of a 0.125× coverage ratio, we still incur in less L1 cache misses than a single cache directory, on average, allowing a significant reduction in directory cache area. For a 0.125× coverage ratio the increase in the number of cache misses is roughly 20%, on average.

This increase in coverage misses translates into a degradation in execution time with respect to a 1× coverage ratio PS directory. However, with respect to a single directory, the execution time is still reduced, even for a 0.125× coverage ratio, as Figure 6.5(b)

Table 6.2: Area (in $mm^2 * 1000$) of the different PS configurations for 16 cores compared with the 1× Single cache directory.

Coverage	Directory	Private	Shared	Private	Total	Area (%)
1×	Single		19,51	–	19,51	100,00%
	PS 1:3	SRAM	6,33	9,50	15,83	81,15%
	PS 1:7	SRAM	3,28	11,08	14,37	73,65%
	PS 1:3	eDRAM	6,33	8,22	14,56	74,61%
	PS 1:7	eDRAM	3,28	9,60	12,88	66,02%
0.5×	PS 1:3	eDRAM	3,28	4,80	8,09	41,47%
	PS 1:7	eDRAM	1,74	4,80	6,55	33,60%
0.25×	PS 1:3	eDRAM	1,74	3,01	4,76	24,39%
	PS 1:7	eDRAM	0,84	3,01	3,85	19,76%

Table 6.3: Static and dynamic energy consumption of the different PS configurations for 16 cores compared with the 1× Single cache directory.

Configurations			Pleakage (mW)			Eread (pJ)		
Coverage	Directory	Private	Shared	Private	Total	Shared	Private	Total
1×	Single		4,2346	–	4,2346	0,0048	–	0,0048
	PS 1:3	SRAM	1,1877	2,2572	3,4450	0,0027	0,0028	0,0055
	PS 1:7	SRAM	0,6404	2,6334	3,2739	0,0016	0,0032	0,0049
	PS 1:3	eDRAM	1,1877	0,5123	1,7001	0,0027	0,0067	0,0094
	PS 1:7	eDRAM	0,6404	0,5977	1,2382	0,0016	0,0078	0,0094
0.5×	PS 1:3	eDRAM	0,6404	0,4114	1,0518	0,0016	0,0035	0,0052
	PS 1:7	eDRAM	0,3650	0,4799	0,8450	0,0010	0,0041	0,0052
0.25×	PS 1:3	eDRAM	0,3650	0,3276	0,6927	0,0010	0,0027	0,0037
	PS 1:7	eDRAM	0,2181	0,3822	0,6003	0,0007	0,0032	0,0039

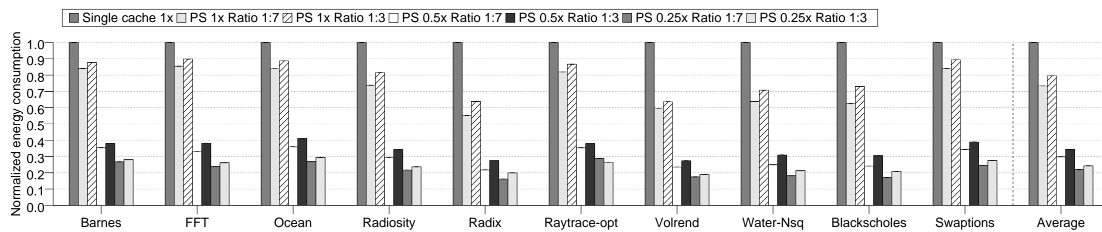


Figure 6.6: Normalized energy consumed by the directory with respect to a single-cache directory.

shows. Therefore, if reducing silicon area is a target design goal, which would be the main reason for a lower coverage ratio, one can opt for reducing the area overhead of the directory without losing performance with respect to a conventional directory. The PS directory is able to improve the performance of a conventional single directory cache while using 8 times less entries.

Table 6.2 shows the area required for different PS schemes with different coverage ratios[1] and the single directory cache. As expected, all the PS configurations are able to reduce area, even those with the same number of entries (1×) as the conventional directory cache. This is due to the fact that the Private cache does not implement the sharer vector field. When the directory coverage ratio is reduced (0.5× and 0.25× coverage ratios), area savings significantly increase up to 80, 24% for the 0.25× 1:7 configuration, while still improving the system performance (as shown previously). Comparing the results for both shared-to-private ratios, we can see that configurations with 1:7 ratio are more area efficient since they are able to reduce area from 12% up to 26% (depending on the directory coverage ratio) over configurations with 1:3 ratio, while providing similar performance results.

Table 6.3 shows the energy (dynamic and static) consumed by the PS directory cache with different coverage ratios and compared to the 1× single directory cache. As it can be seen, the 1× and 0.5× PS configurations consume more dynamic energy per access than the conventional cache, but this is highly offset by the much lower leakage consumed by the PS configurations, which is highly reduced even using SRAM technology in the Private cache. Leakage is reduced from 19% for the SRAM 1× 1:3 configuration up to 86% in the eDRAM 0.25× 1:7 configuration. Comparing 1:3 and 1:7 shared-to-private ratios, we can see that 1:7 configurations are able to reduce leakage consumption from 5% up to 15% with respect to the 1:3 configurations. Taking into account these values, Figure 6.6 shows the energy consumed during the execution of the benchmarks by the PS directory normalized with respect to the energy consumption of a single-cache directory. Smaller coverage ratios lead to less energy consumed at the cost of performance degradation.

Figure 6.7 depicts the area per core scalability for the studied directory configurations. As observed, the conventional directory exhibits the worst area behavior with significant area difference with the PS directory configurations. These differences increase as the number of cores does. Even it requires for 128 cores more area than all the PS configurations with up to 1024 cores, with the only exception of PS 1× 1:3.

As stated in previous section for a 1× coverage configuration, the PS directory

[1] Results for 0.125× are not shown because CACTI is not able to provide results for so small caches.

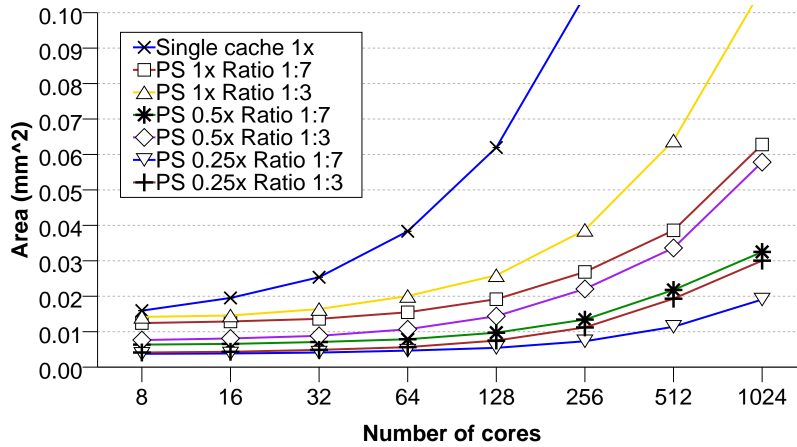


Figure 6.7: Scalability analysis in terms of area.

is able to reduce to 26, 71% (ratio 1:7) and 15, 71% (ratio 1:3) the area required by the conventional directory cache for a 1024–core system using both the same number of entries. Of course, the area is further reduced when using smaller coverage ratios. In particular, for the 0.5× PS configurations, the PS directory is able to require only 14, 47% (ratio 1:3) and 8, 13% (ratio 1:7) the area required by the single cache directory, and for the 0.25× PS configurations only 7, 52% (ratio 1:3) and 4, 77% (ratio 1:7) the area required by the single cache directory.

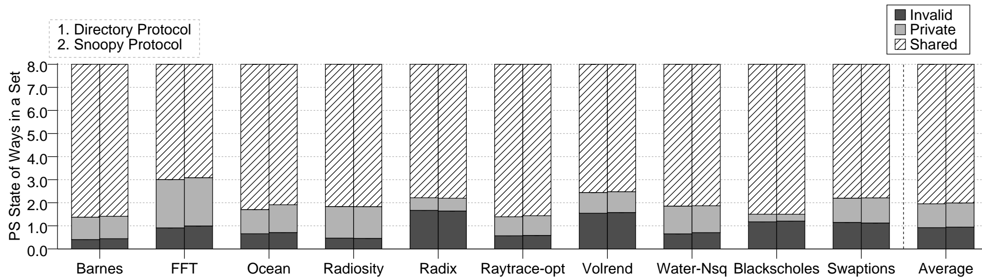
6.2 PS Cache Results

During simulations a L1 cache with 8 ways and a L2 cache with16 ways are assumed.

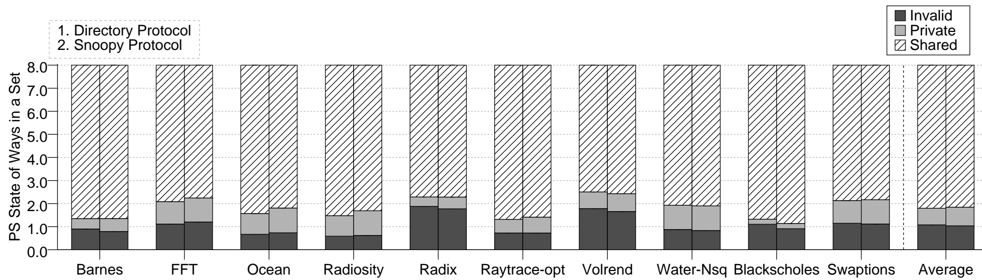
6.2.1 Private-Shared Blocks Behaviour Analysis

Benefits of the proposal will depend on the average number of ways that are looked up by the cache accesses. This number would mainly change depending on which cache we are accessing to (L1 or L2), on the applications behavior and on the type of block we are looking for.

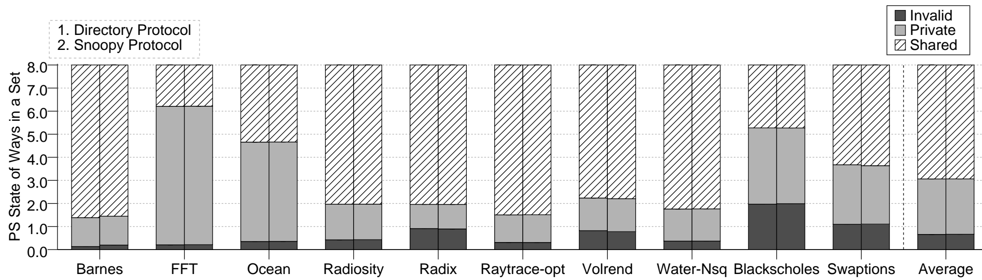
Figure 6.8(a) shows the average number of blocks of each type in the 8-way set associative L1 cache. Results are shown for the snoopy and directory-based protocols considered in this work. As observed, on average, there is just a single private block, whose access would result in important energy savings. However, since some applications, e.g. Ocean, issue some bursts of loads to shared blocks followed by burst of



(a) Average number of ways in a set of each type



(b) Average number of ways of each type seen when looking for a shared block

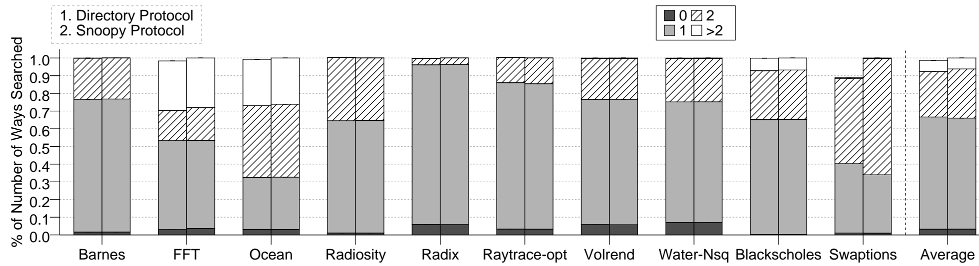


(c) Average number of ways of each type seen when looking for a private block

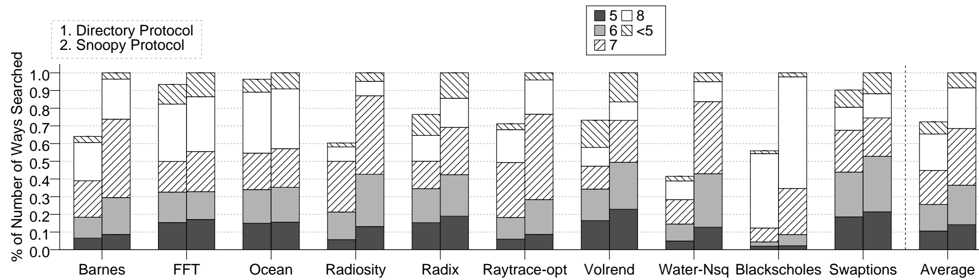
Figure 6.8: Average number of blocks of each type in the L1 cache for both studied protocols.

writes; the final number of searched ways will vary according to the type of block we are looking for and the application dynamic behavior. That is, the average number of blocks of each type seen on the arrival of a request to a private block can widely differ from that seen on the arrival of a request to a shared block. This claim can be observed in Figure 6.8(b) and Figure 6.8(c).

To provide further insights on how much energy savings the proposal is able to bring, Figures 6.9(a) and 6.9(b) show the distribution of the number of ways searched on each access on the arrival of a private or shared request respectively. As observed, most applications search more ways when accessing shared blocks than when accessing



(a) Number of ways searched when looking for a private block



(b) Number of ways searched when looking for a shared block

Figure 6.9: Distribution of the number of ways searched in the L1 cache normalized with respect to the snoopy protocol.

private blocks, with only few exceptions such as *Barnes* and *Radiosity*. An interesting observation is that when looking for a private block, most of the times, i.e. over 93% of the accesses, just look up one or two ways. Benefits, are lower when looking up a shared block, but even in this case, around 40% of times, five or less ways are looked up, which will also bring important energy savings. Regarding protocols, two main observations can be drawn. First, it can be appreciated that the rate of shared to private blocks looked up is quite similar in both protocols regardless if a shared or private block is looked up. Second, major differences among protocols only appear when looking for a shared block. In this case, a directory based protocol reduces the number of lookups on average around 30% with respect to the snoopy protocol. Moreover, this reduction can be as high as 70% in Water-Nsq when looking a shared block.

As mentioned above, the proposal can be applied to any level of the cache hierarchy, thus this section also explores the benefits on the L2 cache. Figure 6.10 depicts the average number of private and shared ways looked up, on average, on each L2 access. As can be seen, with only few exceptions (i.e. *Ocean*), most of the ways (12 of 16) are in an invalid state over the execution time. This is due to the characteristics of the protocol implementation. In the devised implementation, when a block moves from L2

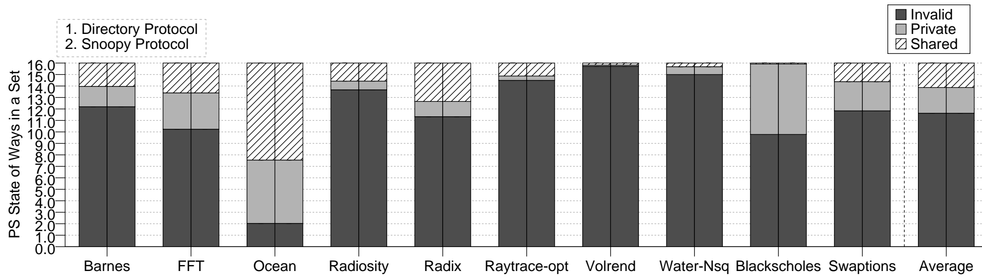
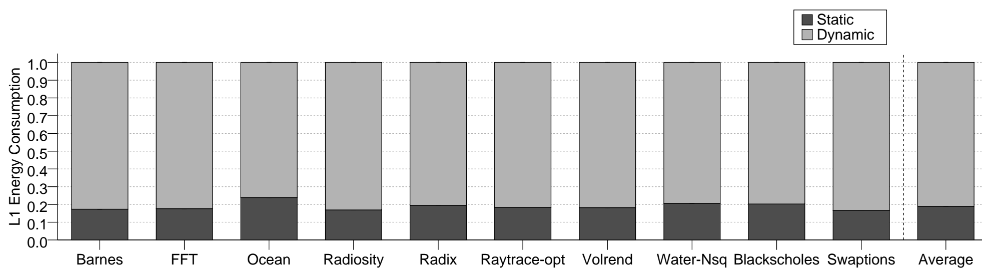
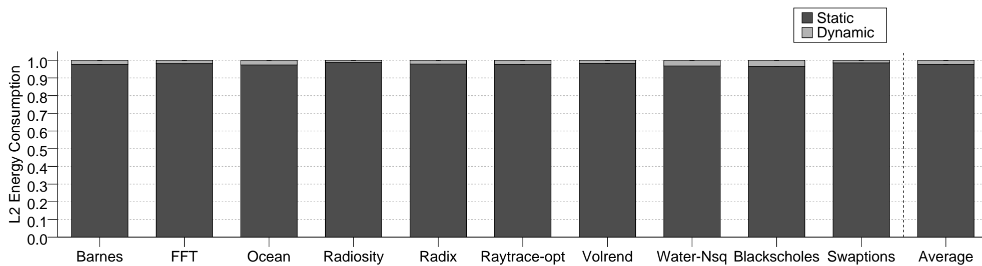


Figure 6.10: Average state of ways during lookups in the L2.



(a) L1 Cache



(b) L2 Cache

Figure 6.11: Static and dynamic energy comparison.

to L1, the associated L2 entry is invalidated. Unlike L1 caches, the amount of private ways looked up is close to that of shared ways. This number is as low as only 2 ways from 16, which indicates that potential benefits achieved by any mechanism addressing energy savings will be minor when compared to those of the L1 cache. In addition, the high number of invalid ways, indicates that significant static energy reduction can be achieved by just turning off all invalid ways.

6.2.2 Power Consumption

This section analyzes the impact of the PS cache in the energy consumption.

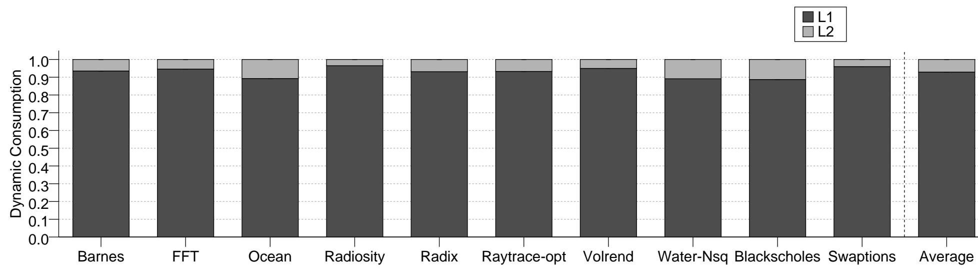


Figure 6.12: Comparison of dynamic energy consumption in L1 and L2.

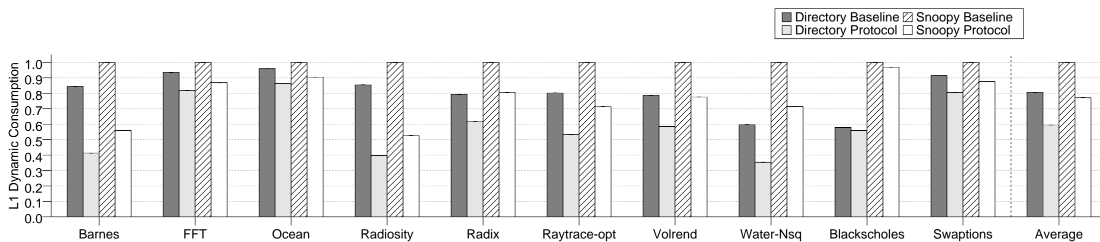


Figure 6.13: Reduction of the dynamic energy consumption.

Figures 6.11(a) and 6.11(b) show a comparison between both dynamic and static energy consumption of all evaluated benchmarks with a conventional directory protocol in both caches of our cache hierarchy. All applications behave mostly the same with 81.07% of the total energy consumed in the L1 being dynamic, and only 2.24% in the L2. This is to be expected, since most of the accesses are filtered in the first-level cache and only few of them arrive to the L2, thus being its static energy consumption much higher.

Since the L2 cache is much bigger than the L1 one, it is quite obvious that its static energy consumption will be much larger, and hence there is minor interest in studying it. On the other hand, a comparison between the dynamic energy in both members of the hierarchy is shown in Figure 6.12. Only 7.09% of the total dynamic consumption of the cache system comes from the L2, which means that our proposal will be much more effective on the highly accessed first-level caches rather than the less frequented ones.

Therefore, for now on, we focus the analysis on how much our proposal can reduce the energy consumption in the first-level cache. Figure 6.13 shows the dynamic energy consumed by a conventional directory and snoopy protocol, where all ways in the cache sets are searched, along with the two variants of our proposal for each of the coherence

protocols.

Experimental results show, on average, a 22.88% decrease in dynamic power consumption for snoopy protocols and a 20.09% for directory protocols. Furthermore, a snoopy protocol with the PS cache architecture can consume less than a conventional directory one. This is simply achieved through the selective search within the different ways of a set provided by our PS bit.

As suggested in the previous section, applications with a large number of private-block lookups, obtain higher energy reductions. For example, *Barnes* reduces dynamic power consumption by 44.04% and 51.09%, for snoopy and directory protocols respectively, and *Radiosity* by 47.51% and 53.47% respectively. On the other hand, applications with a low number of private-block lookups offer no such benefits. Best example of this scenario is the *Blackscholes* benchmark, which only reduces the power consumption by 3.07% and 3.44% respectively.

Chapter 7

Conclusions and Future Work

This chapter draws some concluding remarks regarding the proposals presented in this work. Also future work and a list of publications related with this thesis are presented.

7.1 Conclusions

One of the major design concerns in current high-performance chip multiprocessors is the power consumption, which increases as the number of core counts grows. On-chip caches often consume a significant fraction of the total power budget, and important research has focused on reducing energy consumption in these memory structures at the cost of performance.

This work presents two main contributions: the PS directory and the PS cache. Below we summarize the conclusions for each of them.

PS directory: This work identifies five characteristics that clearly difference the behavior of private and shared blocks from the directory point of view. Based on these differences, we introduce the PS directory, a directory cache that uses two different cache structures, each one tailored to one of these types of blocks (i.e., private and shared). The Shared directory cache, which tracks shared blocks is small, with low associativity and fast. The Private directory cache is aimed at tracking private blocks, which are highly dominant in current workloads. This structure does not store the sharer vector, is larger than the shared cache, and it has higher associativity.

Experimental results for 16 cores show that, compared to a single directory cache with the same number of entries, the PS directory improves performance by 14% due to the separate treatment of private and shared blocks. Additionally, directory area is reduced by 26.35% mainly due to not storing the sharer vector for the private blocks,

and by 33.98% when eDRAM technology is considered for the design of the Private cache. In terms of energy consumption, reductions around 27% are achieved. Thus, we can conclude that by saving area and energy, the proposal provides scalability for the widely preferred sparse directories. Moreover, this is achieved by reaching almost the same performance as the duplicate tags approach (i.e., perfect directory) but with a feasible implementation that scales with the number of cores.

PS cache: In this work, we have proposed an energy-efficient cache design which only accesses a subset of the set ways without hurting performance. Blocks are classified at page level as shared or private according to the TLB information. In addition, the proposal adds a single bit attached to each cache line, which activates only the word line if the blocks whose type matches the one provided by the TLB. In this way dynamic energy is largely saved. Moreover, coherence requests to a remote cache only access the shared subset of ways. This cache design can be implemented in all cache levels of the cache hierarchy, although the most frequently accessed one will achieve the best energy reductions.

Experimental results have proven that, although recent work has concentrated on reducing dynamic energy in mid-level caches (e.g. L2 caches), major benefits can be achieved by focusing in first level caches, since these memory structures dominate, by far, the dynamic energy consumed along the cache hierarchy.

Results have shown that, compared to conventional directory or snoopy protocols where all ways are searched, the PS cache can reduce the dynamic power consumption by a 22.88% and 20.09% for directory and snoopy protocols respectively in the L1.

An interesting observation is that when the proposal is applied to CMPs working with a energy-hungry snoop-based protocol, the consumed energy is on par with that consumed by a *power-aware* directory-based protocol.

7.2 Future Work

As for future work we plan to study and evaluate other filtering mechanisms that can be applied to any kind of application (multithreaded or multiprogrammed). One possibility is to avoid accessing all the set ways on a cache access by using a few bits of the block address to select the ways that are finally accessed. This will work mostly the same as the PS cache in this work but the benefits obtained from this new (or any other) filter will surely vary. On the directory side, one possibility that we are taking into consideration is the viability of an asymmetric directory cache, which would have some ways like the ones in the private cache and other like the ones in the shared

one. One problem arises with this scheme: the migration of blocks in the ways. If this number of migrations were low, though, the potential gains of this would be of interest.

7.3 Thesis-related Publications

Here we will show a list of some publications related to this work:

Joan J. Valls, Alberto Ros, Julio Sahuquillo and María E. Gómez. El directorio PS: Una caché de directorio multinivel escalable para CMPs. In *XXIII edición Jornadas de Paralelismo SARTECO*, 19-21 September 2012.

Joan J. Valls, Alberto Ros, Julio Sahuquillo, María E. Gómez and José Duato. PS-Dir: A Scalable Two-Level Directory Cache. In *21st International Conference on Parallel Architectures and Compilation Techniques (PACT-2012)*, 19-23 September 2012.

Joan J. Valls, Alberto Ros, Julio Sahuquillo and María E. Gómez. PS Directory: A Scalable Multilevel Directory Cache for CMPs. In *IEEE Transactions on Parallel and Distributed Systems (TPDS) (TPDS-2014)*, submitted

Joan J. Valls, Alberto Ros, Julio Sahuquillo and María E. Gómez. PS-Cache: An Energy-Efficient Cache Design for Chip Multiprocessors. In *22st International Conference on Parallel Architectures and Compilation Techniques (PACT-2013)*, submitted

Joan J. Valls, Alberto Ros, María E. Gómez and Julio Sahuquillo. PS-Cache: An Energy-Efficient Cache Design for Chip Multiprocessors. In *9th International Conference on High-Performance and Embedded Architectures (HiPEAC TACO-2014)*, submitted

Bibliography

[AGGD01] Manuel E. Acacio, José González, José M. García, and José Duato. A new scalable directory architecture for large-scale multiprocessors. In *7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 97–106, January 2001.

[AGGD05] Manuel E. Acacio, José González, José M. García, and José Duato. A two-level directory architecture for highly scalable cc-NUMA multiprocessors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 16(1):67–79, January 2005.

[Alb99] David H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *32nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 248–259, December 1999.

[Ali12] Mohammad Alisafaee. Spatiotemporal coherence tracking. In *45th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 341–350, December 2012.

[APJ09] Niket Agarwal, Li-Shiuan Peh, and Niraj K. Jha. In-Network Snoop Ordering (INSO): Snoopy coherence on unordered interconnects. In *15th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 67–78, February 2009.

[AW06] Alaa R. Alameldeen and David A. Wood. Ipc considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, July 2006.

[BGM00] Luiz A. Barroso, Kourosh Gharachorloo, and Robert McNamara, et al. Piranha: A scalable architecture based on single-chip multiprocessing. In *27th Int'l Symp. on Computer Architecture (ISCA)*, pages 12–14, June 2000.

[BJ11] Rajeev Balasubramonian and Norman Jouppi. *Multi-Core Cache Hierarchies*. Morgan & Claypool Publishers, 1st edition, 2011.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, October 2008.

[CG96] Brad Calder and Dirk Grunwald. Predictive sequential associative cache. In *2nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 244–253, February 1996.

[Che93] Guoying Chen. Slid - a cost-effective and scalable limited-directory scheme for cache coherence. In *5th Int'l Conference on Parallel Architectures and Languages Europe (PARLE)*, pages 341–352, June 1993.

[CKA91] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *4th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 224–234, April 1991.

[CKD+10] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache hierarchy and memory subsystem of the AMD opteron processor. *IEEE Micro*, 30(2):16–29, April 2010.

[CRG+11] Blas Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *38th Int'l Symp. on Computer Architecture (ISCA)*, pages 93–103, June 2011.

[CSL+06] Jason F. Cantin, James E. Smith, Mikko H. Lipasti, Andreas Moshovos, and Babak Falsafi. Coarse-grain coherence tracking: RegionScout and region coherence arrays. *IEEE Micro*, 26(1):70–79, January 2006.

[FKM+02] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, Trevor MudgeStefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Drowsy caches: Simple techniques for reducing leakage power. In *29th Int'l Symp. on Computer Architecture (ISCA)*, pages 148–157, May 2002.

[FLKBF11a] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo directory: A scalable directory for many-core systems. pages 169 –180, feb. 2011.

[FLKBF11b] Michael Ferdman, Pejman Lotfi-Kamran, Ken Balet, and Babak Falsafi. Cuckoo directory: A scalable directory for many-core systems. In *17th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 169–180, February 2011.

[GÖF+09] Mrinmoy Ghosh, Emre Özer, Simon Ford, Stuart Biles, and Hsien-Hsin S. Lee. Way guard: A segmented counting bloom filter approach to reducing energy for set-associative caches. In *Int'l Symp. on Low Power Electronics and Design (ISLPED)*, pages 165–170, August 2009.

[GWM90] Anoop Gupta, Wolf-Dietrich Weber, and Todd C. Mowry. Reducing memory traffic requirements for scalable directory-based cache coherence schemes. In *Int'l Conference on Parallel Processing (ICPP)*, pages 312–321, August 1990.

[GWX+10] Song-Liu Guo, Hai-Xia Wang, Yi-Bo Xue, Chong-Min Li, and Dong-Sheng Wang. Hierarchical cache directory for cmp. *Journal of Computer Science and Technology*, 25(2):246–256, March 2010.

[HDH11] Hemayet Hossain, Sandhya Dwarkadas, and Michael C. Huang. POPS: Coherence protocol optimization for both private and shared data. In *20th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 45–55, October 2011.

[HFFA09] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *36th Int'l Symp. on Computer Architecture (ISCA)*, pages 184–195, June 2009.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[JNW07] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[KBK02] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In

10th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS), pages 211–222, October 2002.

[KCG+10] Kamil Kedzierski, Francisco J. Cazorla, Roberto Gioiosa, Alper Buyuktosunoglu, and Mateo Valero. EMC^2: Extending magny-cours coherence for large-scale servers. In *2nd Int'l Forum on Next-Generation Multicore/Manycore Technologies*, pages 1–12, June 2010.

[KHM01] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *28th Int'l Symp. on Computer Architecture (ISCA)*, pages 240–251, June 2001.

[KKH10] Daehoon Kim, Jeongseob Ahn Jaehong Kim, and Jaehyuk Huh. Subspace snooping: Filtering snoops with operating system support. In *19th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, September 2010.

[KSSF10] Ron Kalla, Balaram Sinharoy, William J. Starke, and Michael Floyd. Power7: IBM's Next-Generation Server Processor. *IEEE Micro*, 30:7–15, 2010.

[LAMJ10] Yong Li, Ahmed Abousamra, Rami Melhem, and Alex K. Jones. Compiler-assisted data distribution for chip multiprocessors. In *19th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 501–512, September 2010.

[LHK11] Jongmin Lee, Seokin Hong, and Soontae Kim. Tlb index-based tagging for cache energy reduction. In *17th Int'l Symp. on Low Power Electronics and Design (ISLPED)*, pages 85–90, August 2011.

[LMJ12] Yong Li, Rami G. Melhem, and Alex K. Jones. Practically private: Enabling high performance cmps through compiler-assisted data classification. In *21st Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 231–240, September 2012.

[MBJ09] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. Cacti 6.0. Technical Report HPL-2009-85, HP Labs, April 2009.

[MCE02] Peter S. Magnusson, Magnus Christensson, and Jesper Eskilson, et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.

[MH07] Michael R. Marty and Mark D. Hill. Virtual hierarchies to support server consolidation. In *34th Int'l Symp. on Computer Architecture (ISCA)*, pages 46–56, June 2007.

[MH08] Michael R. Marty and Mark D. Hill. Virtual hierarchies. *IEEE Micro*, 28(1):99–109, 2008.

[MS05] Richard E. Matick and Stanley E. Schuster. Logic-based eDRAM: Origins and rationale for use. *IBM Journal of Research and Development*, 49(1):145–165, 2005.

[MS09] Jiayuan Meng and Kevin Skadron. Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling. pages 282–288, October 2009.

[MSB05] Milo M.K. Martin, Daniel J. Sorin, and Bradford M. Beckmann, et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 33(4):92–99, September 2005.

[ON90] Brian W. O'Krafka and A. Richard Newton. An empirical evaluation of two memory-efficient directory methods. In *17th Int'l Symp. on Computer Architecture (ISCA)*, pages 138–147, June 1990.

[PhYF+00] Michael Powell, Se hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *Int'l Symp. on Low Power Electronics and Design (ISLPED)*, pages 90–95, July 2000.

[PSNB10] Seth H. Pugsley, Josef B. Spjut, David W. Nellans, and Rejeev Balasubramonian. SWEL: Hardware cache coherence protocols to map shared data onto shared caches. In *19th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 465–476, September 2010.

[RAG10] Alberto Ros, Manuel E. Acacio, and José M. García. A scalable organization for distributed directories. *Journal of Systems Architecture (JSA)*, 56(2-3):77–87, February 2010.

[RCFP+12] Alberto Ros, Blas Cuesta, Ricardo Fernández-Pascual, Maria E. Gómez, Manuel E. Acacio, Antonio Robles, José M. García, and José Duato. Ex-

tending magny-cours cache coherence. *IEEE Transactions on Computers*, 61(5):593–606, May 2012.

[RK12] Alberto Ros and Stefanos Kaxiras. Complexity-effective multicore coherence. In *21st Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 241–252, September 2012.

[SBB07] Manish Shah, Jama Barreh, and Jeff Brooks, et al. UltraSPARC T2: A highly-threaded, power-efficient, SPARC SoC. In *IEEE Asian Solid-State Circuits Conference*, pages 22–25, November 2007.

[SK12] Daniel Sanchez and Christos Kozyrakis. Scd: A scalable coherence directory with flexible sharer set encoding. In *18th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 129–140, February 2012.

[SKT+05] B. Sinharoy, R N. Kalla, J M. Tendler, R J. Eickemeyer, and J B. Joyner. POWER5 System Microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005.

[SPJ+12] Karthik T. Sundararajan, Vasileios Porpodas, Timothy M. Jones, Nigel P. Topham, and Björn Franke. Cooperative partitioning: Energy-efficient cache partitioning for high-performance cmps. In *18th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 311–322, February 2012.

[TDF+02] J M. Tendler, J S. Dodson, J S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002.

[VSP+09] Alejandro Valero, Julio Sahuquillo, Salvador Petit, Vicente Lorente, Ramon Canal, Pedro López, and José Duato. An Hybrid eDRAM/SRAM Macrocell to Implement First-Level Data Caches. In *Proceedings of the 42th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 213–221, New York, NY, USA, 2009. ACM.

[WLZ+09] Xiaoxia Wu, Jian Li, Lixin Zhang, Evan Speight, Ram Rajamony, and Yuan Xie. Hybrid Cache Architecture with Disparate Memory Technologies. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 34–45, New York, NY, USA, 2009. ACM.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Int'l Symp. on Computer Architecture (ISCA)*, pages 24–36, June 1995.

[ZSQM09] Jason Zebchuk, Vijayalakshmi Srinivasan, Moinuddin K. Qureshi, and Andreas Moshovos. A tagless coherence directory. In *42nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 423–434, December 2009.