Adaptive Memory Hierarchies for Next Generation Tiled Microarchitectures

Enric Herrero Abellanas

Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

Advisors:

José González González (Intel Barcelona)

Ramon Canal Corretger (Universitat Politècnica de Catalunya)

A thesis submitted in fulfillment of the requirements for the degree of

Doctor of Philosophy / Doctor per la UPC

2011 July

Acknowledgements

En primer lloc vull agraïr l'ajut que m'han donat durant aquests anys els meus directors de tesi, el Pepe i en Ramon. Durant aquests anys m'han iniciat en el món de la recerca. Gràcies a ells he pogut conèixer el que és investigar, que al cap i a la fi és aprendre contínuament i saber qüestionar el que s'aprèn. Són uns coneixements que de ben segur em serviran de molt en el futur.

També vull agraïr al Dean el suport que em va donar durant la meva estada a San Diego i que em va facilitar molt l'adaptació a un país que no coneixia. Thank you Dean. Els meus agraïments també per als membres del tribunal, els seus comentaris han ajudat molt a millorar la qualitat de la tesi. Gràcies també a l'Antonio per donar-me la oportunitat de treballar al grup ARCO i al Mats per introduïr-me en el món de l'arquitectura de computadors.

Vull agraïr als meus pares el suport que m'han donat sempre i sense el qual hagués estat impossible realitzar aquesta tesi. Des de petit m'han ensenyat el valor de la educació i m'han ajudat sempre que ha fet falta. Gràcies.

Un agraïment també al meu germà Guillem, a la famiglia, a la gent de Cardedeu i al sector euetibià que han aconseguit que pogués desconnectar de la feina quan feia falta.

I finalment també agraïr a tots els becaris de la sala C6-E208 (i altres sales...) per els bons moments que hem passat durant aquests anys tot fent un cafè o a fora de la universitat.

Contents

CONTENTS

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Chapter 1

Introduction

Computer microarchitecture has evolved from its beginning with an improvement rate never seen before in other domains. Moore's law [95], which states that the number of transistors that can be placed inexpensively on an integrated circuit doubles every two years, has been kept true thanks to fabrication process improvements and this has allowed computer architects to introduce many optimizations in the computer architecture.

All these improvements have allowed an exponential increase in processor performance. Memory access, however, has not experienced the same improvement rate. Figure 1.1 shows a comparison between the improvements in the processor and the memory system. It can be seen that the gap between both parts has been increasing, forcing new optimizations in the memory hierarchy to alleviate this problem.

One of the solutions to this problem appeared in the early 60's with the introduction of caches [13]. The addition of extra on-chip storage allowed to take advantage of the data locality and showed to be very effective in reducing the number of off-chip accesses. A good measure of this effectiveness is that all current commercial processors make use of this technique. However, the amount of chip transistors is limited and, therefore, the amount of on-chip storage that can be added. This implies that usually not all the application data fits in this storage space and that, as a consequence, cache organizations and policies have a great impact on the system performance.

Another solution to this problem is the usage of Out-of-Order scheduling of instructions [126] which was implemented in the IBM 360/91 in 1966. These type of processors allow to execute instructions as soon as the required data is available and to advance stalled requests. This organization hides some of the memory latency and reduces the impact of the memory gap.

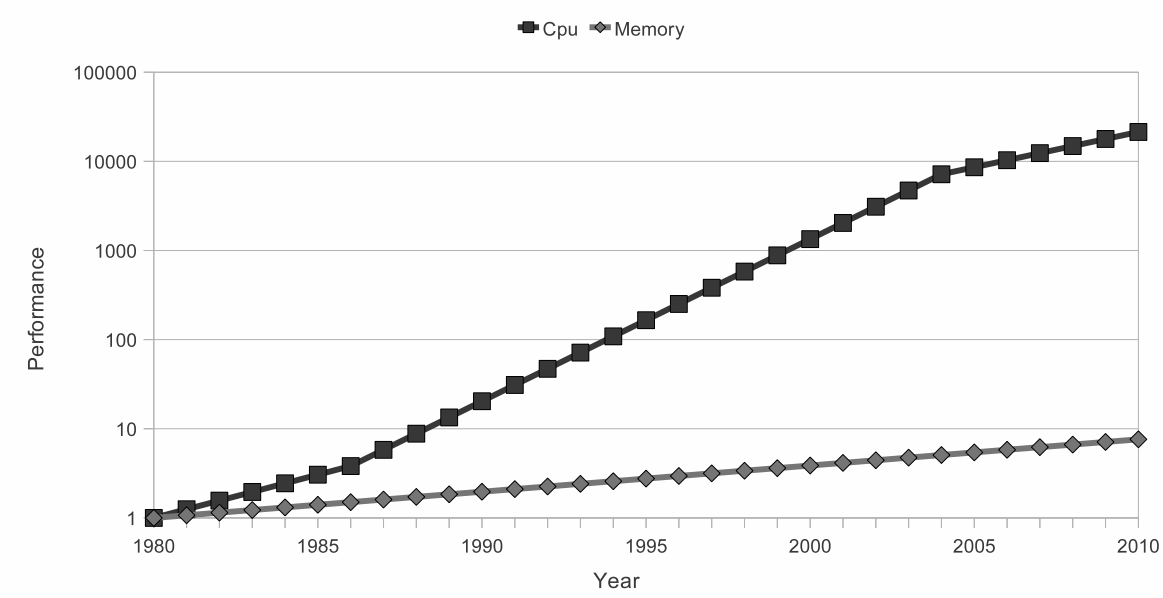


Figure 1.1: Gap between memory and CPU performance [39].

The improvements brought by these solutions along with many others have reduced the impact of increasingly high memory latencies but the problem has remained. Furthermore, the extended usage of chip multiprocessors in the recent years has brought new challenges by adding multiple applications competing for the memory resources. In such environments, cache organization has increased its importance in the overall performance [49].

Therefore, in this thesis we have focused on the design of efficient memory hierarchies for next generation processors, which are a critical part in order to improve processor performance and try to keep sustained performance improvements in the forthcoming years.

1.1 Memory Hierarchy Challenges

There are several challenges brought by the fabrication technologies and the introduction of chip multiprocessors that have driven the research of this thesis. These challenges are the increasing number of processors, the limited off-chip bandwidth and power budget of processors and the arbitration between competing applications in the memory hierarchy.

1.1.1 Increasing number of processors

The increased available on-chip real state and the limited power budget of current processors has forced the apparition of chip multiprocessors, which, rather than providing the

maximum performance from a single core, try to improve the overall performance through parallel execution. Chip multiprocessors started packing a small number of cores but in the recent years several processors have appeared packing up to 80 cores on a single chip [52, 127]. This trend implies that shared structures should be distributed in order to avoid bottlenecks in centralized structures and allow a good scalability in next generation chip multiprocessors.

1.1.2 Increased off-chip miss cost

The cost of off-chip misses has increased in importance due to several reasons that are expected to be maintained in the future [7]. First, memory clock speeds have increased at a slower pace than processor speeds, increasing the miss latency significantly. And second, the usage of chip multiprocessors has introduced the simultaneous execution of more applications in the chip while the pin count has increased very slightly. This implies that the same memory channels must be shared among all threads, increasing the pressure on the memory system.

All these limitations have increased the importance of an efficient use of the off-chip bandwidth. Therefore, new solutions must minimize the number of off-chip misses due to its high cost and in addition optimize the usage of the memory bus to extract the maximum throughput.

1.1.3 Limited power budget

Another important factor in new microarchitectural designs is energy efficiency. Power consumption minimization is a very important issue in current computer architectures. Low energy consumption reduces the energy budget of companies, increases the battery lifetime of portable devices and allows lower operating temperatures that yield a higher reliability and simpler cooling mechanisms.

In the memory hierarchy improvements can come in several ways, by reducing the activity of the system (number of misses, messages, accesses to data structures), introducing simpler mechanisms that consume less energy or avoiding structures that do not increase the performance enough to justify the corresponding increase in energy consumption. In any case, novel improvements in computer architecture must focus not only on performance; energy density and energy consumption must be also taken into account.

1.1.4 Multiprogrammed Environments

Finally, the latest aspect that must be taken into account is the heterogeneous environment of multicore and multithreaded architectures. The simultaneous execution of different applications with different requirements makes necessary to provide an arbitration between them. Not all applications have the same data and instruction locality and, therefore, they require different amount of resources. In addition, some critical applications may require a minimal performance to operate which implies that the arbiter must deal with inter-thread interference and avoid it if possible. All these factors are a challenge in the design of a responsive memory hierarchy, able to provide the best response to all applications. The implementation of an application aware system, however, also has a great potential of improving the overall system performance.

1.2 Contributions

In this thesis we have focused in optimizing the memory hierarchy for next generation multiprocessors taking into account the challenges presented previously. Proposed optimizations range from the cache level to the memory level.

1.2.1 Distributed Coherence Mechanism

Coherence enforcement, as we are going to show, is especially important in multiprogrammed or multithreaded environments. Most of the existing solutions, however, rely in traditional private and shared cache configurations and in most cases make use of centralized structures which limit the scalability of the memory hierarchy. In this thesis we propose a distributed structure, the Distributed Cooperative Caching (DCC) [42], which has the advantages of both private and shared caches and has a better scalability and energy efficiency than existing state of the art configurations. Furthermore, we propose additional improvements to DCC in order to improve the energy-efficiency and network usage. We propose the Distance-Aware and Selective Spilling [44], which increase the spilling efficiency and reduce network usage.

1.2.2 Dynamic and Distributed Cache Allocation

In addition of providing a distributed management of cache resources, an efficient allocation of these resources is mandatory if we want to optimize cache allocation and minimize the

number of off-chip misses. Therefore, in this thesis we propose the Elastic Cooperative Caching (ECC) [43], a distributed cache repartitioning mechanism. ECC is able to detect the memory requirements of each application and reallocate cache resources accordingly during program execution. Most of dynamic repartitioning mechanisms rely on centralized structures. ECC, on the other hand, is able to adapt dynamically using only local information and distributed repartitioning units which make it more suitable for chip multiprocessors with a high number of cores.

All the cache organizations above, in addition to their publication in top conferences like ISCA, PACT or EuroPar, also have led to the publication of a book chapter [16] and an article in the IEEE Transactions of Parallel and Distributed Systems [45].

1.2.3 DRAM organization for multiprogrammed environments

Finally, in this thesis we have focused on optimizing DRAM memories and memory controllers for multiprogrammed environments. Traditional memory bank schedulers have been designed for uniprocessor systems. The advent of chip multiprocessors, however, has led to the apparition of new access patterns and forced a trade-off between memory throughput and fairness. In this thesis we present Thread Row Buffers (TRBs), a modification of the DRAM memory structure which allows to increase the row hit rate significantly under a multiprogrammed environment and avoids the throughput-fairness trade-off. TRBs make possible to design a new generation of bank schedulers focused on an efficient arbitration between threads without hurting memory throughput.

Chapter 2

Background and Motivation

2.1 Introduction

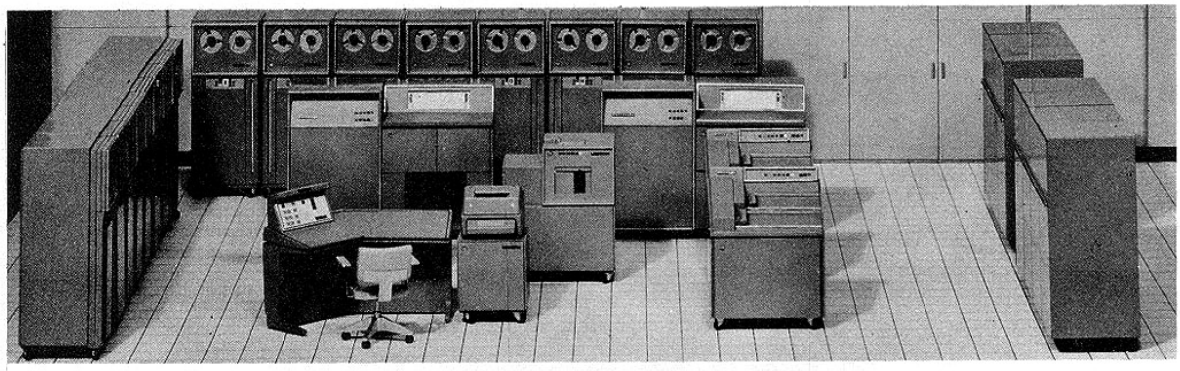
The design of new processors always has been driven by the need of more computing performance. Performance improvements have arrived through the evolution of the fabrication technology, microarchitectural improvements and increase in the parallelism. Parallelism has been exploited at the instruction level, thread level and application level; being the last one the first to be used.

Multiprocessors have existed for many years, the first of them appearing in the early 60's. Burroughs Corporation was the first company to introduce a multiprocessor system[1] with the B5000 introduced in 1961 [24] that operated a second processor in a master-slave configuration. First multiprocessors were asymmetric since the operating system ran in the main processor and the second one executed specific processes, having access to main memory but not to peripherals.

Other vendors also released their multiprocessor like IBM with the model 65MP of System/360 [101] in 1968 or the PDP-10 [11] which added multiprocessing capabilities to its TOPS-10 monitor software in 1972. The PDP-10 was already a Symmetric Multiprocessor (SMP) were all processors had the same importance and capabilities and the operating system managed the shared resources.

Most of these systems, however, used a centralized memory, which posed a limitation in the scalability of the systems. Therefore, in 1973 Bell et al. [12] proposed the usage of independent computer modules with their own memory storage. The implementation of

[1] Some people argues that the BINAC developed by the Eckert-Mauchly Computer Corporation in 1949 was the first one although it was designed to improve the reliability and not to perform different tasks in parallel. [26]



Finally hatched

BURROUGHS ANNOUNCES THE B5000

IT WAS A LONG TIME COMING – but the Burroughs Corporation B5000, that firm's first entry in the solid state computer field, promises to be an interesting addition to the industry.

Featuring an add time of three microseconds and a six microsecond memory cycle, a maximum system may include up to eight memory modules, each containing 4,096 49 bit words.

An interesting feature of the machine lies in the fact that two central processors may be employed simultaneously for what Burroughs calls "true parallel processing."

B5000 will rent from $13,500 to $50,000 a month. The sale price range is $540,000 to $2,000,000. A more detailed article on the B5000 will appear in an early issue of DATAMATION.

Figure 2.1: Article in the Datamation magazine presenting the first multiprocessor on March 1961

multiprocessors with independent memories brought the apparition of new challenges due to the interaction of the different processes in the memory space. This generated two ways of managing the memory space; through message passing or through a shared memory space.

Message passing multiprocessors have independent memory spaces and all communications between processor nodes are explicitly defined by the programming language. Therefore, applications must know in all cases where is the data and ask for it, if necessary, to other nodes. This type of organization is intended for applications with low interaction and optimizes the network usage.

On the other hand, shared memory multiprocessors share the address space among all the nodes, therefore, allowing the programmer to access to all the memory space in a transparent way. This type of organization simplifies significantly the programming and is specially suited for applications with high interaction. The shared memory space, however, generates new requirements to manage the data in a safe way.

In shared memory multiprocessors there are two main types of data, private and shared.

Private data is used by only one processor and, therefore, should be allocated as close as possible to this processor. On the other hand, shared data is used by several processors and should be either stored in a common storage space or replicated in several locations. The existence of shared data requires the implementation of one of the main characteristics of shared memory hierarchies, memory coherence.

A coherent memory system can be defined in a simplistic way as a memory system that enforces that any read of a data item is done to the more updated version of this item. Hennessy and Patterson [39] give a more detailed definition and state that a memory system is coherent if:

- A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.
- A read by a processor to location X that follows a write by another processor to X returns the written value if the read and the write are sufficiently separated in time and no other writes to X occur between the two accesses.
- Writes to the same location are serialized; that is, two writes to the same location by any two processors are seen in the same order by all processors.

In the first multiprocessors, the memory storage was shared and centralized and, therefore, the implementation of coherence was trivial. The usage of a centralized structure does not require to migrate or replicate data and all writes are already serialized in the request queue. Newer shared memory multiprocessors, however, implement independent memory storage which makes coherence enforcement more complex.

2.2 Modern multiprocessors, from NUMA to NUCA

Shared memory multiprocessors with multiple memories are known as Non-Uniform Memory Access (NUMA). Many of these organizations implement migration and replication of data in order to keep the data as close as possible to the user and reduce the access latency and network traffic. These types of systems require the implementation of coherence protocols that store the information of the sharers and the valid copies of the data.

There are three main ways of implementing the coherence protocols, with directories, with a snoop based system and explicitly by software. In implementations with directory [3]

the state of the data is stored in a directory that centralizes all requests and enforces coherence.

Snoop based systems [4, 35], on the other hand, store the information with all the data copies. We have divided this configuration in three types, bus based, fixed mapping and token based. Snoop based systems were traditionally bus based because a bus can be easily monitored by all sharers to update the state when the data is modified. Some implementations, however, allow the implementation of snoop protocols in any kind of network [119]. A variation for NUCA caches is to have a fixed mapping of cache blocks and try to find the block doing a search among the nodes were the data can be. A fixed order is usually defined, accessing first to the local node and having a home node for every block. Some solutions also rely on software hints to use different mappings [38]. Another variation are Token based systems [87, 89] which try to separate performance from correctness and get the benefits of both techniques. This idea tries to optimize the protocol for common cases and allow an unordered interconnect but rely on a correctness substrate to resolve races.

Finally, software coherence is the easiest one to implement by hardware because it basically consists in not providing coherence. Some processors [52] implement this technique because coherence protocols are very expensive in terms of hardware and explicitly managing cache operations can improve performance. The bad part of this type of systems is that programming complexity is much higher and, therefore, not suited for all kind of applications, especially if performance is not critical.

The selection of one memory organization and protocol or another highly depends on the physical constraints, desired performance and energy-efficiency and the type of applications that are going to be executed in the multiprocessor. In general a memory organization can be classified by the following parameters:

- **Resource Placement:** Memories/caches are centralized or distributed and highly banked or not.
- **Coherence Enforcement:** Directory-based, snoop-based or software-based.
- **Migration & Replication:** Ability to migrate or replicate data close to the requesting nodes in order to reduce the access latency and network traffic.
- **Performance Isolation:** Ability of the system to execute independent applications without interfering one with each other.
- **Adaptivity:** Ability of the system to adapt the available memory resources to optimize a certain performance metric depending on the different application requirements.

Common Features	**Multiprocessors Memory (MP)**	**Chip Multiprocessors Cache (CMP)**
Shared Centralized. High capacity, uniformly long latency	UMA [19]	Shared cache (UCA)
Shared Distributed. High capacity, non-uniform latencies	CC-NUMA [2, 68, 73, 74]	Shared banked cache (S-NUCA) [60]
Private Distributed. Lowered capacity, low latency	COMA [36, 110]	Private caches
Partition between private/shared space	RC-NUMA [140]	Adaptive private/shared NUCA [30, 55]
Victim caching	VC-NUMA [93]	Victim replication [138]
Hints for relocation	R-NUMA [32]	R-NUCA [38] CMP-NuRapid [22]
Adaptivity through biased replacement	AS-COMA [67]	CC [17] MLP-Repl [103]
Adaptivity through selective replication	OS support [129] MigRep [113]	ASR [9]
Performance isolation enforcement	Performance Isolation [130]	Fair Caching [62] CQoS [54]

Table 2.1: MP-CMP memory organization similarities

The steady increase in the die real state has made possible the creation of multiprocessors on-chip, also known as chip multiprocessors (CMPs). CMPs started to appear as research projects like the TRIPS [109] or the Piranha [33] in 2000 and in 2001 IBM presented the first non-embedded commercial CMP with two cores, the Power4 [125]. From then the number of cores has been increasing, especially for high-end server market processors with the apparition of chips like the Niagara [65] with 8 cores and 32 threads in 2005 or the Niagara 3 with 16 cores and 128 threads in 2010.

Extensive work has been done to design a suitable memory hierarchy for these microarchitectures and many ideas from traditional multiprocessors have been adapted and reused. Table 2.1 shows the similarities of proposed memory organizations for traditional multiprocessors and chip multiprocessors.

Most of the existing processors use cache coherence protocols that enforce a relaxed consistency. This is that protocols allow reads and writes to complete out of order and any synchronization operation in a multithreaded application is left to the programmer. On the other hand, extensive work has appeared to create a transactional memory [41, 83, 84, 136] to facilitate parallel programming and provide atomicity, consistency and isolation to some code regions called transactions. In this thesis we use cache coherence protocols that enforce a relaxed consistency because of its simplicity and wide adoption in commercial processors. However, more strict consistency models could be added to the presented techniques.

2.3 Cache organizations in the multicore era

In chip multiprocessors, latencies between the different levels of the memory hierarchy have changed significantly compared to traditional multiprocessors, forcing a redistribution and reassignment of resources. In these architectures off-chip bandwidth is much more limited because it must be shared among all the on-chip cores, forcing an optimization of the on-chip cache allocation to minimize off-chip misses. In addition, access latencies between caches inside the chip are much lower than in traditional multiprocessors, allowing cheap communication between them and cache-to-cache transfers. These differences have motivated the apparition of multiple new organizations in the last decade.

Table 2.2 shows a classification of the most relevant cache configurations that have appeared. The different techniques are classified according to the type of cache being used and the coherence protocol. This classification has been done based on the last-level cache (LLC) configuration, in most cases centralized cache structures also have private L1 caches per core. Also, centralized and distributed snoop/token structures usually implement a bit vector of L1 sharers in the cache entries which can be considered a directory. However the search mechanism in the LLC is very different from the directory configuration and, therefore, are classified in different categories.

Another important characteristic of cache organizations is the ability to reassign cache resources in order to optimize cache allocation and reduce off-chip misses. Several studies have evaluated the optimal Partitioning of Cache Memory [48, 76, 118] and, if done correctly, it can have a great impact in the overall performance. We have divided the most recent work in two categories, static and dynamic. In static resource partition mechanisms all threads have the same priority and the amount of cache assigned to each thread is changed through the coherence protocol and replacement mechanisms. Dynamic resource

Structure	Coherence	Memory Organization Type		Migration	Replication	Isolation	Repartition	SW Support
Centralized Cache	Unified Req Queue or Cache Tags	Adaptive Caches [121]						
		MLP-Aware Cache Replacement [103]						
		Dual data Cache [34]						
		V-way Cache [105]					X	
		Heterogeneous Way-Size Cache [1]					X	
		Indirect Index Cache [37]					X	X
		OS-Managed Cache [106]				X	X	X
		Fair Caching [62]				X	X	
		CQoS [54]				X	X	
		Utility-Based Cache Partitioning [104]				X	X	
		Adaptive Set Pinning [115]				X	X	
		Dynamic Partitioning [122]				X	X	
		Adaptive Shared/Private NUCA [30]				X	X	
Distributed Cache	Snoop	Bus	Adaptive L2 snarfing [114]	X	X		X	
			NuRapid [22]	X				
			CMP-NuRapid [22]	X	X			
		Fixed mapping	S-NUCA [50, 60]					
			D-NUCA [50, 60]	X				
			M-NUCA [59]	X				
			CMP-Hybrid [10]	X				
			R-NUCA [38]		X			X
		Token	The Auction [80]		X			
			Adaptive Selective Replication [9]		X			
			ESP-NUCA [55]	X	X	X	X	
	Directory	Adaptive Migratory Sharing [117]		X	X			
		Adaptive Protocol [25]		X	X			
		Molecular Caches [128]			X	X	X	
		PageNUCA [20]		X				
		Victim Replication [138]			X			
		Cooperative Caching [17]		X	X			
		Cooperative Cache Partitioning [18]		X	X	X	X	

Table 2.2: Taxonomy of CMP Cache Organizations

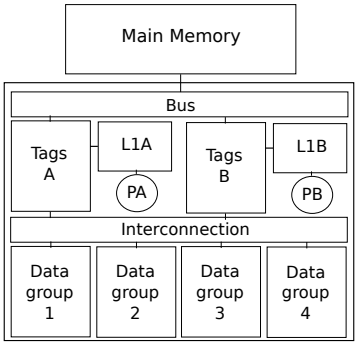


Figure 2.2: CMP-NuRapid Memory Structure.

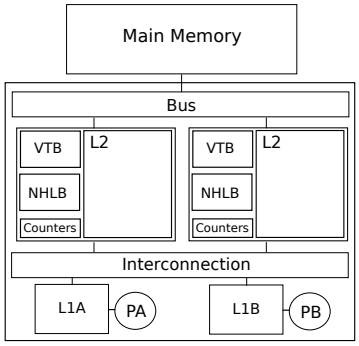


Figure 2.3: Adaptive Selective Replication Memory Structure.

partitioning mechanisms, on the other hand, take an active role and enforce the creation of cache partitions which are adapted to the different application requirements.

2.3.1 Static partitioning

Most of the traditional cache organizations are classified in this category since the allocation of caches to threads is fixed. This does not mean that these organizations are not able to use dynamic policies to optimize the memory performance. Most techniques use dynamic migration or replication to reduce access latencies or the number of off-chip misses.

CMP-NuRapid [22] from Chisti et al. is one of the static partitioning schemes. This work proposes a duplication of tags in each node like in Figure 2.2. In this scheme, tags are copied to the local node tag set the first time the block is accessed, and the data replicated in a closer cache if the block is accessed again. This way, all subsequent accesses will have a smaller latency. It has the advantage that several blocks of the same set can be in the closer cache if they are used often with no risk of being replaced since tags and data are separated. This proposal has the power and performance limitation of requiring most of the times a transfer of the least used block to a slower group when we want to add a new one to the fast and is not scalable because blocks are found via snoop requests.

Beckmann et al. proposed the Adaptive Selective Replication mechanism [9] which adapts the level of replication dynamically. This system uses a distributed shared memory with every address mapped to only one L2. As it can be seen in Figure 2.3 the system also has extra hardware to measure if the level of replication is adequate. This hardware is a group of Next Level Hit Buffers (NHLB), a group of Victim Tag Buffers (VTB) and some counters. With this hardware a percentage of the blocks evicted from the L1 is replicated in the local L2. This percentage is known as the level of replication and is adjusted dynamically.

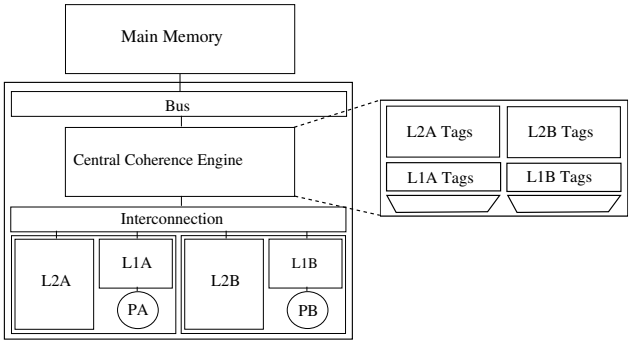


Figure 2.4: CC Memory Structure.

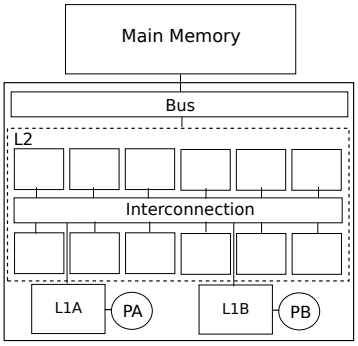


Figure 2.5: NUCA Memory Structure.

Cooperative Caching [17], on the other hand, uses a directory based protocol and private caches. This technique proposes to replicate all cache tags in a centralized directory, as shown in Figure 2.4, to allow sharing between distributed private caches. In addition, it proposes a mechanism to forward blocks with only one on-chip copy to other caches when evicted to reduce the number of off-chip misses and share the available cache space.

Finally Kim et al. [50, 60] propose a shared pool of small cache banks that can have different degrees of sharing. Dynamic mapping (D-NUCA) allows data to be stored in multiple banks but requires a tag check of all the possible destinations. Results show that statically mapping (S-NUCA) banks has similar performance and much less complexity. Different additional techniques [78, 79, 80] have been proposed to optimize replacement policies in this type of organization.

Organizations with static partitioning either have inter-thread interferences (e.g. Cache-intensive threads may degrade performance of other applications by replacing their blocks) or are not able to give all the cache space to a single thread if the others are not using the cache. This is logical if we consider that the cache space is either completely shared or statically mapped to threads, and can lead to a non-optimal usage of resources in unbalanced workloads.

2.3.2 Dynamic partitioning

Dynamic resource partition mechanisms, on the other hand, dynamically modify the amount of memory that is assigned to every node. In addition to the benefits of previous proposals, they eliminate inter-thread cache conflicts and allow the implementation of fair systems or even to provide a Quality-of-Service. Thread interference is mitigated by allocating inde-

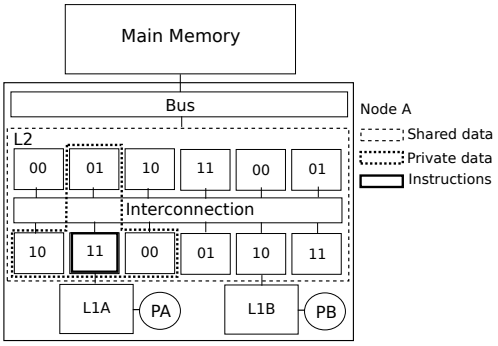


Figure 2.6: R-NUCA Memory Structure.

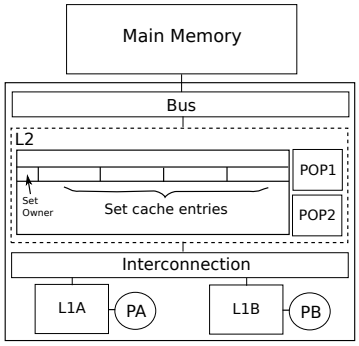


Figure 2.7: Adaptive Set Pinning Memory Structure.

pendent partitions of resources. These resources can be divided in banks, sets or ways and require an arbitration mechanism that can be software or hardware based.

Software-based dynamic configurations delegate resource allocation to the OS. Most of these organizations divide resources in independent sections to be able to provide QoS [18, 38, 54, 100]. Virtual Private Caches [100] divide resources in ways and implement a hardware arbiter to dynamically distribute the unallocated space in accordance to a fairness policy. In the Cooperative Cache Partitioning [18] resources are not only partitioned spatially, but also in time. They apply Multiple Time-sharing Partitions to expand the cache capacity of some threads at a given time and increase throughput. Iyer [54] studies different mechanisms to provide QoS in the memory hierarchy. However, it is focused on the distribution of a unified shared last-level cache. Liu et al. present the Shared Processor-Based Split L2 [81], a cache configuration that also allows software-based distribution of cache resources. The purpose of this configuration is not to provide QoS but to be able to select private or shared caches depending on the application. This organization is limited by a snoop based protocol to access cache data that requires broadcast messages to all cache banks.

Finally, R-NUCA proposes a variable block mapping depending on the kind of data; allocating private data close to the requesting node and replicating shared read-only blocks. Figure 2.6 shows the structure and the data locations for Node A. Classification is done at a page-level granularity at the time of the TLB miss by the OS.

Hardware-based dynamic organizations [30, 50, 104, 115], on the other hand, are able to implement the repartitioning policy in hardware, reducing the programming complexity. They are based on performance counters to measure the benefit of increasing the cache size for each thread.

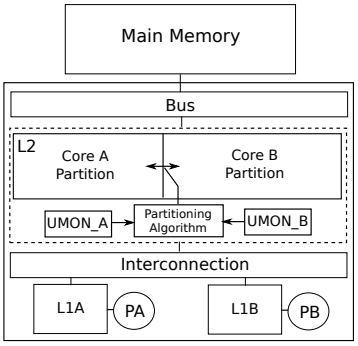


Figure 2.8: Utility-Based Cache Partitioning Memory Structure.

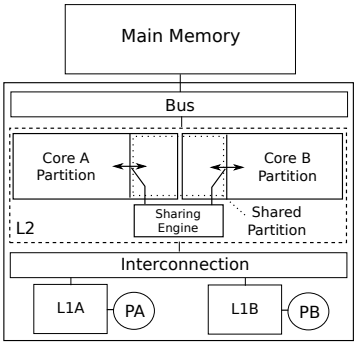


Figure 2.9: Adaptive Shared-Private NUCA Memory Structure.

Srikantaiah et al. [115] presented the Adaptive Set Pinning, a technique to reduce interprocessor misses by assigning a replacement ownership to every set. This ownership is varied dynamically to optimize the cache usage. In addition, it uses extra small private caches (POPs) to allocate blocks of threads that do not have the ownership of the corresponding set. Figure 2.7 shows the proposed structure.

The Dynamic Spill-Receive (DSR) [102], on the other hand, uses private L2 caches to allocate evicted blocks from other caches if a performance improvement is expected. The ability to reallocate or accept evicted blocks is decided through set dueling using miss information of all caches. Therefore, for a given cache, miss information of all caches must be provided for the tested sets to decide its behavior. This makes this technique interesting for a small number of nodes but unfeasible for a large number of them due to the communication overhead that this would entail.

The Utility-Based Cache Partitioning [104] uses a big unified 16-way cache. In this cache, ways are assigned to nodes according to the benefits that can produce to each thread. Figure 2.8 shows the structure of this technique, which uses a centralized cache partitioned with the column caching technique [21]. In addition, the system uses Utility Monitors (UMON) which track the utility of each way for each core and decide the optimal partition of the cache. Dybdahl et al. [30] proposed a similar technique, depicted in Figure 2.9, but with a different selection criteria for the sharing mechanism. Both proposals do not try to reduce latency by allocating blocks in the closer nodes and are not scalable due to the centralized nature of the last level cache.

Another hybrid proposal is the Victim Replication [138] protocol. This configuration has a traditional distributed shared memory but adds a new replacement mechanism to reduce the miss latency. By default, blocks have a fixed L2 cache for being stored but in L1 replacements the block is replicated in the local cache if there is a spare place. The main limitation

of this configuration is that under heavy load conditions behaves as a normal shared cache configuration. Finally, Qureshi [102] proposed a dynamic technique that spills blocks to nodes in order to reduce the overall cache misses. This technique, however is based on a snoop protocol and is only useful for a small number of processors.

2.4 Memory Controllers

During the last decade memories have greatly evolved in terms of capacity and integration but still remain one of the main limiting factors of current processor performance. The steady increase in processor performance in the last decades has not been followed at the same pace by memory systems and, therefore, has created what is known as the memory gap.

This problem has been exacerbated with the introduction of chip multiprocessors, which require much larger amounts of data and have different access patterns. The simultaneous execution of multiple applications in a single core also introduces inter-thread interference and a competition to use shared resources like the memory bus. Such changes suggest that the memory hierarchy must be adapted to deal with the new requirements.

In the overall memory performance, memory schedulers have a great influence due to its capacity to prioritize different type of requests according to a given goal. Traditionally for uniprocessors throughput has been the main concern when designing these schedulers. Due to the large size of memory arrays, memories use row buffers which store a whole page to allow faster reads and writes. This buffer needs to be updated every time a different row is read or written, consuming time and energy. Therefore, it is critical that memory systems make as much use as possible of row locality to both increase performance and reduce energy consumption. The First-Ready First-Come-First-Serve (FR-FCFS) policy [108, 133, 142] is the most used organization for uniprocessors due to its simplicity and high row hit rate.

In addition, the usage of a shared resource like DRAM memory by different threads makes it necessary for the system to provide some kind of fairness or performance isolation control. Several solutions [97, 98, 99] have appeared that enforce that all threads receive a similar amount of service.

Other solutions have seen that prioritizing certain threads or regions of data can be beneficial to the overall system throughput [63, 64, 141]. Therefore, they have presented techniques not centered in memory throughput but in providing critical data in a reduced latency and in some cases combined with a fairness control.

In these microarchitectures, prefetchers continue to play an important role by increasing the memory level parallelism. The usage of prefetchers, however, can also degrade memory performance and penalize demand requests. Therefore, several organizations have been proposed [31, 70, 71, 77] in order to take into account the different request priorities and optimize the interaction of prefetchers and memory controllers.

And finally, energy efficiency has also risen as an important concern in the memory system which already can account for 30% of total system power [8]. Therefore, several new organizations have appeared [69, 137] to tackle this problem through a better usage of the DRAM power modes or through a simplification of the memory structures.

In this thesis, we present a survey of the most recent schedulers proposed for chip multiprocessors and its taxonomy. We show that many different approaches exist in the design of the memory controller and that depending on the optimization goal different solutions may be desirable.

2.4.1 Memory Organization

Memory performance is highly tied to its structure and any optimization must always consider the trade-offs brought by it. Therefore, it is important to know how a DRAM memory is implemented. In this section we present a brief explanation of the main characteristics of DRAM memories, for a more detailed explanation we refer to the work of Wang et al. [131].

Figure 2.10 shows the organization of a typical DRAM memory. These memories have several chips (typically 8 + 1 to provide ECC), each responsible for providing a part of the block simultaneously. Inside the chip, memory is organized in banks, each holding a part of the address space. Since memory parts are very big, and to reduce access latency, data is accessed inside the memory in rows (also called pages). In order to access data every bank has a row buffer and every time that an address is accessed the corresponding row is loaded to the row buffer. Subsequent accesses to addresses in the same row require a much smaller access time since the row is already in the row buffer. This organization generates different situations that can arise when accessing memory:

Row Hit: In this case the row is already in the row buffer so we can read it directly. Access latency will be the Column Access Strobe latency (T_{CL}), the time between column access and data return by the DRAM.

Row Closed: There is no data in the row buffer. Access latency is the one required to load (*activate command*) the row and then read. Access latency will be the Row to Column

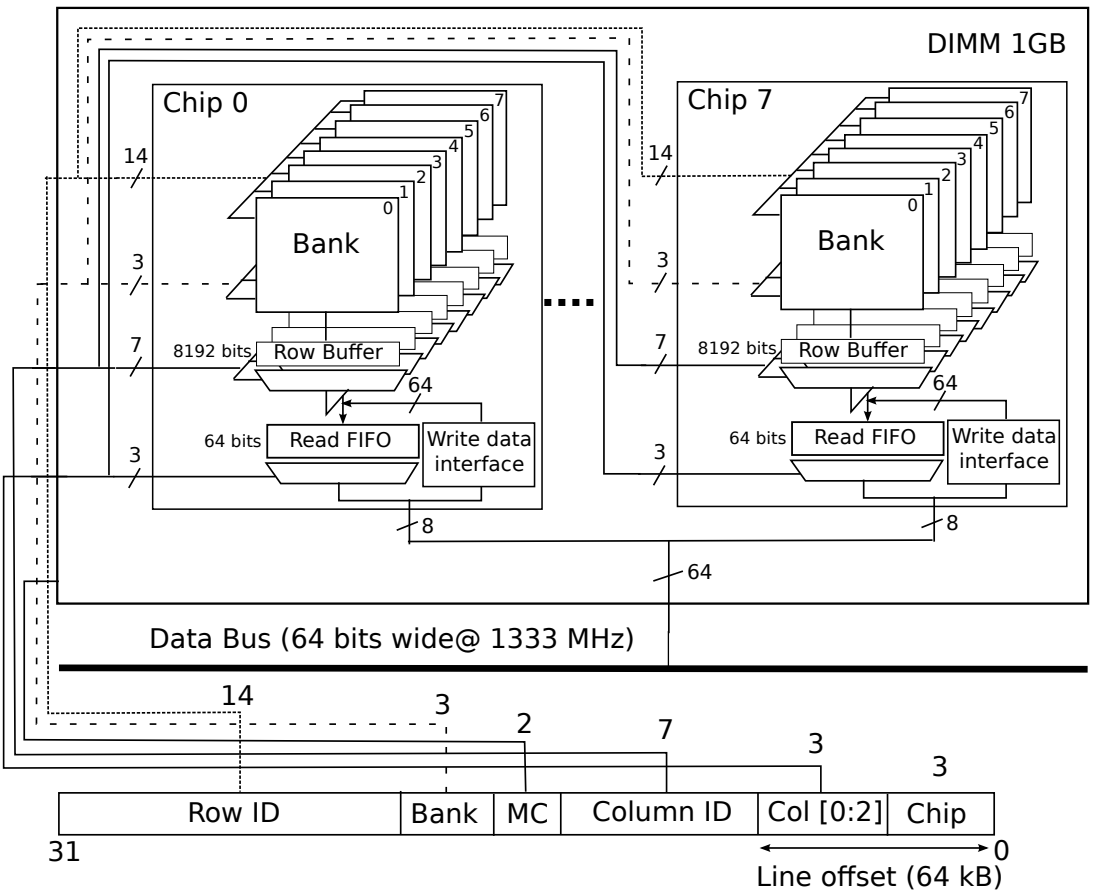


Figure 2.10: Memory Organization and Mapping.

command Delay (T_{RCD}), the delay between the row access command and the data ready at the row buffer, plus the read latency; T_{RCD} + T_{CL}.

Row Conflict: In this case, we have a different row in the row buffer and, therefore, we need to writeback this row (*precharge command*) and load the one we want to access before reading. Access latency will be the Row Precharge time (T_{RP}) plus the activate and read latency; T_{RP} + T_{RCD} + T_{CL}.

Since no data from one bank can be transfered during its row activation or row precharge, multiple banks are used. Therefore, when a row is activated in one bank a block can be read in another one and there is always data available to be transferred.

Address mapping to physical memory also has great influence in the overall performance. In a general configuration suited for all kind of applications it would be desirable to distribute memory accesses among all memory banks and also maximize the hit rate. Figure 2.10 shows the address mapping used in this thesis. In this mapping Column ID is mapped to the least significant bits to keep consecutive addresses in the same row and

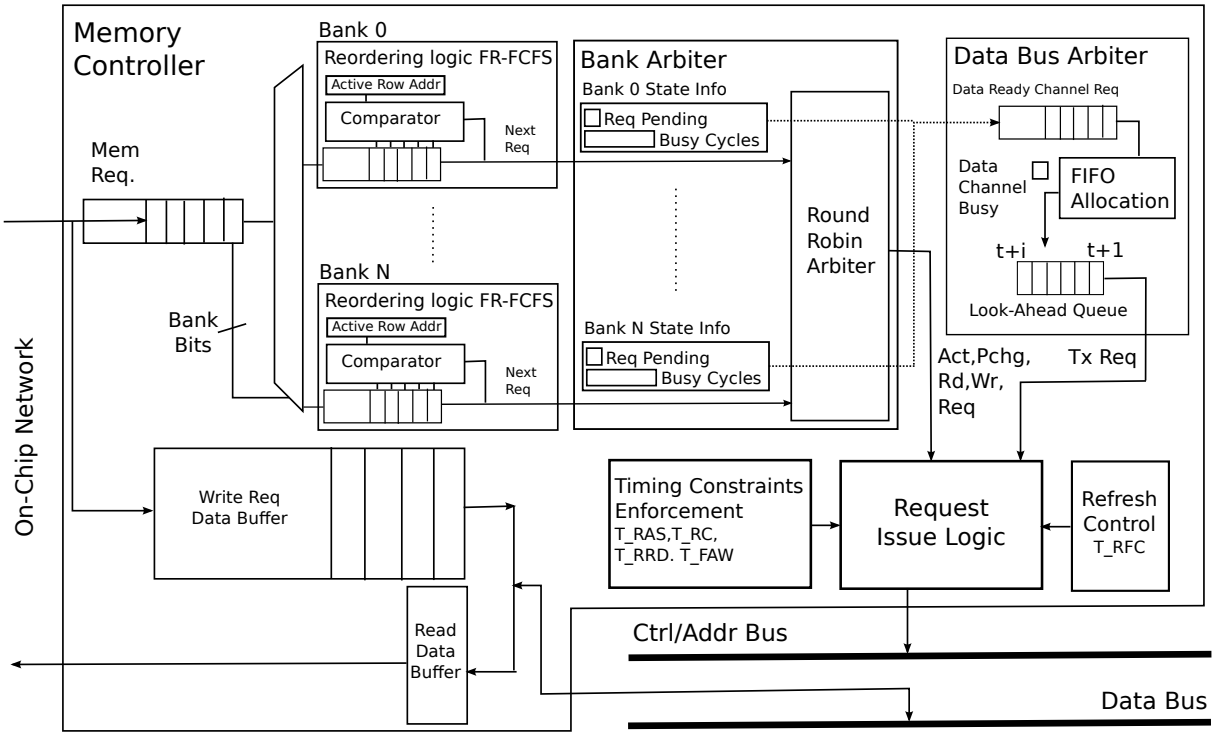


Figure 2.11: Memory Controller Structure.

maximize the hit rate. The next bits after the row are mapped to DIMMs and banks in order to spread requests among memory controllers. Finally, more significant bits are devoted to Row ID to reduce the row miss rate. This mapping is intended for open-page configurations (row is not precharged after being accessed). For closed-page configurations (row is always precharged after being accessed) adjacent lines are usually mapped to different banks to take advantage of the available bandwidth and because spatial locality is expected to be small between consecutive accesses.

2.4.2 Memory Controller structure

Tightly connected to the memories are the memory controllers. Memory controllers arbitrate between requests to different banks, arbitrate the data bus usage and enforce the memory timing constraints.

Figure 2.11 shows the structure of a memory controller. Requests are separated in different queues depending on the bank where the address is mapped. In these queues accesses are reordered following the desired scheduler policy. Several types of schedulers exist depending on the desired behavior of the memory system and which are going to be analyzed and classified in the following section of the chapter.

The second part of the memory controller is the bank arbiter. This part usually implements a simple round-robin arbitration policy. However, only idle banks can issue requests; therefore, it is necessary to keep track of bank states to know if they are performing a row activation or precharge or if a previous request is waiting to use the data bus.

Once the desired row is activated, requests are enqueued in the data bus queue. The data bus arbiter is a critical part of the memory controller since the limited bandwidth usually is the bottleneck in current processors. Due to the delay after read requests and data availability a look-ahead queue is required to reserve the bus at the time the data is in the Read FIFO. This arbiter also must take into account the 1 cycle delay incurred when requests change from read to write or vice-versa.

Another important function of the memory controller is the enforcement of timing constraints. Due to energy and thermal limitations memories are constrained by some parameters like T_{RAS}, T_{RC}, T_{RRD} and T_{FAW}. It is also necessary to refresh the data to ensure no information is lost and this must be done every T_{RFC} cycles for a given region of memory. Therefore, one important part of the memory controller must include cycle and event counters that enforce these constraints regardless of the number of requests.

Finally, the request issue logic must prioritize the request that is going to use the ctrl/addr bus, priority is given first to memory constraints, then the data bus arbiter, and finally the bank arbiter.

2.5 DRAM Bank Schedulers for multicore processors

In the previous section we have seen the main parts composing a typical memory controller. Most of the scheduling done in the memory controller is fixed and based on physical constraints (like the power related timing parameters) or must be First-Come-First-Serve like the bus scheduling due to complexity and starvation issues. Bank scheduling however is much more flexible since it allows to implement different reordering policies depending on the desired behavior of the memory controller and this has allowed the apparition of multiple schedulers oriented to different goals.

We have classified existing memory bank schedulers in two categories; memory oriented and system oriented. Memory oriented schedulers only consider memory metrics while system oriented take into consideration the whole system. Then we have divided these categories depending on the optimization goal. Table 2.3 shows this classification.

It is important to note that the schedulers can be optimized for two different types of throughput: memory throughput and system throughput. Memory throughput is the number

Optimization		Configurations
Memory Oriented	Memory Throughput	First-Ready First-Come-First-Serve (FR-FCFS) [108, 142]
		Adaptive History-Based memory scheduler [51]
		Intel's 870 system controller [14]
		Self-Optimizing memory controller [53]
		Virtual Write Queue [120]
	Fairness	Fair Queuing [99]
		Stall-Time Fair scheduling (STFM) [97]
		Parallelism-Aware Batch Scheduling (PAR-BS) [98]
System Oriented	System Throughput	Fine-grain Priority scheduling [141]
		ATLAS [63]
		Thread Cluster Memory (TCM) [64]
	Prefetch -Aware	Hierarchical Prefetcher Aggressiveness Control (HPAC) [31]
		Low priority prefetch [77]
		Prefetch-Aware DRAM Controller (PADC) [70]
		BLP-Preserving Multicore Request Issue [71]
	Power/ Area-Aware	Power aware page allocation [69]
		Complexity Effective scheduling [137]

Table 2.3: Schedulers Taxonomy

of memory requests that can be serviced in a given amount of time while system throughput is the number of instructions executed in that time. While it may seem that the higher memory throughput can provide the highest system throughput, it is not always the case as we are going to see. Several schedulers [63, 64] try to improve system throughput by prioritizing the least-attained service, assuming that thread service requirements follow a Pareto distribution.

2.5.1 Memory Throughput Oriented Schedulers

There has been extensive work in optimizing memory controllers for chip multiprocessors. This work has been greatly centered in the optimization of bank and channel arbitration. Request arbitration in banks first was mainly focused on memory throughput.

The First-Ready First-Come-First-Serve (FR-FCFS) policy [108, 142] is an optimal solution from the memory throughput point of view and it is widely used. This technique prioritizes accesses to active rows, minimizing the amount of row activations. Some mechanism

to avoid starvation must be included in order to ensure that under heavy load conditions all requests are serviced. Since only memory throughput is considered this technique can present fairness and isolation problems.

Many techniques have appeared lately that take into consideration the interaction between threads. One of them is the Adaptive History-Based memory scheduler [51] which reorders requests making use of the command history and the set of available commands to minimize the request delay. This technique extends the FR-FCFS policy to also consider change of rank and port delays. In addition, it proposes a mechanism to enforce that the number of reads and writes matches the application behavior in order to avoid filling the reorder queues. This technique, however, does not consider the delay of switching from a read to a write operation in the memory channel which can add a significant overhead if not taken into account.

Several techniques have considered the R/W switch delay and the fact that write requests are typically not latency critical which allows grouping and stalling requests. This write request stalling is known as write caching and is implemented in commercial controllers like Intel's 870 [14]. Write grouping allows to reduce the number of bus turnaround delays and use it more efficiently. Other techniques like Eager writeback [72], although not being memory schedulers, also consider these trade-offs and perform writebacks before the block is evicted. This technique does not necessarily group requests but allows to perform writebacks when the bus is less congested.

The Self-Optimizing memory controller [53] uses reinforcement learning techniques to estimate the long term performance impact of each action and reorder requests to maximize long-term performance. The goal of this system is also to maximize the memory throughput and this is done through the calculation of a value (Q-value) for each request which is used to select the next request to issue. The Q-value is calculated based on the number of read and write requests in the queue, the number of reads in the queue due to load misses and their arrival order, and if the reads and writes produce a row hit. The hardware solution proposed is able to compare only 12 requests from the queue which they consider is enough to achieve a good performance.

Other work increases performance by a memory-aware management of the last-level on-chip cache to simplify the memory scheduler. The Virtual Write Queue [120] packs the cache writeback operations to enable longer write bursts and amortize bus turnaround times with read operations.

2.5.2 Fairness Oriented Schedulers

With the advent of chip multiprocessors, however, new problems have arisen. Performance isolation and fairness have become as important as throughput and several publications have appeared centered on solving these problems.

Some techniques are based on Network Fair Queuing algorithms to grant fairness. In such environments each user has a service share (proportion of the resource time that is assigned) and the scheduler enforces it. In the case of not having pending requests of the user that has to be serviced other users are served. The usage of this idle time is not considered in the following service partitions, therefore, not penalizing the users taking advantage of it. Fair Queuing [99] is one of these schedulers and enforces the service share of each thread through a Virtual Time Memory System. This system calculates the virtual finish time and prioritizes requests with smaller values. An interesting part of this technique is that, to avoid hurting memory throughput with the fair scheduler, every time that a row is activated it is kept active for x cycles if there are more requests to that row. This time is set to T_{RAS} in order to enforce a more fair scheduling. Rafique et al. [107] proposed an improvement to Fair Queuing by treating each memory request as a unit of scheduling. This assumption allows the usage of start time fair queuing which improves worst case latencies.

Stall-Time Fair scheduling (STFM) [97], on the other hand, provides Quality of Service by reordering requests to equalize the memory-related slowdown (S) between threads. This technique computes every DRAM cycle the slowdown of every thread in stall time. If the ratio between the maximum and the minimum slowdown exceeds a certain threshold, requests of the thread with highest slowdown are going to be prioritized. Otherwise, a traditional FR-FCFS scheduler to optimize the memory throughput. A set of registers is required for each thread to calculate the memory-related slowdown. Since it is difficult to estimate the slowdown of the alone execution, it is computed as the shared slowdown minus the interference.

$$S_t = \frac{T_{shared}}{T_{alone}} = \frac{T_{shared}}{T_{shared} - T_{interference}}$$

To estimate the interference every thread has a $T_{interference}$ register initialized to zero and increased every time that a thread request has to be stalled due to other threads.

And finally, Parallelism-Aware Batch Scheduling (PAR-BS) [98] is a technique that enforces fairness and performance by processing requests from a thread in parallel in the DRAM banks to reduce the memory-related stall-time experienced by the thread. One of the main problems when enforcing fairness in the memory scheduler is starvation avoidance. PAR-BS creates request batches to ensure that all requests within a batch are

serviced before creating a new one. This allows more aggressive reordering techniques without starvation issues and provide some performance isolation. However, batching requests can also hurt performance since it is prioritized over achieving a higher row hit rate. The reordering policy implemented in PAR-BS is based on the creation of a thread rank. The thread ranking is computed giving higher priority to the thread with lower number of requests in the batch and in case of tie with lower number of marked requests in all the banks. This policy tries to increase the intra-thread bank parallelism within a batch

The main problem in all the existing reordering techniques is that the important influence of row buffer locality in the memory system throughput generates a trade-off between throughput and fairness or performance isolation.

2.5.3 System Throughput Oriented Schedulers

Some techniques, on the other hand, are more focused in system throughput (executed instructions) rather than memory throughput (serviced requests). Several techniques have appeared in this field, either prioritizing critical data or shortest-queue tasks.

Fine-grain Priority scheduling [141] splits memory references into sub-blocks with minimal granularity and maps them to different channels. This technique is based under the assumption that only a portion of cache lines contain the required data but the other portions are likely to be needed in the near future. Therefore, sub-blocks that contain the desired data are marked as critical and the rest are treated normally. This technique allows to effectively use the bandwidth of all channels when retrieving a block and to prioritize only the critical sections. The distribution of blocks across the different channels, however, has the limitation that all memories are going to increase the number of requests to different blocks. Therefore, the row hit rate is going to decrease and as a result memory throughput is going to be reduced.

ATLAS [63], on the other hand, reorders thread priorities based on the service they have attained previously, prioritizing the ones that have requested the least service. This technique is based on queuing theory [135] which shows that when the job size distribution is exponential and the arrival process is Poisson, then the shortest-queue task assignment policy is optimal.

This technique calculates the attained service every time quanta (a fixed number of cycles) in order to rank the different threads (LAS-rank). The following formula shows how it is calculated, taking into account both the long and short term usage with the α parameter

that they set to 0.875. To avoid starvation any request that waits more than T cycles is prioritized (TH). T is set to 100k cycles.

$$TotalAS_t = \alpha TotalAS_{t-1} + (1 - \alpha)AS_t$$

Thread Cluster Memory (TCM) [64] scheduling also is focused on system throughput and divides threads into two separate clusters; latency-sensitive and bandwidth-sensitive. This scheduler is also based on the assumption that the system throughput benefits of prioritizing memory-non-intensive threads over memory-intensive ones. Therefore, this technique prioritizes always latency-sensitive threads. Within the latency-sensitive cluster lower MPKI threads are prioritized following the same principle. Within the bandwidth-sensitive cluster, threads should fairly share memory bandwidth to ensure no single thread is disproportionately slowed down. In this type of configuration it is important that the threads placed in the latency-sensitive cluster always consume a small fraction of the total memory bandwidth to avoid starvation. To classify threads, TCM monitors its memory intensity and memory bandwidth usage and then adds the threads with lower usage to the latency-sensitive cluster until this cluster represents a bandwidth between 1/12 and 1/4.

2.5.4 Prefetch-Aware Schedulers

Also, some work has appeared that studies the interaction of different parts of the chip like caches or prefetchers and the memory controllers and proposes a coordinated control between them. Prefetchers are a very important part in many existing processors and contribute significantly to reduce miss latencies. Moreover, the usage of prefetchers has a great impact in the memory system usage and access patterns. McKee et al. [90] show that prefetching blocks in streams can reduce the row alternation and increase the row hit rate. Prefetching, however, if not done correctly, also can increase unnecessarily the pressure in the memory system degrading the overall performance. Therefore, several works have studied the interaction between prefetchers and memory controllers and have proposed techniques to optimize their interaction.

Ebrahimi et al. [31] show that prefetchers can cause a significant interference to other cores and, therefore, need to be dynamically adjusted in a coordinated way. They propose the Hierarchical Prefetcher Aggressiveness Control (HPAC) composed of local and global prefetch control structures. Local control adjusts the aggressiveness at each node to maximize the performance of that core as many prefetchers. Global control, on the other hand, monitors the inter-core interference and overrides local decisions if necessary to maximize the overall system performance and bandwidth efficiency.

Lin et al. [77] also study the influence of prefetchers and propose a memory controller with two priorities, high for demand requests and low for prefetch requests. This technique allows to schedule prefetches only during idle cycles and do not harm the overall latency of demand requests.

Always prioritizing demand requests over prefetch requests, however, can degrade performance in some cases since if these prefetches are useful they can reduce the number of demand misses. Srinath et al. [116] realized of this fact and proposed a prefetcher to dynamically adapt the aggressiveness of prefetchers but without considering the memory controller. Later, this idea was exported to the memory controller with the Prefetch-Aware DRAM Controller (PADC) [70]. This technique proposes the usage of three components, an Adaptive Prefetch Scheduling (APS), an Adaptive Prefetch Dropping (APD) unit and a prefetch accuracy monitoring system. Every core measures over a certain time interval the accuracy of its prefetches and then this information is used by the APS and APD in the memory controller. The APS is responsible of setting the priority of demand/prefetch requests based on the prefetch accuracy estimated for each core. This prioritization gives higher priority to useful prefetches and demand requests and issues the rest of the prefetches only during idle cycles.

Lee et al. [71], on the other hand, propose a Bank-Level Parallelism aware prefetcher (BAPI) and scheduler (BPMRI). Since banks can operate concurrently, the best way of granting a continuous amount of data to the bus is to maximize the bank parallelism of requests. The proposed prefetcher prioritizes requests to different banks over requests to the same bank While prioritizing requests to different banks it is possible to increase the memory-level parallelism, this prioritization also to reduces the row locality that determines the overall row hit rate. In uniprogrammed configurations it may not affect significantly because no other requests arrive to the banks and the used rows remain active. In multiprogrammed environments, however, this configuration encourages row alternation and, therefore, a reduction of the row hit rate which greatly impacts in the overall performance and energy consumption of DRAMs.

2.5.5 Power/Area-Aware Schedulers

In addition to performance and fairness, memory controllers and DRAM memories also are limited by energy and area constraints. DRAM power in modern server systems can account for 30% of total system power [8]. Reordering mechanisms in the memory controller add a significant complexity to the design which increases the required area and

energy of the system. Therefore, several techniques have appeared to improve these limitations. Extensive work has been done in reducing the energy consumption in memories but mostly focused in modifying the DRAM or the data allocation and not specifically in the bank scheduler.

Lebeck et al. [69] propose a hardware/software approach to reduce the memory power consumption. They propose a system with independent chip management, allowing the memory controller to select different power modes for each chip. DRAM power modes are active, standby, nap and powerdown and each of them has different latencies and energy consumptions. In detail, they propose a power aware page allocation which clusters application's pages to the minimum number of DRAM chips, therefore, allowing the other chips to be switched to low power modes. The proposed dynamic mechanism measures the time between accesses to a chip for transitioning to lower power states. If a chip is not accessed for a threshold amount of time it transitions to the next lower power state. This mechanism can achieve significant savings in power consumption but is tied to an independent chip management. Conventional DRAMs, however, generally require multiple chips to achieve high bandwidth. Therefore, the application of this technique would sacrifice this bandwidth in order to reduce energy consumption.

And finally, Yuan et al. [137] have focused on reducing the complexity of DRAM schedulers. This study is focused in GPGPUs and the fact that out-of-order memory schedulers incur in a significant area overhead. This technique advocates for a simplification of the memory scheduler by using a simple in-order scheduler and use a memory-aware interconnect. GPUs have memory access patterns with high locality but requests are interleaved in the interconnect, leading to a reduction in the row hit rate. This technique uses memory-aware routers and banked controllers to keep the row locality and avoid complex schedulers.

2.5.6 Throughput-Fairness Trade-off

Memory schedulers are limited by the trade-off of providing the maximum throughput or any kind of fairness or performance isolation. Reordering in the bank queues is done selecting the request with highest priority. Priorities are calculated concatenating different parameters depending on the type of scheduler. Figure 2.12 shows an example of how the request priority is calculated for the FR-FCFS scheduler, the PAR-BS [98], the ATLAS [63] and the TCM [64] schedulers.

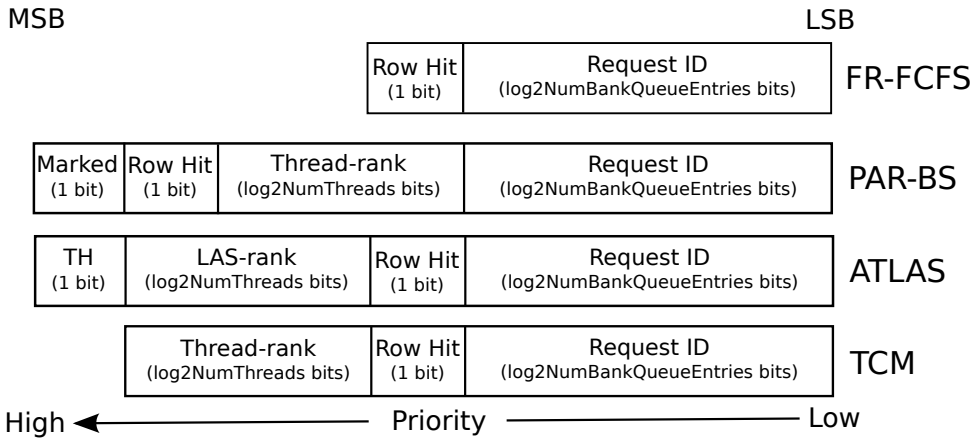


Figure 2.12: Priority calculation for FR-FCFS, PAR-BS and TRB-SP.

Scheduling Example

Figure 2.13 shows an example of how the different scheduling priorities work. This example does not capture all the details considered in the presented techniques but allows to see the tradeoffs between throughput and fairness prioritization. We assume two threads (A and B), each of them always accessing the same row and that threads are stalled until the first two requests are serviced. For each configuration we can see on the top the requests stored in the bank queue and on the bottom the requests issued to the DRAM. We also assume for the TCM configuration that thread B is classified in the latency-sensitive cluster. In the ATLAS case we would have the same behavior since it would be a thread with low attained service and, therefore, with high rank.

In the first example we can see the memory scheduling of a First-come-first-serve (FCFS) scheduler where no reordering is produced. This technique is very simple and easy to implement but is inefficient since it does not take advantage of the active rows. Therefore, it is possible to see that it is the technique with worse memory throughput since it requires the maximum amount of time to complete the requests. In addition, the unstalling time of both processors is very high.

On the other hand, FR-FCFS is the technique that achieves the highest memory throughput. This is because it prioritizes requests if there is a row hit, saving time in precharges and activations. This technique however is not fair and in some cases can stall a request from a different thread for a long time if multiple requests to the active row arrive. This is the case of processor B that has to wait until cycle number 10 to unstall.

This problem is solved in PAR-BS which prioritizes fairness. The example shows how requests are grouped in a batch at the beginning and how requests from threads with fewer

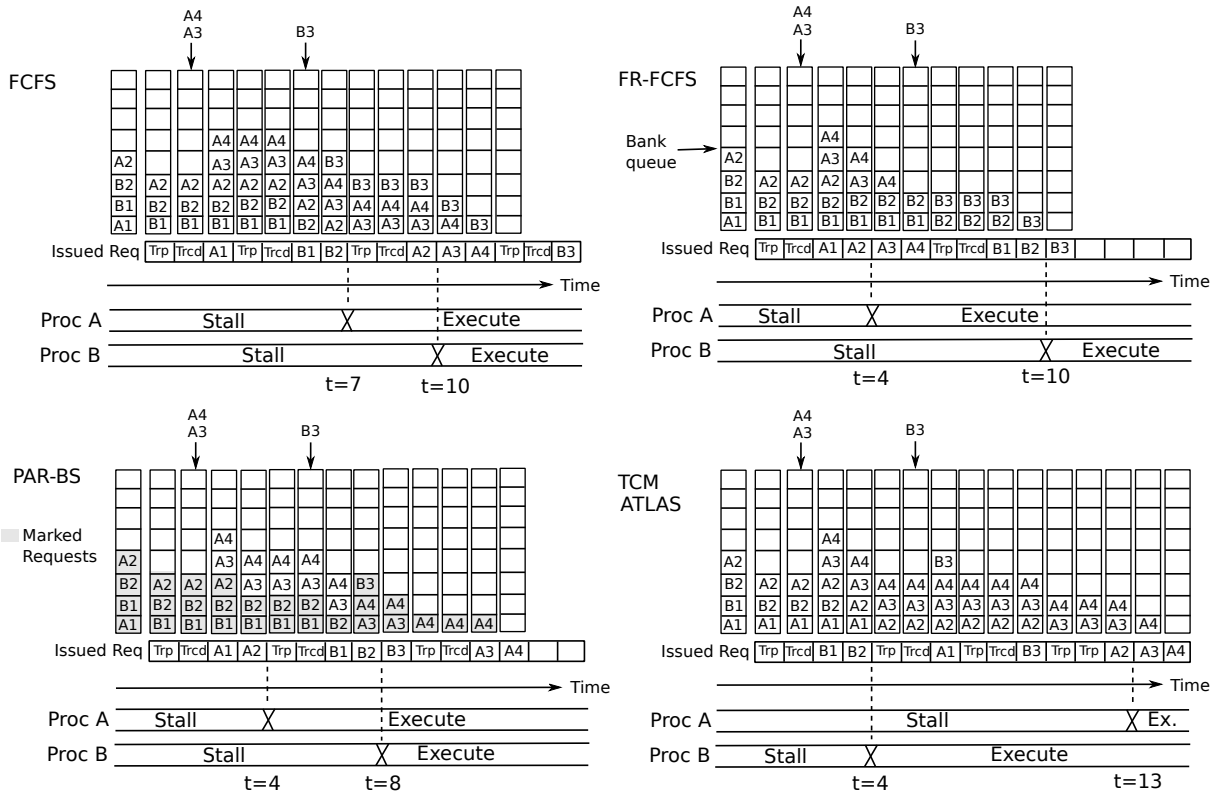


Figure 2.13: Scheduling example.

requests are given priority. Therefore, this technique allows to unstall both processors with the minimum amount of time. Batching, however, does not allow new requests to advance those within a batch even if they are to the active row. This example benefits this technique because when the second batch is created B3 is advanced thanks to be the thread with less requests and it finds the row already active. This may not be the case and it would imply extra row precharges and activations. Therefore, this technique has a smaller row hit rate and memory throughput than FR-FCFS in exchange of a more fair system.

Finally, TCM shows a memory throughput like the FCFS policy. This behavior, however, is not representative of the long term usage of this technique but allows to see the prioritization in an interesting time of the execution. Therefore, although in this example TCM seems worse than FCFS, in the long term it is going to behave better. In TCM, any request from the latency-sensitive cluster forces the activation of the corresponding row without prioritizing requests to the active row. This can lead in some cases to a reduced row hit rate and lower memory throughput as in this section of the execution. Requests from processor B, however, are in the latency-sensitive cluster and represent a small part of the overall requests. Therefore, for the most part of the execution requests are going to be prioritized

in a FR-FCFS fashion. On the other hand, this technique is able to be the first to retrieve the data for what is considered a latency sensitive cluster. This lower latency for threads with a small number of requests allows to have a higher system throughput in the long term.

2.6 Progress beyond the state-of-the-art

The advent of chip multiprocessors in the processor market over the last few years and research projects like Intel's Tera-scale [127] processor or Intel's Single Chip Cloud Computer [52] show that the number of cores per chip is going to increase in the future. These configurations must make an efficient use of the growing silicon real estate and try to take advantage of the existing parallelism of applications. Server and high-end applications are the most benefited from these platforms and it is also expected that future desktop applications for recognition, mining and synthesis [29] are going to require a high number of cores. These architectures are going to exacerbate existing challenges such as power dissipation, wire delays and off-chip memory bandwidth. In this environment, a power and performance optimized memory hierarchy is crucial. Such configuration must minimize off-chip misses by optimizing on-chip memory usage and also reduce miss latency by placing data close to the requester.

As we have seen in this chapter, there is a wide range of organizations for the memory hierarchy of chip multiprocessors. The selection of one solution is always determined by the physical limitations of the fabrication process used and the requirements of the system in terms of performance, energy-efficiency and performance isolation. We have focused on the optimization of two different parts of the memory hierarchy; cache organization and DRAM management.

2.6.1 Cache organization

Existing solutions for organizing on-chip caches have a main limitation to be used in chip multiprocessors with a high number of cores; the scalability. Most of the systems use centralized structures to manage the storage resources creating possible bottlenecks or are managed by software increasing the programming complexity. In addition, most of the proposed solutions do not take advantage of the heterogeneous nature of applications and are not able to redistribute resources depending on the application requirements. Therefore, one of the objectives of this thesis has been to design and evaluate new organizations of the memory hierarchy which optimize the cache usage while avoiding centralized structures

that may hinder the scalability of future CMPs. Such organizations must be energy-efficient and able to dynamically adapt to the execution environment.

2.6.2 DRAM management

In the DRAM management part of the memory hierarchy there are also many optimization opportunities.

As we have seen bank arbitration in the memory controller is an open topic that tries to fairly distribute memory bandwidth among threads. The trade-off between memory throughput and fairness shows a need of new organizations able to deal with the memory access patterns generated by multicore architectures. Therefore, in the second part of this thesis we have studied in detail the behavior of the memory controller under multiprogrammed environments and proposed solutions for the existing bottlenecks.

Chapter 3

Methodology

3.1 Simulation Infrastructure

All the techniques presented in this thesis have been evaluated with Simics [85], a full-system execution-driven simulator extended with the GEMS [88] toolset that provides a detailed memory hierarchy model. In addition, we have added a power model to the simulator in order to evaluate the energy consumption of Cpus, caches, interconnection network and DRAM memories.

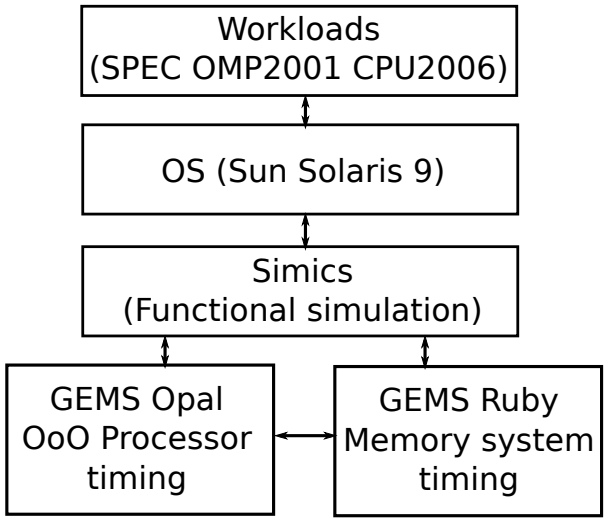


Figure 3.1: Simulation infrastructure.

Figure 3.1 shows the simulation infrastructure. Since it is a full-system simulator, all the evaluated applications have been executed on top of Sun Solaris 9. The GEMS simulator is divided in two main parts; Ruby and Opal. Ruby provides a timing for all memory requests

Parameter	Value
Number Processors	16
Instr Window/ROB	16/48 entries
Branch Predictor	YAGS
Block size	64 bytes
L1 I/D Cache	16 KB, 4-way
L2 Cache	256 KB, 8-way
DCE Size	8192 entries
Network Type	Mesh with 2 VNC
Hop Latency	3 cycles
Link BW	16 bytes/cycle
Memory Bus Latency	250 cycles

Table 3.1: Configuration Parameters

Parameter	Value
Number Memory Controllers	4
Number of channels	4
Data bus width	64 bits
Data bus frequency	1333 MHz
Off-chip BW	42.6 GB/s
Memory Capacity	4 x 1 Gb
Memory Clock Speed	667 MHz
Memory Speed Grade	-15
Number of memory banks	8 per DIMM

Table 3.2: Memory Configuration Parameters

and simulates the whole memory hierarchy including caches and the interconnection network. On the other hand, Opal provides the timing for all the processors and simulates an Out-of-Order processor.

Table 3.1 shows the values for the most important configuration parameters for all the evaluations unless a different configuration is stated. Our configuration uses simple cores with small primary caches to improve the aggregate thread throughput by a high number of processors [28].

For the evaluation of the memory controller techniques a detailed memory controller and DRAM memory model has been added including a power model based on the Micron DDR3

Parameter	Value
Num Processors	16
Instr Window/ROB	16/48 entries
Branch Predictor	YAGS
Technology	70 nm
Frequency	4 GHz
Voltage	1.1 V
Block size	64 bytes
L1 I/D Cache	16 KB, 4-way
L2 Cache	256 KB, 8-way
DCE Size	8192 entries
Prefetcher Streams	32 per MC
Stream entries	8

Parameter	Value
Network Type	Mesh with 2 VNC
Hop Latency	3 cycles
Link BW	16 bytes/cycle
Num Memory Controllers	4
Num of channels	4
Data bus width	64 bits
Data bus frequency	5333 MHz
Off-chip BW	170.4 GB/s
Memory Capacity	4 x 1 Gb
Memory Clock Speed	667 MHz
Memory Speed Grade	-15
Num of memory banks	8 per DIMM

Table 3.3: Configuration Parameters

energy requirements [92]. Static power and area of the Row Cache has been evaluated with Cacti [123]. Table 3.2 shows the configuration values of the memory controller and off-chip bus.

Symbol	Parameter	Value
T_{RCD}	Row to Column command Delay	10 cycles (15ns)
T_{RP}	Row Precharge time	10 cycles (15ns)
T_{CL}	Column access strobe Latency	10 cycles (15ns)
T_{WR}	Write Recovery time	10 cycles (15ns)
T_{RAS}	Row Access Strobe	24 cycles (36ns)
T_{RC}	Row Cycle time	34 cycles (51ns)
T_{RRD}	Row activation to Row activation Delay	4 cycles (6ns)
T_{FAW}	Four bank Activation Window	20 cycles (30ns)
T_{RFC}	Refresh Cycle time	74 cycles (111ns)

Table 3.4: Memory Timing Parameters [91]

In the detailed memory model DRAM parameters have been extracted from a commercial memory [91]. Table 3.4 shows the values of the delays for the different memory operations.

3.1.1 Metrics

To evaluate the proposed techniques we have used as a performance metric the number of Instructions per Cycle (IPC). This metric is widely used in the microarchitecture community and allows to measure the *speed* of the execution by dividing work by time. It also allows to compare the performance of executions that have lasted a different number of cycles or instructions. Similarly, most of the metrics used to evaluate the proposed techniques have been divided by the number of instructions executed in order to compare the usage of a certain resource with different number of executed instructions.

Energy efficiency has been measured in $MIPS^3/W$ or the ED^2 product which are equivalent. The energy x $delay^2$ metric is the most appropriate metric for high performance computers according to Brooks et al. [15] and allows a voltage-and-frequency invariant power-performance characterization. Therefore, improvements seen with this metric are not achievable through voltage or frequency scaling.

In the second part of the thesis we present some performance isolation oriented techniques which require some different metrics to estimate the system fairness. System throughput in this case has been measured using Weighted speedup [112] and normalized performance with respect to the baseline configuration, a FR-FCFS scheduler.

$$W.Speedup = \sum_{i=0}^{N} \left(\frac{IPC_i}{IPC_i^{FR-FCFS}} \right)$$

In order to measure the performance isolation in the memory system we have used two different metrics, the average memory latency of each application and its standard deviation. These metrics have been computed individually for each benchmark from all the executions with other applications. The standard deviation allows to see if the execution in conjunction with very demanding applications degrades the memory latency for each of the evaluated benchmarks.

3.2 Power Model

Power consumption reduction is a very important issue in current computer architectures. From high-end servers to laptops it is desirable to reduce the power density and the overall energy consumed. High-end servers require a small power consumption to reduce the energy budget of companies and portable devices like laptops need to use as less energy as possible to increase the battery lifetime. In all cases, a reduction in the energy density

allows lower operating temperatures that yield a higher reliability and also simpler cooling mechanisms. Therefore, we see that novel improvements in computer architecture must focus not only on performance; energy consumption must be also taken into account.

Therefore, in this thesis an architectural-level power model has been added to the existing simulator to allow the characterization of the power consumption in the network and the memory hierarchy. The model is derived from Orion [132] for modeling the buffers, crossbars, arbiters and links with some improvements in the router models. We have estimated the capacitances of all this components taking the technology parameters from Cacti [123]. Also, we have used Cacti to calculate the dynamic and static energy consumption of all the caches. And finally, the DRAM power model has been extracted from Micron power data [92].

The implemented power model has been validated against data of real multiprocessors. We have compared our implementation against power numbers of the MIT Raw chip multiprocessor [61] and the ASIC design of Mullins [96]. Validation results show a relative error of about 10%. We also include a power estimation for the cores based on power values found for similar configurations in the literature [94].

In the following sections we explain how the dynamic and static energy consumption is calculated in the implemented power model.

3.2.1 Power Calculation Methods

3.2.2 Dynamic Power

In all digital circuits the dynamic power dissipation is calculated based on the supply voltage, the load capacitance and the activity of the measured component.

$$P = aC_L V_{dd}^2 f$$

Therefore, the energy consumption for a given logic is:

$$E = C_L V_{dd}^2 P_{0->1}$$

Where C is the load capacitance , V the supply voltage and P the probability that the device consumes energy. The implemented power model calculates the capacity for every logic element and the interconnects with Cacti functions using the transistor sizes. Dynamic consumption is only produced when the load capacitance is charged; this is when the studied node changes from state 0 to 1. In our power model multiple activity counters keep track

Parameter	Value
Technology (λ)	70 nm
Frequency	4 GHz
Voltage	1.1 V
L1 Avg Temperature	75 °C
L2 Avg Temperature	60 °C
Buffers Avg Temperature	60 °C
Inter-node distance	4 mm
Intra-node distance	1 mm

Table 3.5: Power related configuration parameters

of the activity and calculate the overall power. Whenever it has been necessary to compute the hamming distance between two consecutive blocks of data we have considered a value of 0.5.

3.2.3 Static Power

Leakage power has been increasing exponentially with technology and is one of the major contributors to the total power dissipation in current microprocessors. Cacti 4 implements the transistor level model proposed by Zhang et al. [139] and also implemented in eCacti [86]. This model is shown to be accurate and allows to evaluate the effect of variations in temperature and supply voltage. Leakage current is calculated with the following equation:

$$I_{Lkg} = \mu_0 C_{Ox} \frac{W}{L} e^{b(V_{dd} - V_{dd0})} v_t^2 \left(1 - e^{\frac{-V_{dd}}{v_t}}\right) e^{\frac{-V_{th} - V_{off}}{n v_t}}$$

Most of the equation, however only depends on the technology and supply voltage which is constant. Therefore, for a given temperature and transistor length (L) we can calculate the leakage of a transistor with aspect ratio 1 and then the leakage can be directly calculated for each transistor by simply multiplying by the assigned width value.

$$I_{Lkg} = W I_{lkg1}(T)$$

We have evaluated the leakage power for all the data structures with Cacti using the voltage and operation temperature provided in Table 3.5 and added the resulting energy consumption to the power model.

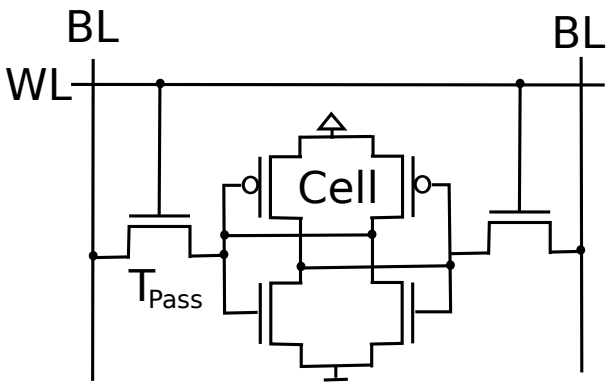


Figure 3.2: Typical structure of a 6T Memory Cell.

3.2.4 Cache

For calculating cache power consumption an interface has been implemented to allow communication between the timing simulator and Cacti [123]. Our implementation allows the simulator automatically calculate the power of the different L1 and L2 caches with the parameters given in the simulator configuration file and then calculate the access energy and the leakage power. Cacti is a well known program that allows to calculate the area, delay and power consumption of a given cache configuration. Cacti 4 allows simulating caches with technologies up to 70 nm and uses 6T memory cells like the one depicted in Figure 3.2. Technology parameters are derived from scaling the values of the 0.8 um process and adapted through ITRS projections [7].

The cache is divided into seven main components for which the delay, area and power consumption is estimated and then the global values are calculated combining them. These main cache parts are the decoder, wordlines, bitlines, sense amplifiers, comparators, multiplexer drivers and output drivers. Cacti gives an estimation of the dynamic and static energy for each of these components and also the overall energy for a read or write operation. Then we combine this information with the activity counters introduced in the timing simulator and we obtain the total power consumption.

3.2.5 Network

This architectural-level model calculates power consumption in the same way that Cacti does. Load capacitances are calculated for each component and then dynamic power is derived from them and the activity counters. We have calculated the energy consumption of buffers, routers and interconnect and added activity counters to the simulator in order to calculate the overall network power.

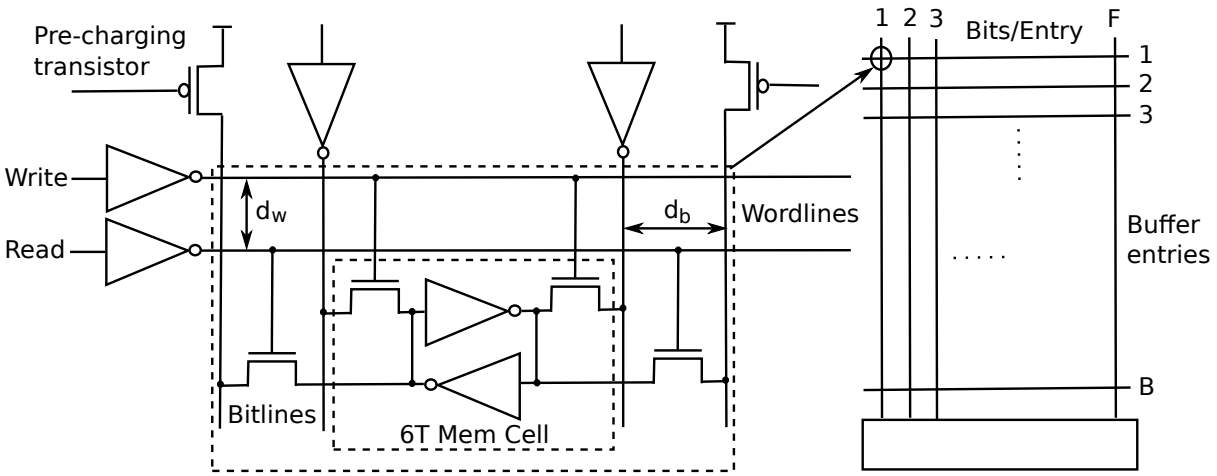


Figure 3.3: Buffer structure.

Buffers

For the input buffers of the network, a common 6T memory cell has been modeled, however it has 8 transistors because it has one write and one read port. Figure 3.3 shows the structure of a buffer bit. Energy is calculated in the same way as in the Orion power model [132] and transistor sizes are shown in Table 3.6.

Symbol	Parameter	Value
T_{inv}	Inverter Transistor	12λ (P) 6λ (N)
T_{pr}	Pass Transistor connecting read ports and memory cells	10λ (N)
T_{pw}	Pass Transistor connecting write ports and memory cells	5λ (N)
d_w	Wordline spacing	15λ
d_b	Bitline spacing	15λ
h_{cell}	Memory cell width	40λ
w_{cell}	Memory cell height	20λ

Table 3.6: Buffer transistor sizes

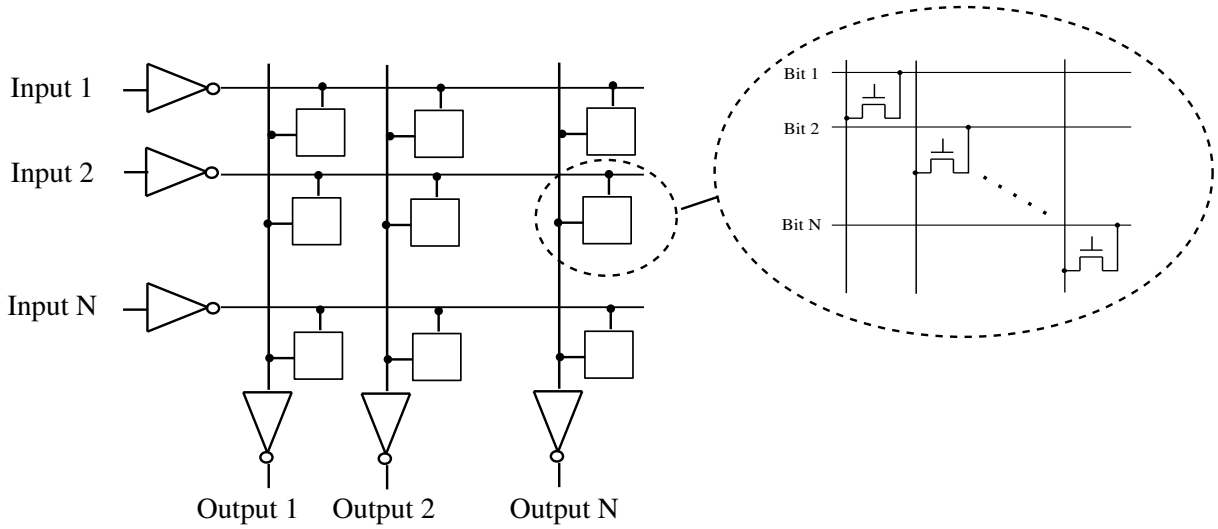


Figure 3.4: Crossbar structure.

Routers

Calculations of the router energy are divided into its main components, the crossbar interconnect (E_{Xbar}), the arbiter (E_{Arb}) and the routing table (E_{rt}).

$$E_{router} = E_{Xbar} + E_{Arb} + E_{rt}$$

The crossbar is the element that allows connecting an input to an output. A matrix crossbar with transmission gate connectors was implemented in the simulator like the one depicted in Figure 3.4.

The arbiter is the part of the router that checks the priority of all the messages in the queues and decides which one is going to cross the crossbar the next cycle according to a given policy. In the implemented model each virtual network (VNC) has a different priority which is fixed and higher for the response message VNC to avoid deadlocks. On the other hand, inputs follow a round-robin policy. Both structures can be seen in Figure 3.5.

The energy consumption of these parts has been modeled as in Orion. In addition, we have added a model for the Routing Table. The routing table is the part of the router that has the information of the possible outputs for a given message. Message address is compared with a destination vector for each output of the router and one or more matches are produced. Figure 3.6 shows the structure of the modeled routing table.

The Routing Table energy is then derived from:

- 3 Input OR gate (Puts together signals of addresses to a same node): Always only one is activated and the previous one deactivated.

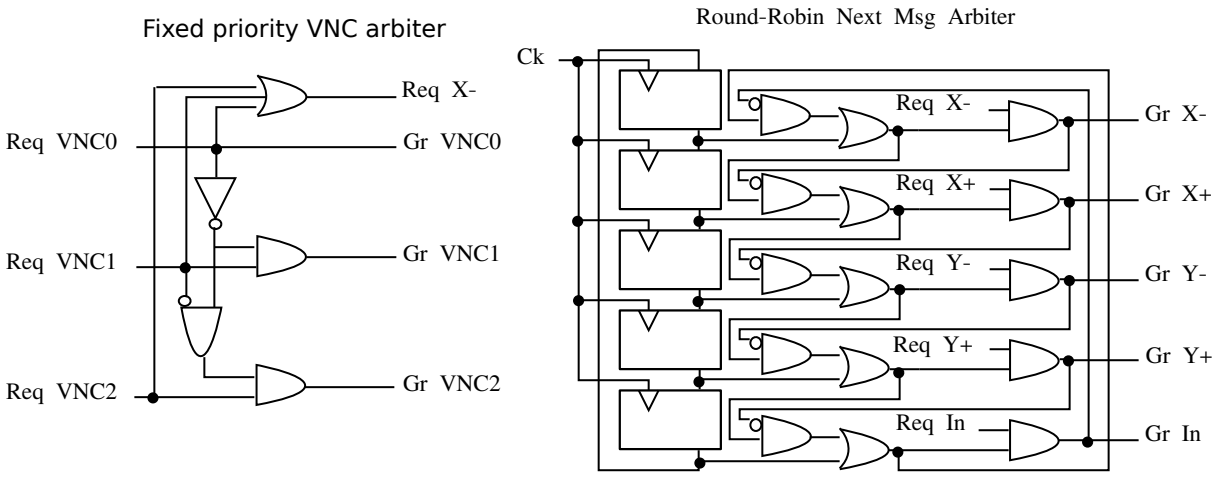


Figure 3.5: Arbiter structure.

Symbol	Parameter	Value
T_{conn}	Crossbar Connector Transistor	20λ (P) 10λ (N)
w_t	Crossbar wire pitch	15λ (N)
T_{arb}	Arbiter and RT Transistor	76λ (P) 13.5λ (N)

Table 3.7: Router transistor sizes

- Hardware Routing Table (Implemented with 2 Input OR gates): The number of gates increases significantly with the number of nodes. However only the ones associated to the activated outputs are used.

Modeling assumption: The number of activated OR gates is approximated to NumNodes/4. However it is bigger in routers with less number of outputs than the considered for each configuration. It is assumed also that only two bits change of state every cycle (Old and new destination) despite the destination may be the same. Therefore, the total capacitance of the routing table is then:

$$C_{rt} = 2\left(C_a(T_{OR3}) + \frac{N}{4}C_a(T_{OR2})\right)$$

Where C_a represents the sum of the gate and the drain capacitances of the transistors. Transistor sizes for router components can be found Table 3.7

Interconnect

Finally, we have also added the power consumption due to the interconnects. The interconnection network capacitance has two main components; the wire capacitance and the

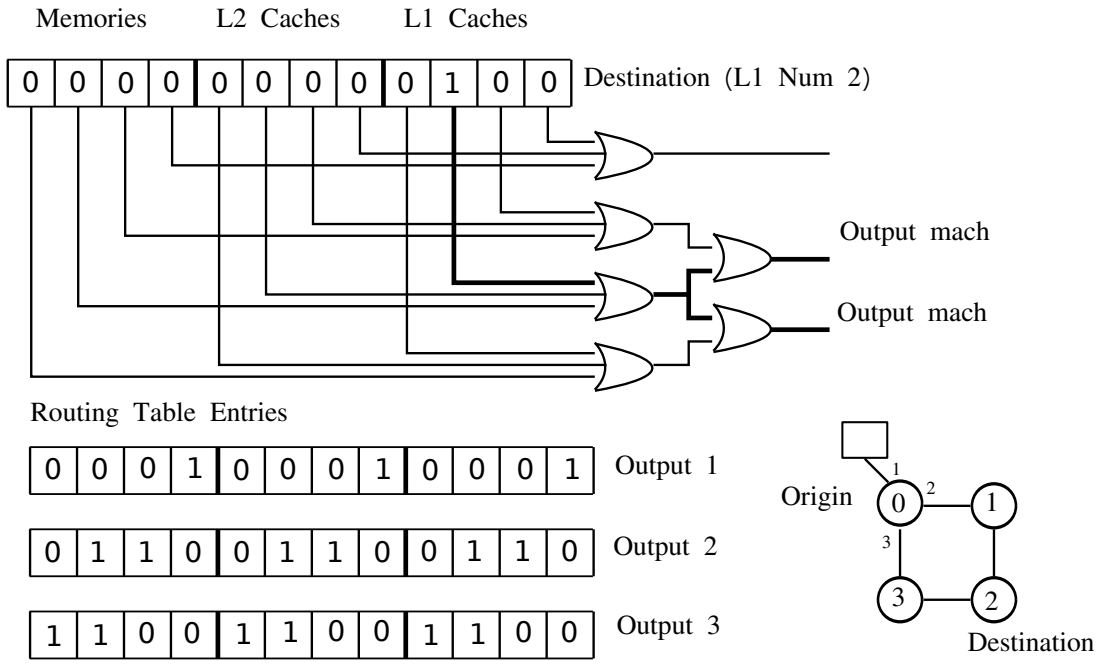


Figure 3.6: Routing Table structure.

drivers capacitance. Since interconnects between nodes are very long, the load capacitance is very high. A common technique to reduce the delay of this lines is to use several drivers.

To calculate the number of drivers first we need to know the relative Fanout, this is the load capacitance (C_L) respect the input capacitance of a unit inverter.

$$f = \frac{C_L}{C_g(T_{inv})}$$

If we put several drivers in cascade with an optimal relative fanout between them we can reduce the delay. This optimal relative fanout is proved to be e if we do not have parasitic delays and we can calculate it for different number of drivers (n) with the following formula:

$$f = \sqrt[n]{\frac{C_L}{C_1}}$$

The delay of all the wire is going to be:

$$t_d = n\,(f t_{el})$$

Checking the delay for different number of drivers we can get the optimal configuration. Transistor sizes are going to be:

$$W_1 = f^0, W_2 = f^1, W_3 = f^2 ... W_n = f^{n-1}$$

So the interconnection capacitance for each wire is:

$$C_w = C_{wire}(L) + \sum_{i=0}^{n} (C_g(T_{Wi}) + C_d(T_{Wi}))$$

We have found minimal delay using 4 drivers for local interconnects and 6 drivers for node-to-node interconnects.

3.3 Benchmarks and Characterization

An important part in any evaluation of new microarchitecture improvements are the applications that are used to prove the usefulness of these improvements. These applications must be real and be representative of the behavior of most common applications.

In this section, we present the benchmarks that have been used in this thesis with an evaluation of their main characteristics. All the applications used in this thesis are from the Standard Performance Evaluation Corporation, a non-profit organization that aims to "produce, establish, maintain and endorse a standardized set" of performance benchmarks for computers. We have used two main sets; the SPEC OMP 2001 and the SPEC CPU 2006.

3.3.1 SPEC OMP2001 and SPEC CPU2006

The SPEC OMP2001 benchmark set is a suite of multithreaded programs in Fortran and C which have been made parallel using the OpenMP API. These applications are representative of High Performance Computing (HPC) applications and threads share data in various degrees. We selected these applications because of the different cache requirements which allows to test cache organizations with different usage conditions.

A more detailed description can be found in the evaluation of Aslot et al. [5]. Based on this study the simulation points for each application were found and all simulations have been done using them as starting point. Table 3.8 shows the details for each benchmark.

To have also a set of uniprogrammed applications from a wider range of fields the SPEC CPU2006 [40] benchmark set also has been used. Applications from this set are single threaded and represent a more broader set of computing applications. Sharing in this benchmark set is almost nonexistent since every thread has an independent memory space. However, they are useful in order to stress the memory system and to allow the evaluation of applications not related to HPC. The high memory requirements of some of

Name	File	Function	Line #
ammp	rectmm.c	mmfvupdate	341
applu	ssor.f	ssor	122
apsi	apsi.f	run	1038
art	scanner.c	scanreco	1545
equake	quake.c	smvp	1270
fma3d	platq.f90	platq_internal_forces	259
gafort	gafort.f90	shuffle	1089
galgel	sysnsN.f90	sysnsn	23
mgrid	mgrid.f	resid	360
swim	swim.f	calc1	276
wupwise	muldoe.f	muldoe	63

Table 3.8: SPEC OMP2001 evaluated starting point

these applications are specially useful to stress the memory controller in the second part of this thesis.

Table 3.9 shows the starting simulation points derived from a detailed profiling of applications [134].

3.3.2 Benchmark set 1

Since one of the main motivations of this thesis is to optimize the cache usage, it is important to know the influence of last-level cache sizes for the different benchmarks. The design of dynamic memory hierarchies requires the implementation of policies to distribute cache resources. Applications can benefit from cache memory up to a certain point and the allocation mechanism must know the individual needs of each of them.

In this section we study the behavior of applications in order to analyze the potential performance benefit of reallocating cache resources. Therefore, all the SPEC OMP2001 applications were evaluated with different private L2 cache sizes ranging from 64KB to 1MB. Figure 3.7 shows the IPC, the speedups compared to the 256KB configuration and the number of L2 cache misses per instruction. All the characterization was done with a 8 processor configuration running every application with 8 threads. It can be seen that most of the applications do not scale very good for this region of the execution except for the art benchmark which is able to have an IPC of more than 16. Regarding the sensitivity of applications, we can divide them between sensitive and non-sensitive applications. Ammp,

Name	File	Function	Line #
400.perlbench	regexec.c	s_regmatch	2324
401.bzip2	decompress.c	BZZ_decompress	235
403.gcc	regmove.c	reg_is_remote_constant_p	869
429.mcf	pbeammp.c	primal_bea_mpp	133
445.gobmk	matchpat.c	do_matchpat	229
456.hmmer	fast_algorithms.c	P7Viterbi	106
458.sjeng	neval.c	std_eval	405
462.libquantum	gates.c	quantum_toffoli	83
464.h264ref	mv-search.c	SetupFastFullPelSearch	327
471.omnetpp	cmsgheap.cc	c MessageHeap::shiftup	196
473.astar	Way_.cpp	wayobj::makebound2	53
483.xalancbmk	ValueStore.cpp	ValueStore::contains	267
410.bwaves	block_solver.f	mat_times_vec_	166
433.milc	m_mat_na.c	mult_su3_na	17
434.zeusmp	hsmoc.f	hsmoc_	594
435.gromacs	innerf.f	inl1130_	3920
436.cactusADM	StaggeredLeapFrog2.F	bench_staggeredleapfrog2_	301
437.leslie3d	tml.f	fluxk_	1268
444.namd	ComputeNonbondedUtil.C	select	144
447.dealII	dof_constraints.cc	ConstraintMatrix::add_line	89
450.soplex	ssvector.cc	assign2productFull	978
453.povray	spheres.cpp	pov::Intersect_Sphere	281
454.calculix	e_c3d.f	e_c3d_	129
459.GemsFDTD	update.F90	updateE_homo_	191
470.Lbm	lbm.c	LBM_performStreamCollide	180
481.wrf	module_advect_em.F90	advect_scalar	2790
482.sphinx3	cont_mgau.c	mgau_eval	591

Table 3.9: SPEC OMP2001 evaluated starting point

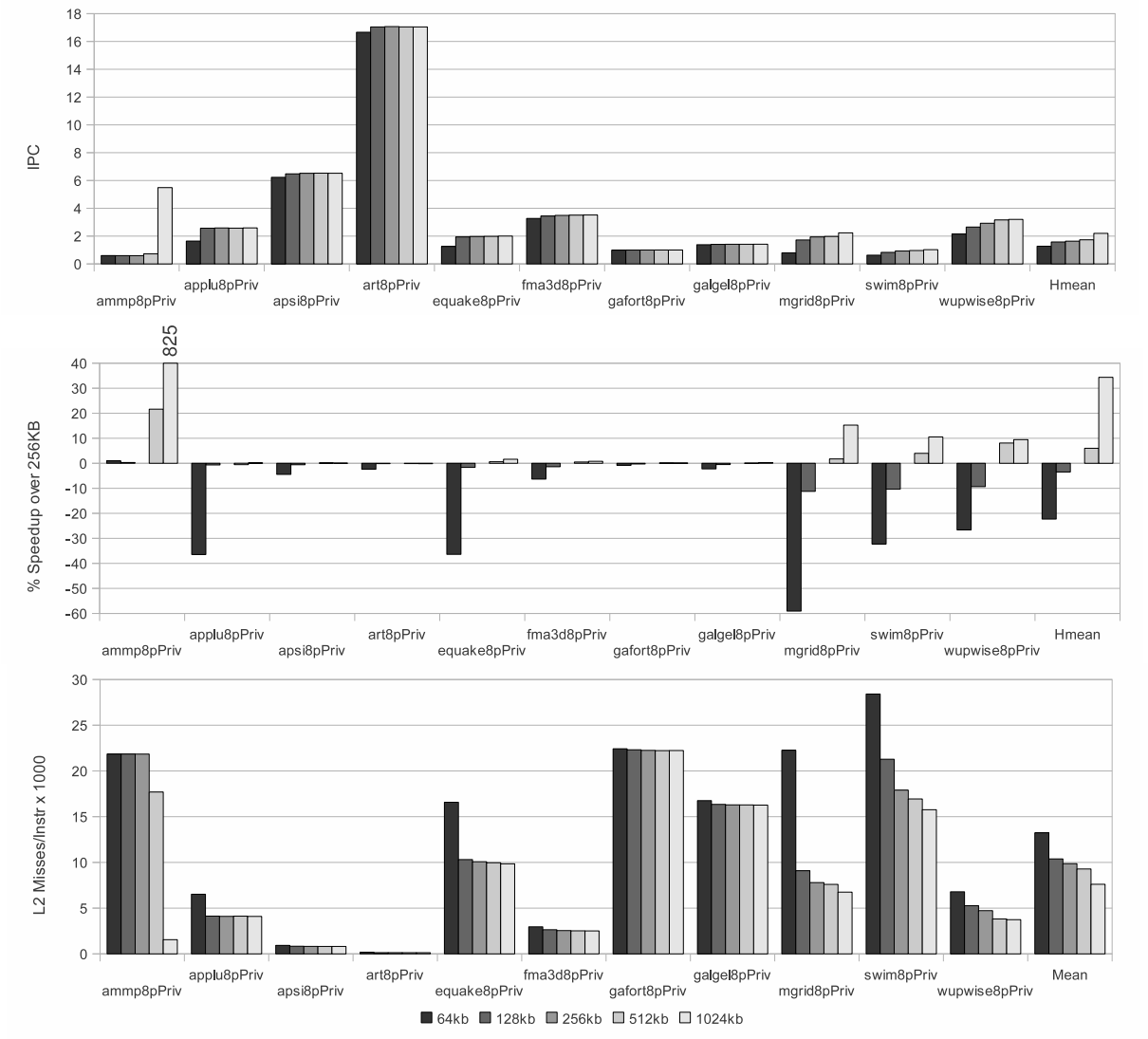


Figure 3.7: SPEC OMP Characterization.

mgrid, swim and wupwise are highly affected by the cache sizes while the others do not change their performance significantly. Only in the case of applu and equake when the cache size is 64KB have a big degradation in performance, probably because the working set does not fit anymore in the cache. Finally, if we look at the number of L2 cache misses we can see that is highly correlated with the sensitivity of applications to cache sizes. Ammp is the application with a bigger impact in the reduction of cache misses when it reaches a cache size of 1MB.

A common technique that is used to repartition cache resources is the column caching technique [21]. Therefore, it is interesting to study the SpecOMP benchmarks for varying number of ways and sizes. Figure 3.8 shows the evaluation of the SpecOMP benchmark set

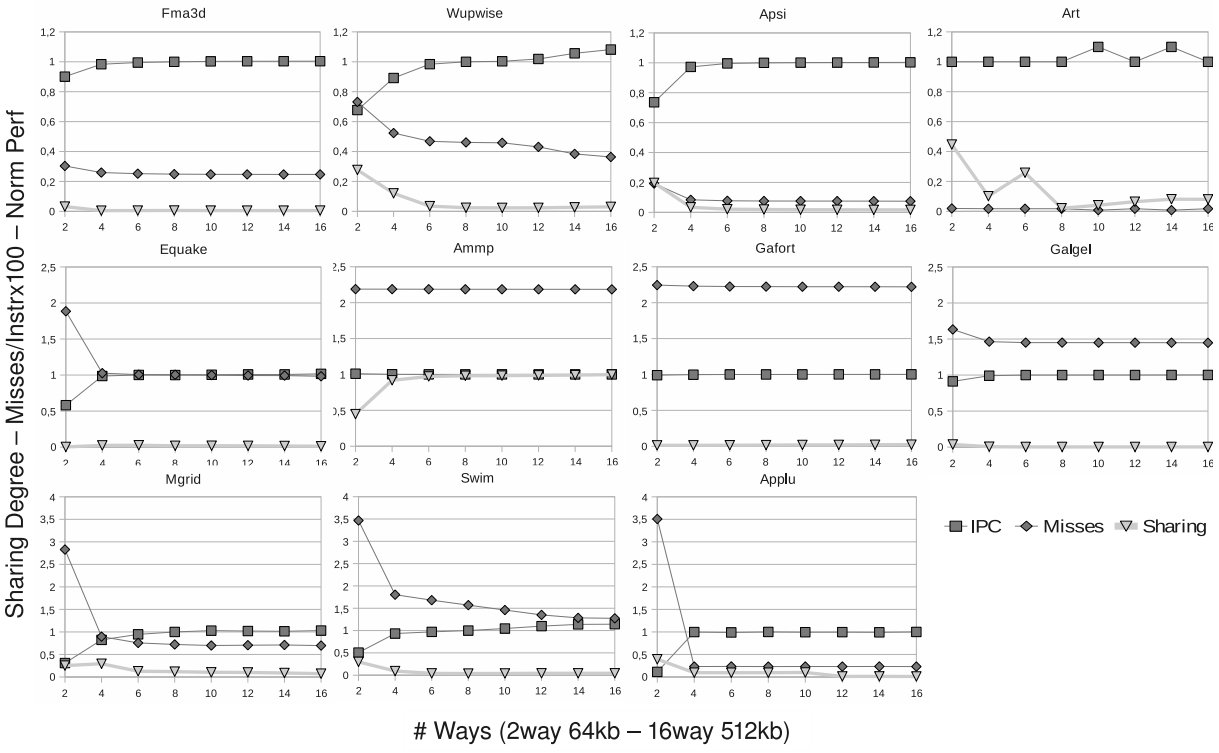


Figure 3.8: Spec OMP 2001 benchmark characteristics for different L2 sizes/associativity.

with 8 nodes, private 16kb-4way L1s and private L2s of varying size and associativity. Size is incremented with the addition of extra ways to evaluate the benefits in way-partitioned caches. Execution is started in each benchmark's most important parallel regions once all data structures are initialized. Three different parameters are shown in Figure 3.8; normalized performance compared to the 8way 256kb configuration, number of misses per instruction, and the sharing relation. Each row uses different graph scales to allow a better clarity in application behavior variations. The sharing relation is the average number of nodes that share a block before its eviction and indicates if threads have independent data sets. Therefore, this parameter shows the potential savings in cache space a shared cache would provide by reducing replication.

Previous studies of dynamic cache repartitioning [104] have divided applications in three categories; low utility, high utility and saturating utility. However, after evaluating all applications we have detected that these categories do not consider the amount of sharing, which is important for an efficient cache partitioning.

Therefore, we have divided the benchmarks into four categories:

Saturating Utility: These type of applications are characterized by having a small working set that fits in the cache. Therefore, granting more cache space to them has no impact

Type	Benchmark
Saturating Utility	Applu, **Apsi**, **Art**, **Equake**, Fma3d
Low Utility	**Gafort**, Galgel
Shared High Utility	**Ammp**
Private High Utility	Mgrid, **Swim**, Wupwise

Table 3.10: Benchmark Classification

on their performance. These applications are characterized by improving performance with each increase of cache size until the working set fits in it. If extra cache space is provided, performance is not affected (e.g. Equake).

Low Utility: This category is for benchmarks with low temporal locality that make an intensive use of the memory hierarchy but do not have reuse. These applications are especially harmful for competing benchmarks that would benefit from more cache space. They are characterized by not improving performance when we increase cache sizes (e.g. Gafort).

Shared High Utility: In these applications, there are several threads that share a large number of blocks. Therefore, to optimize cache usage replication should be reduced for shared blocks reducing private regions. Replication in highly reused blocks, however, is still granted by L1 caches. These applications are characterized by a high sharing (e.g. Ammp).

Private High Utility: Finally, Private High Utility applications are those that benefit from larger levels of memory hierarchy but do not share data between threads. They are characterized by always improving performance when cache size is increased and by low data sharing among nodes (e.g. Swim).

During the evaluation of cache organizations we have used a set of multiprogrammed configurations with benchmarks from each category to study the behavior of our dynamic cache organizations under all possible combinations. Table 3.10 shows the classification of the SpecOMP benchmarks in the previously defined categories with tested benchmarks in bold. The usage of a multiprogrammed benchmark set allows to see the influence of memory intensive benchmarks over applications with lower cache requirements.

3.3.3 Prefetch Influence in memory access patterns

In the last section of this thesis, several techniques have been evaluated to improve the behavior of the memory controller in multiprogrammed environments. As we have seen

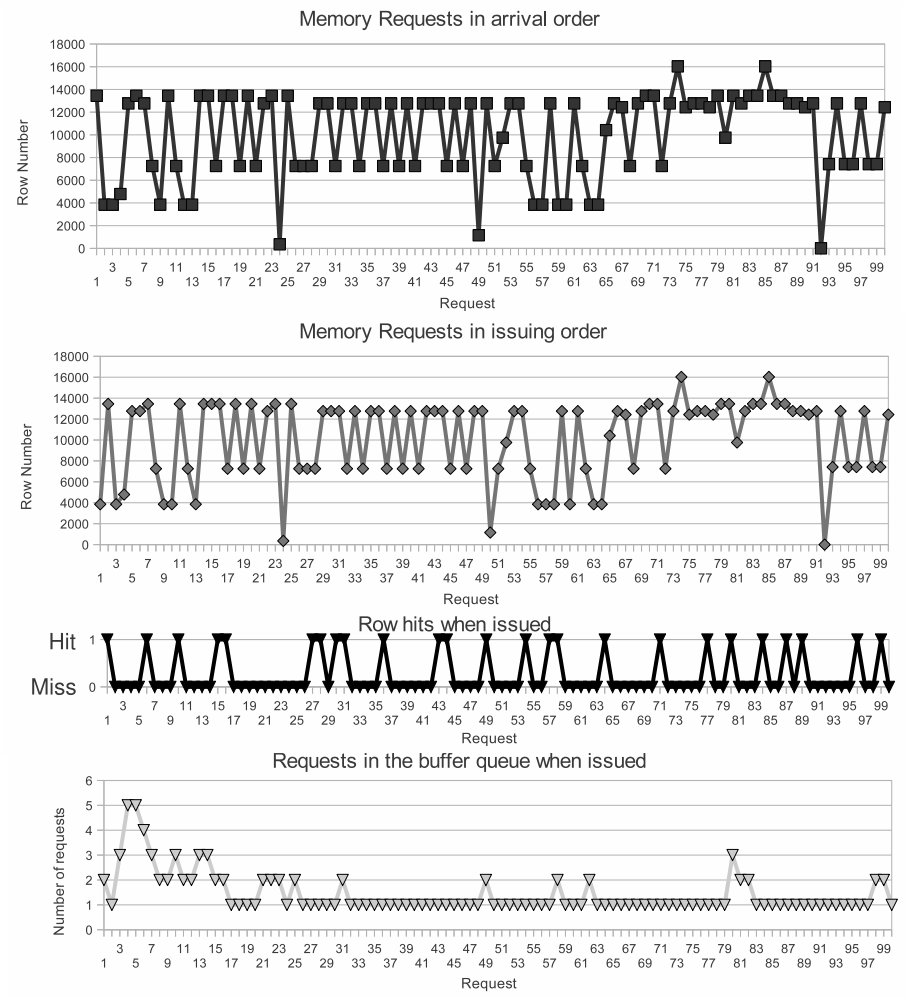


Figure 3.9: Request Behavior in Bank 0 in a multiprogrammed environment

from the memory organization in chapter 2, it is very important to achieve the maximum hit rate to minimize the number of row activations and, therefore, reduce access latency and energy consumption. In configurations running a single application the usage of a FR-FCFS policy achieves a reasonably good row hit rate [108]. The chip multiprocessor consolidation, however, has brought new execution environments where this solution is insufficient.

Figure 3.9 shows the requests received in Bank 0 of a multiprogrammed execution (over a short sample). The first plot shows the row numbers of requests in arrival order and the second one the order in which these requests are finally issued. It can be seen that, although FR-FCFS reorders some of the requests (for ex. Req 7 is issued before Req 6), we can see an alternation in the rows being accessed; leaving room for optimization. The request reordering mechanism is not able to reorder more requests because, as can be

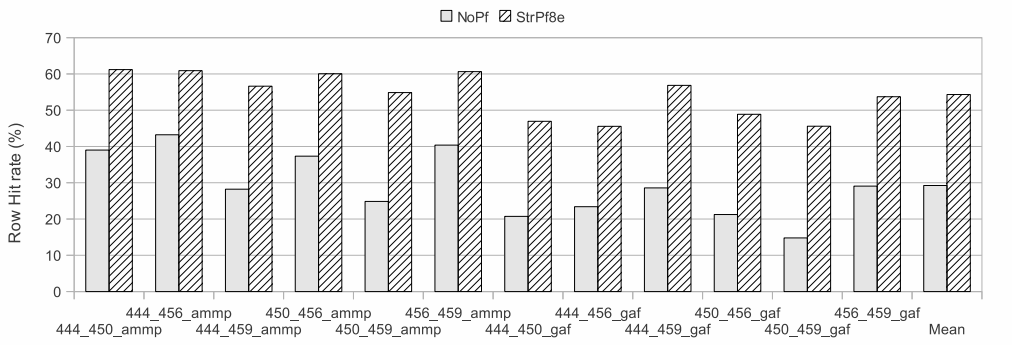


Figure 3.10: Row hit rate with and without prefetch.

seen in the last plot, the number of requests waiting in the buffer queue is small. This small amount of requests is explained by the limited memory level parallelism of applications.

One way of improving row hits and reduce the row alternation is to use on-chip stream prefetchers [57] to group requests. Stream prefetchers increase the MLP and do not interfere with caches. Therefore, we have evaluated the influence of stream prefetchers in the memory controller.

Figure 3.10[1] shows how a the addition of 8-entry stream prefetchers in the memory controller can significantly improve the row hit rate, by generating extra memory parallelism and grouping requests. On average, row hit rate increases from 29.2% to 54.3%.

Figure 3.11 shows the performance improvement brought by a stream prefetcher [57] integrated in the memory controller. Every prefetcher has 32 stream buffers of 16 entries each. Two other configurations were also simulated to see the influence of the bus latency in the overall performance. The first configuration (100Lat) adds 100 extra cycles to all memory requests to see the sensitivity of applications to the extra latency brought by a more saturated off-chip bus. In the opposite side, another configuration (Ideal16e) evaluates the performance improvement brought by the prefetchers when prefetch requests do not use the bus and have no extra latency due to bus contention.

In general it can be seen that those applications that benefit more from prefetching are the ones that are more affected by an increased memory access latency. The second plot of Figure 3.11 shows the prefetch accuracy of the stream prefetcher. This is the percentage of prefetched blocks which are used.

Figure 3.12 shows the memory row hit rate, which depends on the amount of pending requests that need to access to the same row. Prefetching greatly helps in all cases to

[1] Combination of two SPECCPU (4 copies of each) and one SPECOMP (with 8 threads) benchmarks. Further details of the simulation environment can be found in Section 3.1.

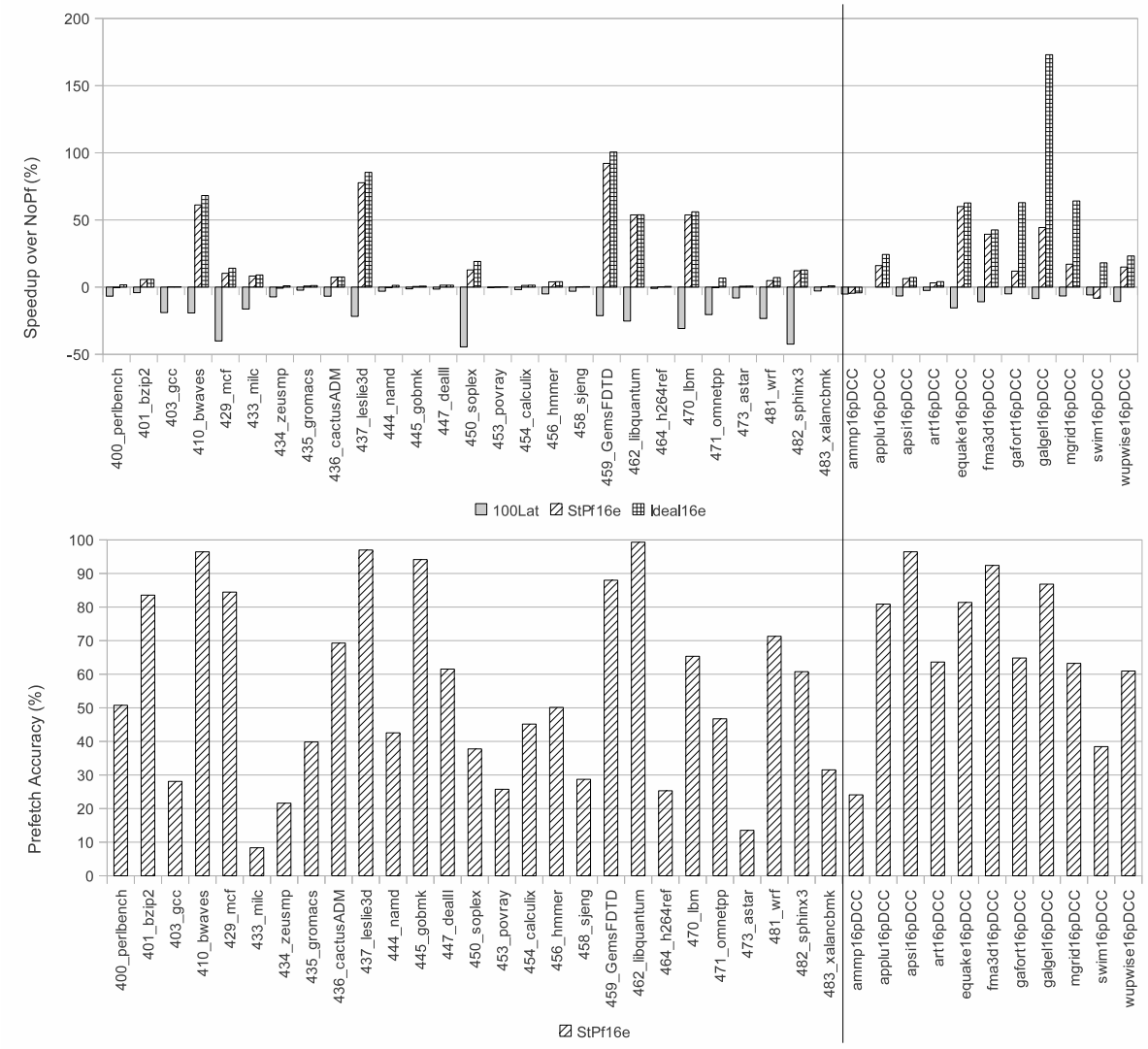


Figure 3.11: Prefetch influence on performance (Speedup and accuracy).

increase the hit rate by grouping requests together and shows that prefetching not only can reduce the data access latency but also reduce the memory usage if the accuracy is high.

One important limitation of the evaluated applications is the limited memory level parallelism that they have. It can be seen in the second plot that stream prefetchers greatly help to improve this parallelism, also allowing new request reordering mechanisms to be proposed to improve memory performance.

Finally, the third plot of Figure 3.12 shows the amount of off-chip bandwidth which is used. It must be noted that SPEC CPU applications only run one thread while SPEC OMP run 16 threads. This explains the much higher bandwidth used. Prefetching in some cases

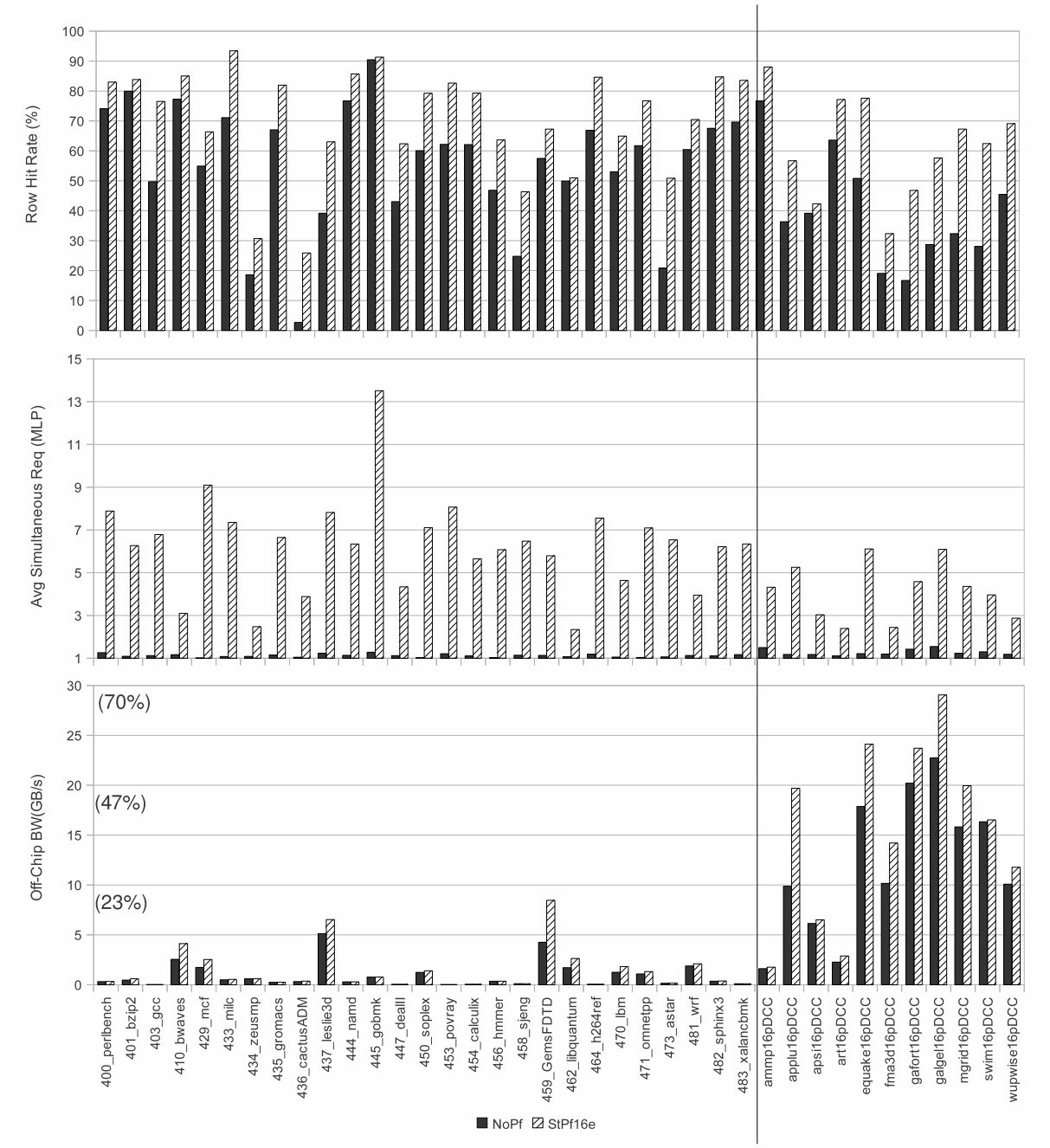


Figure 3.12: Prefetch influence on performance (Row hit rate, MLP and off-chip bandwidth).

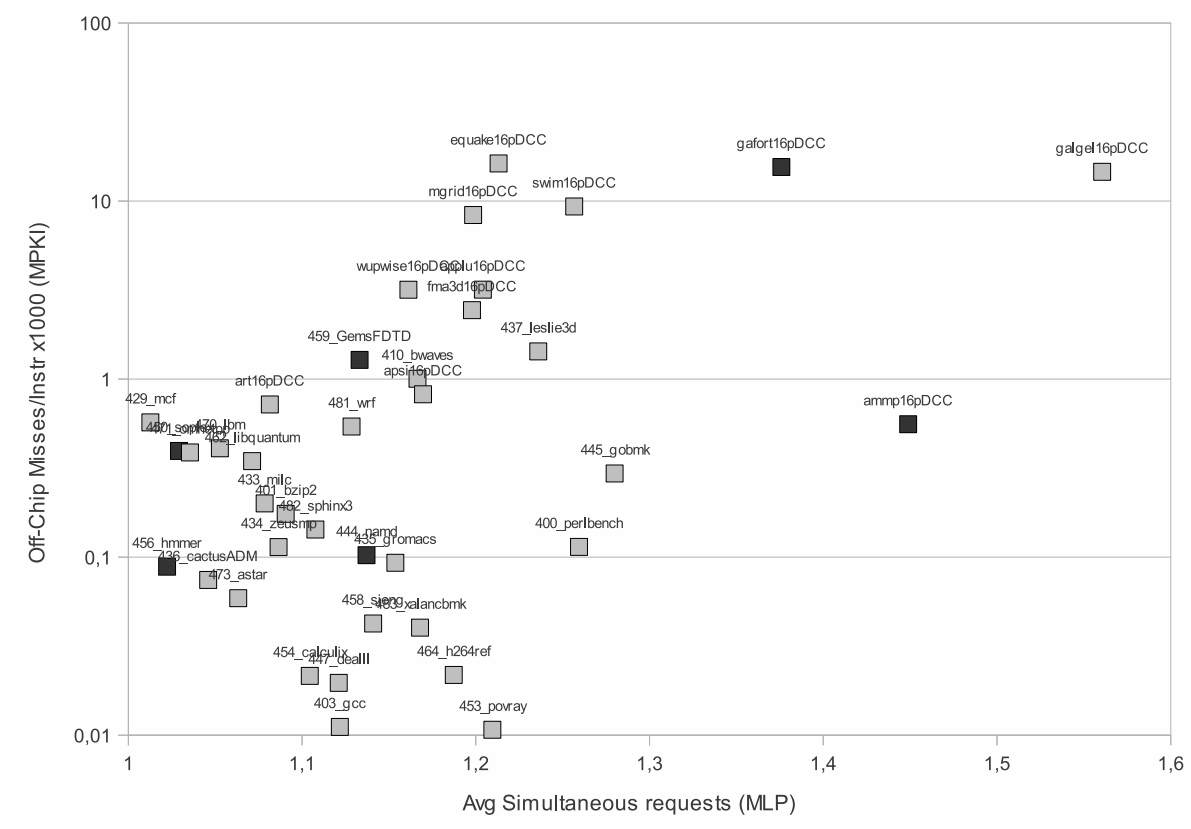


Figure 3.13: Spec OMP2001 and CPU2006 characterization without prefetch.

doubles the amount of data transfered and, therefore, must be used carefully to avoid higher access latencies if the accuracy is low.

Figure 3.13 shows the SPECOMP2001 and SPECCPU2006 application characterization without prefetch. The aforementioned reduced memory level parallelism of applications can be seen since the maximum from all applications is less than 1.6 on average.

Due to these limitations it is important to add on-chip prefetchers to evaluate DRAM memory management techniques since they already solve some of the problems of multiprogrammed environments. Therefore, we have used as baseline a configuration with stream prefetchers in the memory controller.

3.3.4 Benchmark set 2

To emulate the multiprogrammed execution environment of current chip multiprocessors we have used a mix of single and multi-threaded applications. Two multi-threaded applications from the SPECOMP2001 workload set have been combined with single-threaded applications from the SPECCPU2006 workload set. All applications are simulated with reference input sets. In each combination one SPECOMP2001 application is executed with 8 threads in combination with two SPECCPU2006 applications, each of them running 4 copies with independent input data.

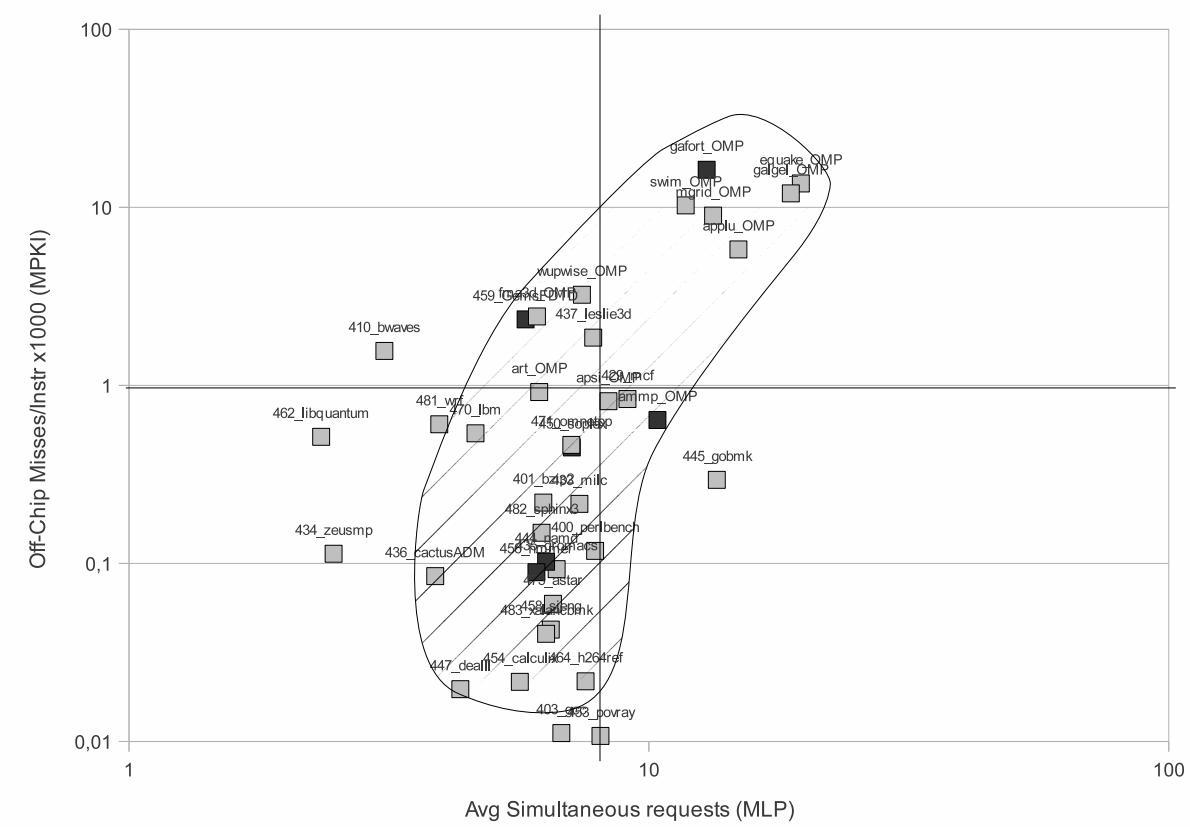


Figure 3.14: Spec OMP2001 and CPU2006 classification with prefetch.

Figure 3.14 shows with a shaded area the region with more representative applications executed with stream prefetchers. The addition of prefetchers significantly increases the memory level parallelism to an average of 8 simultaneous requests, avoiding unsuitable access patterns. From the shaded area, applications have been divided in 4 categories depending on the number of off-chip misses (MPKI) and the memory level parallelism (MLP). Figure 3.14 shows the division, which considers applications with more than 1 miss per

1000 instructions to have a high number misses and to have a high MLP when the average number of simultaneous requests is higher than 8. Then we have selected representative applications from each category to combine them and have all types of applications represented. The evaluated applications are 459.GemsFDTD, Gafort, 450.Soplex, 456.hmmer, 444.namd and Ammp, which have been used in the evaluation of the proposed techniques.

These applications have been synchronized to the most significant execution regions and after warming up the caches they have been executed for 50 million cycles (which represents 400 million instructions on average).

Chapter 4

Distributed Cooperative Caching

4.1 Background and Motivation

As it has been shown in Chapter 2, traditional multiprocessors have mainly used two different ways of granting cache coherence, with snoop or with directory based schemes. Directory based schemes need an extra level of indirection but have a better scalability and can provide the best configuration for multiprocessors with a high number of cores. Since in this thesis we have focused in the optimization of the cache hierarchy of large organizations, we have considered directory-based systems more suitable.

Another important aspect when defining a memory hierarchy is the physical placement of the storage resources. When designing the on-die L2 cache, two different alternatives come up, private and shared caches. Logically-unified shared L2 cache provides a good response for processors with a reduced number of cores because the global number of L2 misses is usually smaller and they make an efficient use of the available L2 cache space. This is the most common organization for the last-level cache due to its simplicity and good performance. However, for a higher number of cores this type of configuration generates a bottleneck in centralized aggregate caches or produces a high demand on the interconnection network for distributed Non Uniform Cache Access (NUCA) architectures. The network usage increase has three negative effects: it increases the overall power consumption, it requires a network with higher bandwidth and it increases the miss latency. Private L2 caches, on the other hand, provide a uniform and usually lower latency since data is stored in the local nodes. These configurations have the additional advantage of avoiding inter-core cache conflicts. However, since not all threads running in the cores have the same cache requirements, there is an inefficient use of L2 cache space, and these caches often

require a higher number of off-chip accesses with the inherent latency and power penalizations.

To find a compromise between these two solutions several proposals have appeared that try to achieve the latency of private configurations and the low number of off-chip accesses of shared configurations [17, 23, 50, 104, 138]. One of the most interesting is the Cooperative Caching framework [17]. This organization duplicates all cache tags in a centralized coherence engine to allow block sharing between nodes and reduce off-chip misses. Furthermore, it uses private L2 caches to allocate blocks closer to the requester and reduce the L1 miss latency. This technique, however, does not have an efficient use of all the cache space so it also implements a forwarding mechanism for the evicted blocks. This mechanism spills replaced blocks to other L2 caches to avoid future off-chip accesses.

This organization has two main limitations. First, the centralized structure of the replicated tags becomes a bottleneck for a high number of nodes; and second, the coherence engine -even if banked- may have a particular address in any of them. This means that all banks must be accessed on each request and a high number of tags must be compared, increasing the power consumption significantly for a system with many processors.

We propose the Distributed Cooperative Caching scheme in order to overcome the power and scalability issues of the Cooperative Caching framework. We have redesigned the coherence engine structure and its allocation mechanism to allow a distribution of the replicated tags across the chip. Our allocation mechanism reduces the number of tags checked on every request thus reducing the energy consumption. Another benefit of our organization is the possibility of having a smaller number of replicated tags while the centralized Cooperative Caching requires a replica for every cache tag. In Cooperative Caching the tag entry of a block that is shared by several caches is going to be allocated in all the tag replicas of the cache entries. On the other hand, in our proposal only one entry is going to be used for each block, making a more efficient use of space. We will show that our organization gets an average performance improvement over the Cooperative Caching of 57% and a $MIPS^3/W$ relation improvement of 4.30x for a 32-core CMP thanks to a request distribution and bottleneck avoidance

Furthermore, we study different power-efficient spilling policies to improve the efficiency of the N-chance forwarding mechanism [44]. Compared to traditional random Spilling, Distance-Aware Spilling technique provides an energy efficiency improvement ($MIPS^3/W$) of 16% on average, and a reduction of the network usage of 14% in a ring configuration while increasing performance 6% for the multiprogrammed SpecOMP benchmark set. On the other hand, the Selective Spilling technique is able to avoid most of the unnecessary

reallocations and double the reuse of spilled blocks, reducing network traffic by an average of 22%. A combination of both techniques allows to reduce the network usage by 30% on average without degrading performance, which leads to an increase of the energy efficiency of 9%.

4.2 Distributed Cooperative Caching

4.2.1 Cooperative Caching

Distributed Cooperative Caching is based on the Cooperative Caching (CC) framework [17] proposed by Chang and Sohi. CMP Cooperative Caching tries to create a globally-managed, "shared", aggregate on-chip cache with private caches. The main objectives of this configuration are to reduce the average miss latency by approaching the blocks to the local node, to improve the cache utilization and to achieve as much performance isolation between nodes as possible. The hardware requirements for this approach are a central directory with a duplicate of all the L1 and L2 cache tags. This directory (Central Coherence Engine) is the responsible for maintaining blocks coherent and to share the blocks between caches. Figure 4.1 shows the memory configuration of this organization.

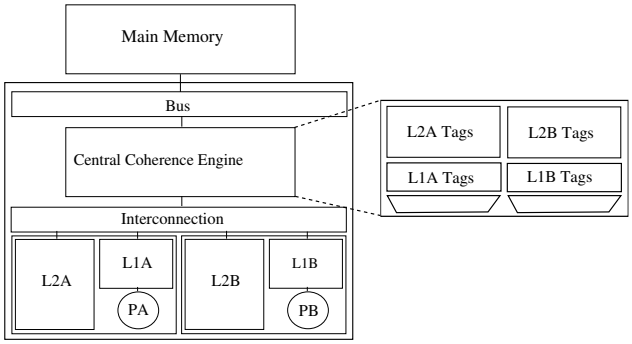


Figure 4.1: CC Memory Structure.

The working principle of this approach is that all the L2 misses are handled by the CCE -which keeps a copy of all the tags. If a particular cache access misses in the local L2 and the block is stored in another cache, the CCE is going to have a hit and the request will be forwarded to the owner. Then the data is sent through a cache-to-cache transfer and the CCE is acknowledged of the end of the transaction.

To be able to use efficiently the cache space, the cooperative caching also implements the N-Chance Forwarding algorithm for replacements. Figure 4.2 shows the working prin-

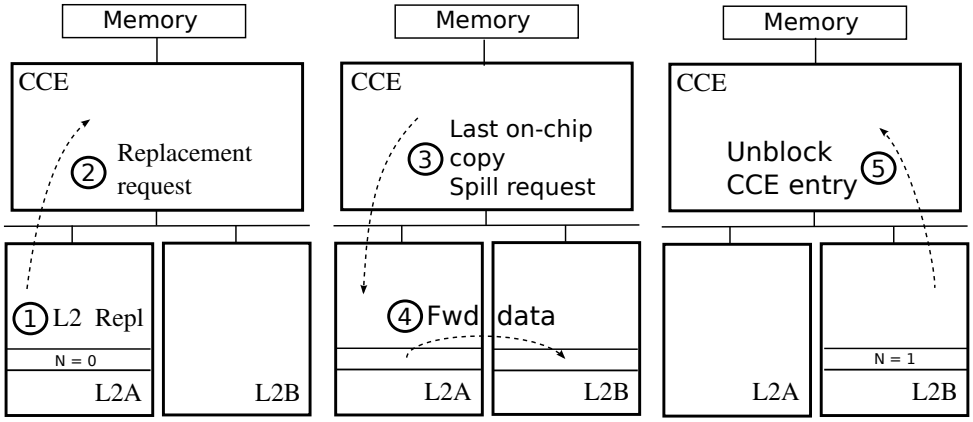


Figure 4.2: Spilling Example.

ciple of this mechanism. When a block is evicted from an L2, the CCE requests the L2 to forward it to another cache if it is the last copy in the chip. To avoid infinite forwardings a counter is set for each block. By default each block is forwarded N times before being evicted from the chip and if the block is reused the counter is reset. To avoid a chain reaction of replacements a spilled block is not allowed to trigger a subsequent spill. When applied to CMP Cooperative Caching, N is set to 1 since a replication control is already employed and further spilling would degrade performance by evicting newer blocks. Therefore, if the block of the example was evicted again from the L2, the CCE would send it back to memory.

This approach, however, has some limitations that the Distributed Cooperative Caching tries to solve. The first one is that a centralized directory presents important restrictions to the scalability of the multiprocessor. The centralized nature of the coherence engine limits the number of processors that can handle without creating a bottleneck and degrading the performance. The second limitation of this configuration is the power consumption of the centralized directory. The number of tags that must be checked on each request increases with the number of nodes, raising also the overall power consumption. Making the CCE scalable is somewhat challenging since the centralized version is already banked and does not behave well for a high number of processors. In the next section, we will further discuss these limitations and our suggested solutions.

4.2.2 The Distributed Cooperative Caching scheme

The Distributed Cooperative Caching (DCC) scheme is designed to solve the scaling issues of the previous configuration by using distributed coherence engines that can be spread

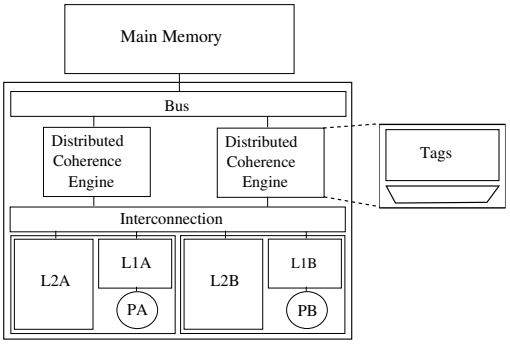


Figure 4.3: DCC Memory Structure.

across the nodes. This avoids bottlenecks and limits the number of tag checks that must be done on each request.

In our approach, the Coherence Engine is partitioned into several Distributed Coherence Engines (DCE) that are responsible for a portion of the address space. The number of DCEs and the number of entries each one has does not depend on the cache sizes. Figure 5.6 shows the memory structure used in our proposal. Addresses in the Coherence Engines are interleaved and every DCE entry stores which nodes are storing that block. On a local L2 cache miss, the corresponding DCE for that address is accessed and if the cache entry is found the request is redirected to the owner.

The organization of the directory in the Distributed Coherence Engine is completely different to the one in the Cooperative Caching scheme. In the DCC framework, tags are interleaved across the DCEs in order to distribute DCC requests across the network and thus avoid bottlenecks. This distribution implies that, unlike the centralized configuration that has a tag for each cache entry, tag entries are allocated just in one DCE depending on its address. As a result, if the entries are not perfectly distributed in the address space, we can have more entries in the caches than in a given DCE set. Because of that, it is necessary to extend the coherence protocol to be able to handle the invalidation of cache blocks due to DCE replacements.

Figure 4.4 shows the organization of the Coherence Engine for both Centralized and Distributed versions of Cooperative Caching. We can see that the organization of the centralized version is formed by a unique structure that has the replicated tags distributed in banks, each one representing a cache. In the DCC, we have an arbitrary number of Coherence Engines that store tags from all caches. We can also see in the figure that the number of tags compared for every request is significantly smaller in the DCC scheme, and this results in a reduced energy consumption. In the Distributed version the number of

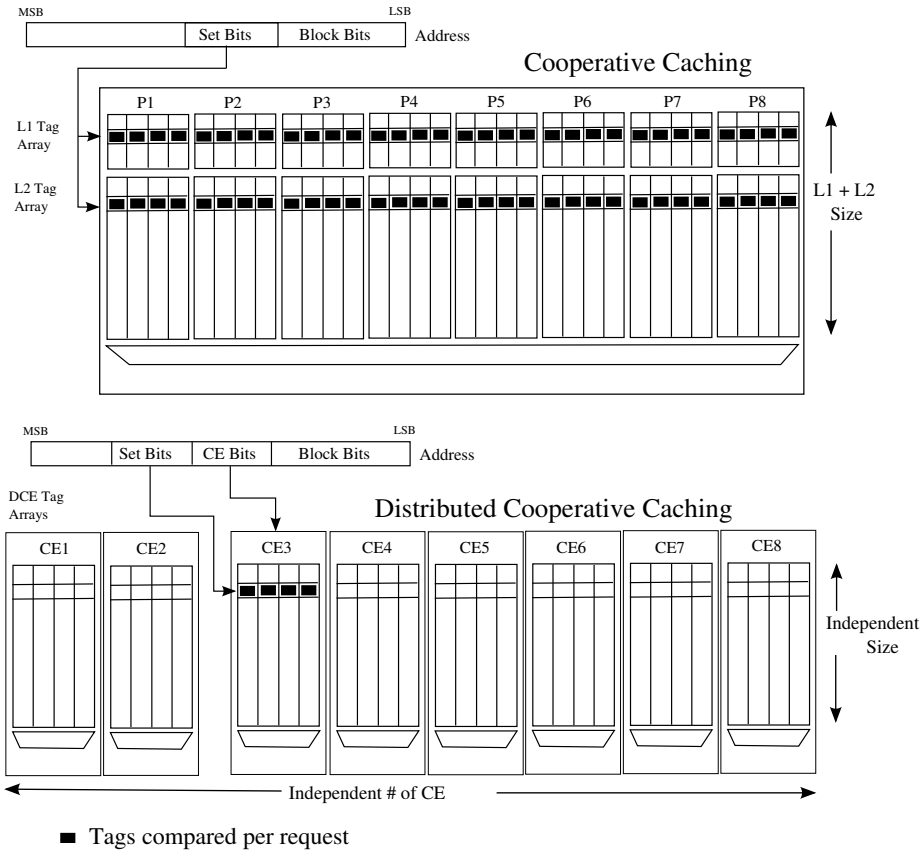


Figure 4.4: Directory structures.

checked tags depends on the associativity of the DCEs. On the other hand, for the Centralized version, the number of tags compared depends on the number of processors and the associativity of their caches. This number increases with the number of nodes; making this configuration suitable only for a CMP with a reduced number of processors. The example of Figure 4.4 shows the Coherence Engines of an 8-core CMP with 4-way associative caches. We can observe that for the centralized version 64 tags are compared while for the distributed version only 4 are compared.

In addition to all these benefits, the Distributed Cooperative Caching also allows hardware design reuse since its modular and scalable structure can be replicated as we add more processors on a chip.

The hardware overhead of DCC compared to CC is the storage area used to keep track of the sharers in each tag. DCEs use Full map directories (Dir-N), which require one bit per sharer. The coherence state machine, in addition, requires 4 bits to maintain the DCE state. This means that if we assume 32 processors and a total L2 of 8 MB (i.e. 256KB per processor), the overhead per DCE would be 18KB. We believe the hardware overhead is

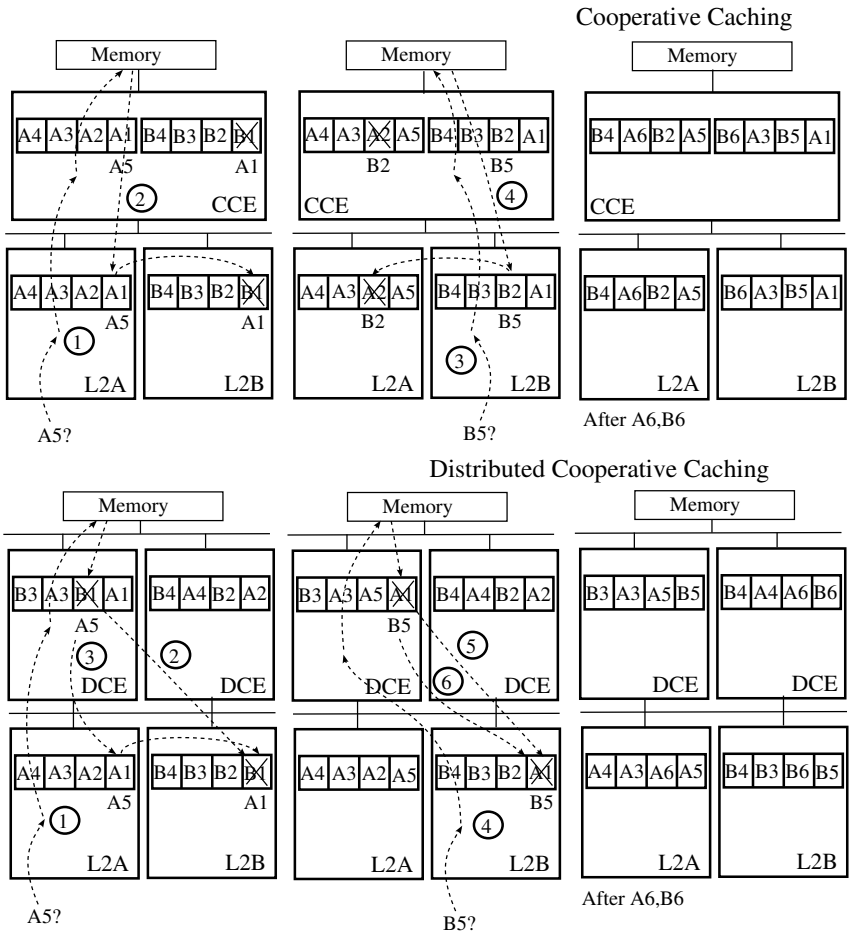


Figure 4.5: Working Example.

reasonable and we do not need to invalidate any sharer, reducing the protocol complexity. This organization, however, may limit the scalability for CMPs with more processors. Partial map directories may be an interesting solution for these configurations but are left for future work.

The DCE replacement policy: an example

To show the benefits of the DCE tag replacement policy, Figure 4.5 demonstrates the working principle of the Centralized and the Distributed versions of Cooperative Caching.

The situation depicted shows the L2 caches and Coherence Engines of a system with two nodes (A and B) for simplicity. It considers the situation of two threads, one per node, that make an extensive use of their caches. It is also considered that node A always makes requests slightly before than node B. Blocks in the cache are represented by the letter of the requesting node and a number that indicates the time when that block was requested.

We start in a warmed-up situation where both caches are full to see how replacements are handled.

In the upper part of the figure the behavior of the Centralized Cooperative Caching is shown. Let's suppose that node A makes a request for a new block (Action 1). In this case, since the block is not in the local L2, the CCE is checked. Since the block is neither in any other cache, memory is accessed. Block A5 is then sent to the requester (Action 2). Since there is not enough space, block A1 is spilled to node B. Block B1 is evicted from the chip since subsequent spillings are not allowed.

In the second request, node B asks also for a block to the CCE (Action 3). Request is forwarded to memory that sends the block to the requester (Action 4). Since there is not enough place, a replacement is done. The locally oldest block, B2, is spilled to node A; evicting from the chip A2.

In the bottom part of Figure 4.5 the behavior of the Distributed Cooperative Caching is depicted. As in the previous case block A5 is requested (Action 1), but now to the corresponding DCE. Since the block is not in any other cache, memory is accessed. In this configuration, when the block is sent to the DCE it generates an eviction. In order to make the example more interesting, although the result is the same, block B1 is replaced, invalidating the entry in the L2 (Action 2). Then the block is allocated in the corresponding DCE and sent to the requesting node (Action 3). Since the cache is full, block A1 is spilled to node B and is placed in the invalidated entry.

In the second request node B accesses also the DCE and memory asking for the block (Action 4). When block B5 is allocated in the DCE, it triggers also another replacement. In this case the oldest block of the set in the DCE is evicted (Action 5), this is A1. Finally B5 is sent to cache B and allocated where the invalidated block was.

The right part of the figure shows the final state of caches after requesting blocks A6 and B6 for both configurations. It is clear from the result that in the distributed version cache blocks are closer to the requesting node, improving access latency. We can also see that the distributed version also keeps all the newer blocks in the cache, reducing the number of off-chip accesses. The Distributed Cooperative Caching, however, does not enforce actively a local allocation. In the DCC example cache blocks are closer to the requesting node thanks to the replacement mechanism of the coherence engines, which may have inherently data from all cores. Replacements in the coherence engine entries evict oldest blocks avoiding them to be spilled when evicted, which would force an eviction in the destination cache of a newer block. This effect, however, has a limited impact if

coherence engines have a reasonable number of entries. Distance-Aware Spilling, on the other hand, takes into account distances and enforces closer reallocations.

4.2.3 Differences between CC and DCC

The main differences between these proposals are:

- In the centralized version, tags are just a copy of their corresponding caches while in the distributed version tags are ordered like a big shared cache in the DCEs and store information about the owners. Since tag entries are not restricted to represent only one cache entry, this organization makes a more efficient use of them. Furthermore, the distributed organization does not require to reallocate a tag when a block is spilled or allocated in another cache. It is only necessary to update the tag information.

- The number of tags checked per request in the DCE is equivalent to its associativity -independent of the associativity of the L1s and L2s. In the CCE the number of tags checked is #L1*L1 Associativity + #L2*L2 Associativity, which leads to a reduction of the energy consumption.

- The DCEs implement a LRU replacement policy that favors a broad view of evicted blocks instead of the individual replacement of private caches in the centralized version. This provides a more efficient use of cache entries.

- The size of the DCE is independent of the sizes of the L1s and L2s while in the CCE the number of tags has to be equal to the number of L1 and L2 cache entries. Therefore, the Set Bits selecting the Coherence Engine entry in CC are the same ones that are used in the L1 and L2 caches. Therefore, for 16KB 4-way L1 and 256KB 8-way caches, a different number of bits is going to be used in the CCE for the L1 (6 bits) and the L2 (9 bits) entries. On the other hand, the number of bits used in the DCEs depends on the number of DCE entries, the number of DCEs and their associativity. Therefore, if we use as many DCE entries as L2 cache entries and as many DCEs as nodes (16), we are going to use 4 bits to map the DCEs (CE Bits) and 9 bits to map the different sets (Set Bits). The coherence protocol of the Distributed Cooperative Caching framework also needs to be able to handle DCE replacements.

- The Distributed Cooperative Caching makes use of several coherence engines that can be distributed across the chip. This organization avoids bottlenecks in the on-chip network and can handle requests in parallel. This becomes more important as we increase the number of processors on a chip.

4.3 Power-Efficient Spilling Techniques

As it has been shown, the N-Chance Forwarding mechanism is able to take advantage of the unused cache space with private caches. However, random spilling of all the evicted blocks can introduce unnecessary network traffic by forwarding to far nodes or by forwarding blocks that are not going to be reused. This extra traffic is going to increase the overall power consumption of the memory hierarchy and it will degrade its performance. In the following sections, we describe two techniques to reduce power consumption without degrading performance, the Distance-Aware Spilling and the Selective Spilling.

4.3.1 Distance-Aware Spilling

Although a random selection of the destination node for spilling techniques is a good method to distribute the blocks across the chip, the reuse information of spilled blocks shows interesting optimization opportunities.

Figure 4.6 shows the percentage of spilled blocks reused by the evicting node for the SPEC OMP and SPEC CPU benchmarks. As we can see, for the SPEC OMP benchmark set most of the applications reuse evicted blocks in the same nodes that previously spilled them. In the case of the SPEC CPU, since all applications are single threaded, reuse is completely local. In these type of applications if random spilling is used, data from one CPU in a corner of the chip may end up in the opposite corner and then it is probably going to be reused by the original node. In a 4x4 mesh, this means traversing 6 hops per message. In recent architectures like the Intel Larrabee chip multiprocessor [111] this effect is even exacerbated since a ring topology is implemented. In this case, the maximum number of hops would be 8 for a 16 core configuration. This data transfers are going to increase the network traffic unnecessarily, and thus, the energy consumption and access latency.

Distance-aware spilling techniques aim to reduce the distance between the nodes involved in the spilling. In this case, a set of fixed destinations is assigned to each node. Figure 4.7 shows the simulated mesh structure. The spilling destinations of each node are

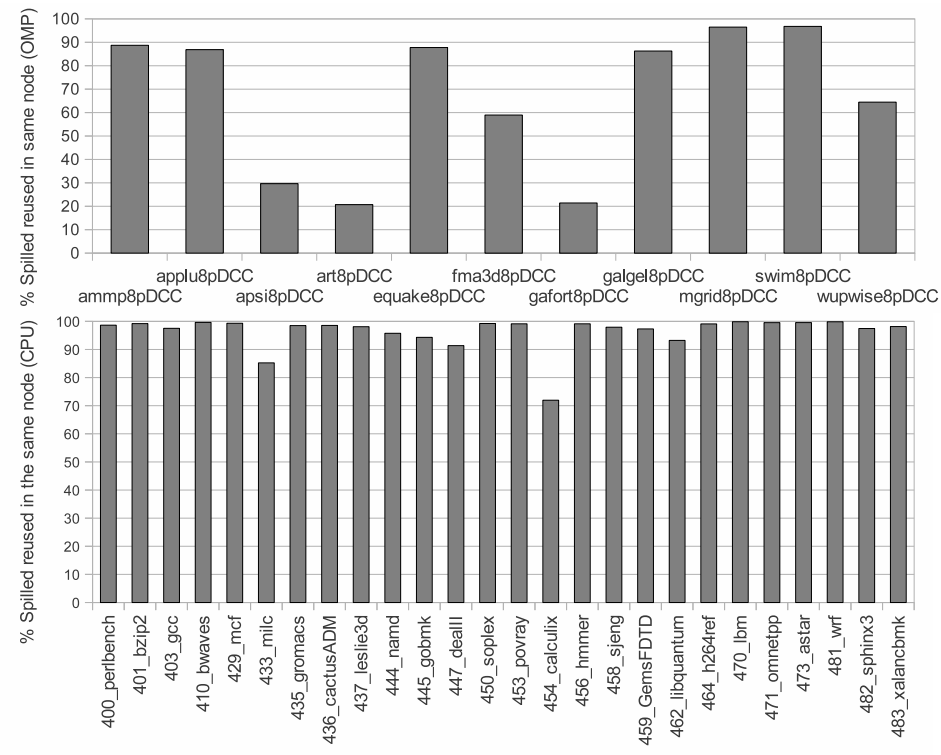


Figure 4.6: Spilled blocks being reused by the evicting node.

represented by arrows. For example, the evicting node (5, in dark grey) has 4 arrows departing from it that indicate the possible destination nodes (1,4,7,13 in light grey) in the 4 nodes configuration. These destinations are selected using a round-robin policy. As we can see, node assignments distribute the spilled blocks uniformly across the chip so every cache receives blocks from the same number of nodes.

Figure 4.8 shows the average number of nodes at each distance for the random and the two Distance-Aware spilling policies. The figure also shows the average distance for these configurations. Due to the topology of the mesh network, spilling to the four closer nodes in the DAS4n configuration would imply that nodes in the center of the mesh would receive spilling messages from more nodes and this would produce a bottleneck. Therefore, we propose the depicted mapping which allows to distribute spilled blocks uniformly. We can see that the average distance is 1 hop when using 2 destinations (DAS2n) and 1.5 hops when using 4 (DAS4n), while for a random destination selection the average distance is 2.7 hops. In addition, Distance-aware distribution policies ensure that all evicting nodes will have the same distance to their destination nodes, while in the random distribution it depends on the position of the evicting node (middle, side or corner).

Figure 4.9 shows the destination assignment for a ring network. In this case, the benefit

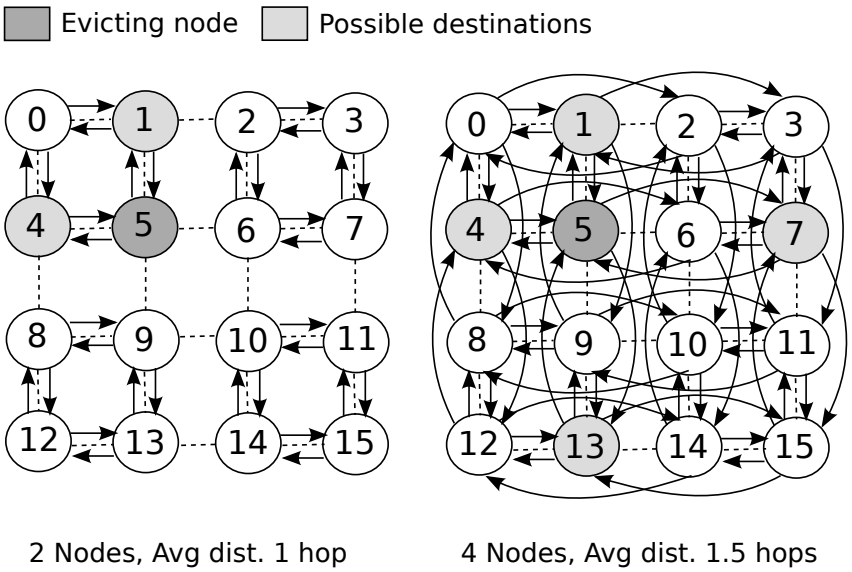


Figure 4.7: Distance-Aware Spilling node assignment in a mesh network.

of Distance-aware spilling is much higher since the average node distance is 4.27 for the random distribution (as shown in Figure 4.10). Distance-aware policies have the same average destination distance as the mesh; 1 hop for the 2 destinations configuration (DAS2n) and 1.5 hops for the 4 destinations configuration (DAS4n).

Hardware requirements of the Distance-Aware spilling are very low, since only a round-robin arbiter per node is required for the 2 or 4 available nodes. This technique, however, while limiting the number of destination nodes also reduces the available cache space. Therefore, it may hurt performance in case of having highly unbalanced memory requirements between threads since those requiring more cache space are not going to be able to spill to all nodes. However, if the memory requirements are more balanced, our technique will be able to reduce the access latency and the network usage.

4.3.2 Selective Spilling

Another interesting optimization opportunity for the spilling mechanism comes from the fact that not all applications are going to benefit of the extra cache space provided by the N-chance forwarding technique. Therefore, it is interesting to have an adaptive mechanism that allows spilling only when blocks are expected to be reused.

Figures 4.11 and 4.12 show the percentage of blocks that are reused after being spilled. While it is interesting to keep the spilling ability for applications like *Art*, other applications like *Gafort* do not make reuse of spilled blocks and they have a high number of evic-

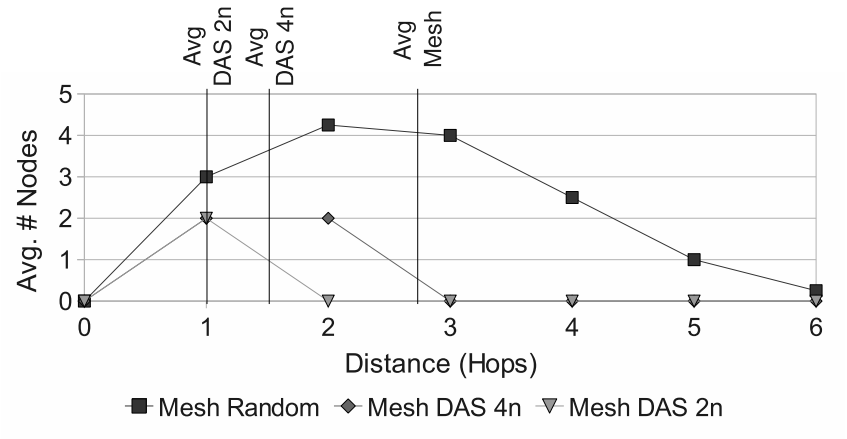


Figure 4.8: Average distance to destination nodes in a mesh network.

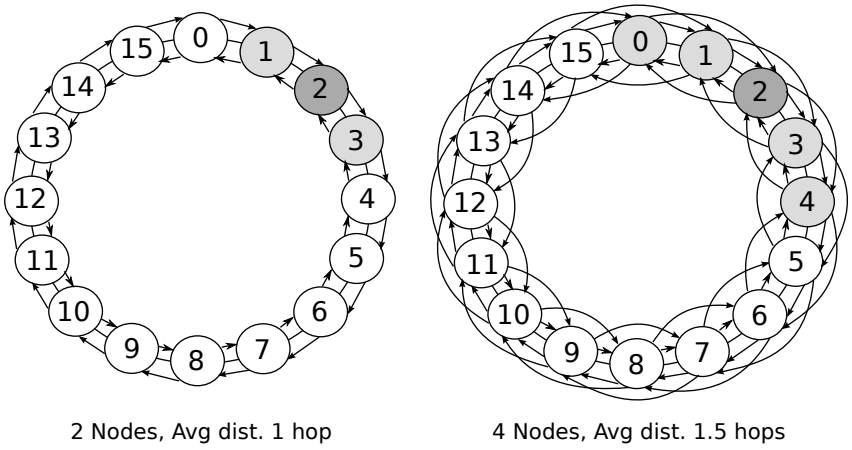


Figure 4.9: Distance-Aware Spilling node assignment in a ring network.

tions/forwardings. These type of applications are going to insert a high amount of unnecessary network traffic and to reallocate blocks that are not going to be reused.

Our Selective Spilling mechanism decides whether to spill or not depending on the reuse of the previously spilled blocks. Our technique spills all the evicted blocks during a period of time C_t where a counter in each node keeps track of all the blocks that are being reused by that node. After this period of time every node decides if it is useful to spill or not. Then, this decision is maintained during 9*C_t cycles. The common spill period is followed by all nodes to be able to detect program phase changes. In our simulations we have used a C_t of 100k cycles and spilling is allowed if 5 blocks have been reused in this period of time. The period of time C_t and the threshold number of blocks have been determined empirically to provide a good performance with a reasonable overhead. The extra hardware required to implement this technique is only one counter of the reused blocks per node. It is possible

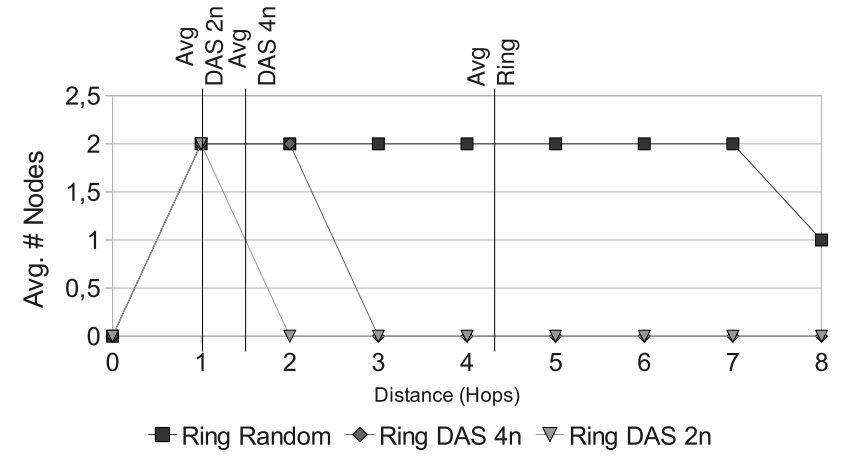


Figure 4.10: Average distance to destination nodes in a ring network.

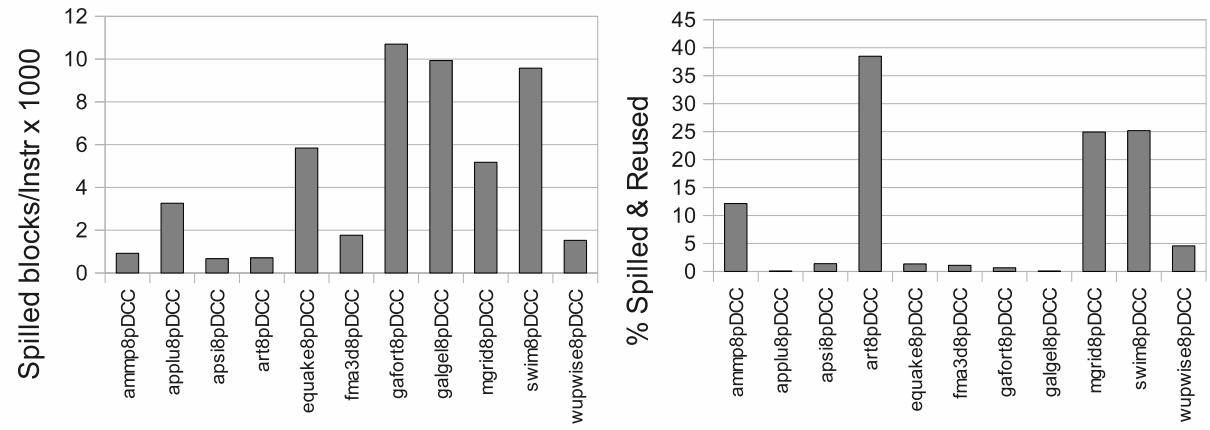


Figure 4.11: Spilling characterization of SpecOMP2001 benchmarks.

to know if a block was spilled or not by the extra bit indicator that all configurations with the N-chance forwarding mechanism already have.

4.4 Evaluation

4.4.1 Simulated Configurations

We have used all the Spec OMP 2001 workload set with the reference input sets for uniprogrammed configurations and ten pairs of these benchmarks for the multiprogrammed configurations. In multiprogrammed configurations each benchmark runs in half of the processors. Threads are allocated together so one half of the die runs the first benchmark and

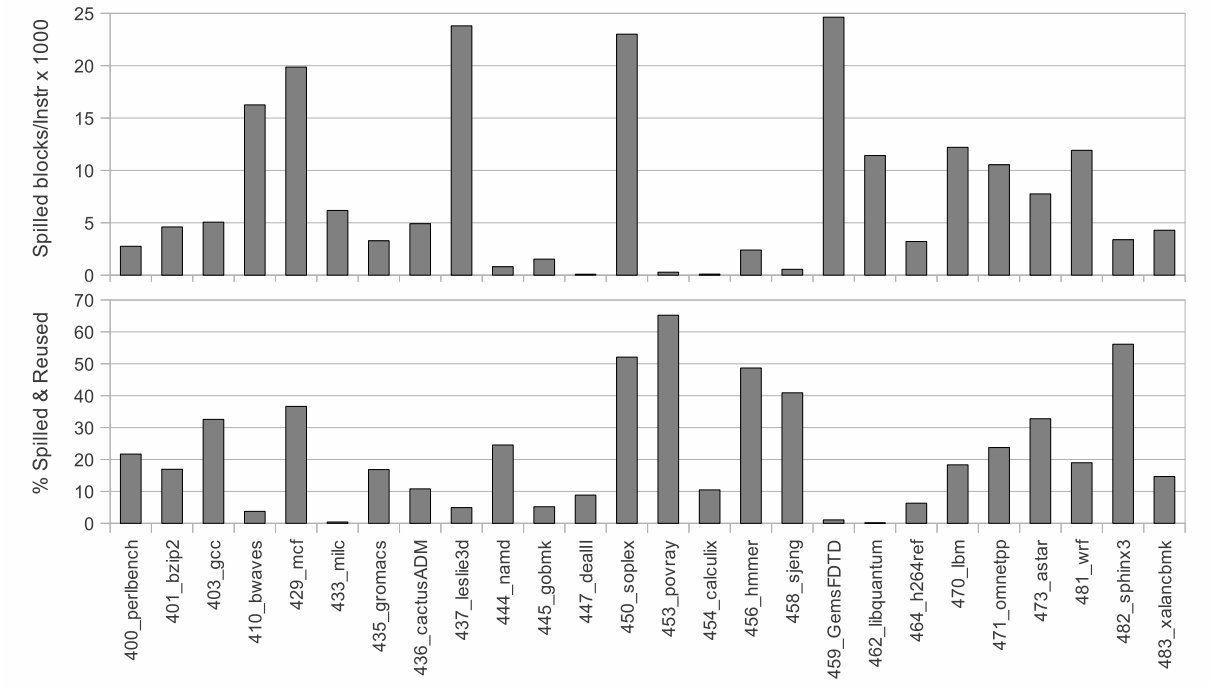


Figure 4.12: Spilling characterization of SpecCPU2006 benchmarks.

the other half the second. Finally we have also studied another set of multiprogrammed benchmarks combining multithreaded Spec OMP 2001 applications with single threaded Spec CPU 2006 applications.

All the Distributed Cooperative Caching configurations have been compared against traditional organizations such as shared or private last level cache and also against the centralized Cooperative Caching. In all the tested configurations, two levels of cache are used; as well as a MOESI protocol to grant coherence between nodes. All simulations use a local and private L1 cache and a shared/private L2 cache for every processor. Evaluated configurations are:

Shared Memory. This configuration assumes a Non-Uniform Cache Access (NUCA) architecture. L2 cache is physically distributed across the nodes and logically unified. Addresses are mapped to cache banks in an interleaved way to try to distribute requests in the network. L1 and L2 caches are inclusive and the L2 also includes the directory information for the allocated entries. On a L1 miss, the L2 bank corresponding to the address is accessed. If the block is located in another L1 in read-only mode, then it is replicated in the requesting node L1. Otherwise, the owner is invalidated without having to access the off-chip directory. This configuration tries to optimize cache usage and reduce off-chip accesses.

Private Memory. In this design, a L2 cache bank is assigned to every processor. On a L2 cache miss, memory must be accessed to check if the block is shared and to retrieve the data. This configuration makes a very small usage of the on-chip network and tries to optimize the access latency by placing all cache blocks in the local L2.

Cooperative Caching. (CC) This configuration evaluates the Cooperative Caching framework. The default Cooperative Caching version uses a Coherence Engine capable of doing 2 transactions per cycle.

Distributed Cooperative Caching. (DCC) The Distributed Cooperative Caching proposal also has been evaluated with two configurations. Both of them use 1 DCE for each node/ processor with 2 R/W ports and 8-way associativity. The default configuration uses as many tags as the L2, requiring 64k entries for a total L2 of 4 MB. The extra cost in bits for each tag in the case of a 16 core CMP with 16 DCEs is one bit per sharer and 4 bits for the DCE state. Therefore, each DCE will have a size of 10 KB. The second configuration uses twice as many entries and is labeled with 2x. This configuration is used to reduce the effect of invalidations and check the degradation in performance they induce.

Distance-Aware Spilling A configuration of the DCC mechanism with distance aware spilling to two neighboring nodes (DCC_DAS2n) and to four neighboring nodes (DCC_DAS4n),

Selective Spilling (DCC_SS5) with C_t of 100k cycles and a spilling threshold of 5 reused blocks.

Power-Efficient Spilling (DAS4n_SS5) A configuration with the Distance-Aware spilling (4n) and the Selective spilling together.

We have fully implemented the cache coherence protocol with the DCE invalidation mechanisms. Invalidation implies two extra states in the DCE state machine and one extra state in the cache state machines. Results shown in the next section already include the extra overhead.

4.4.2 Single Multi-threaded Benchmarks Evaluation

Mesh Network

In this section, we are going to present the evaluation of DCC with different types of spilling compared to traditional memory configurations and the centralized cooperative caching. Figure 4.13 shows the performance, energy efficiency and network usage of the studied configurations normalized over the DCC2x organization and using a mesh interconnection network.

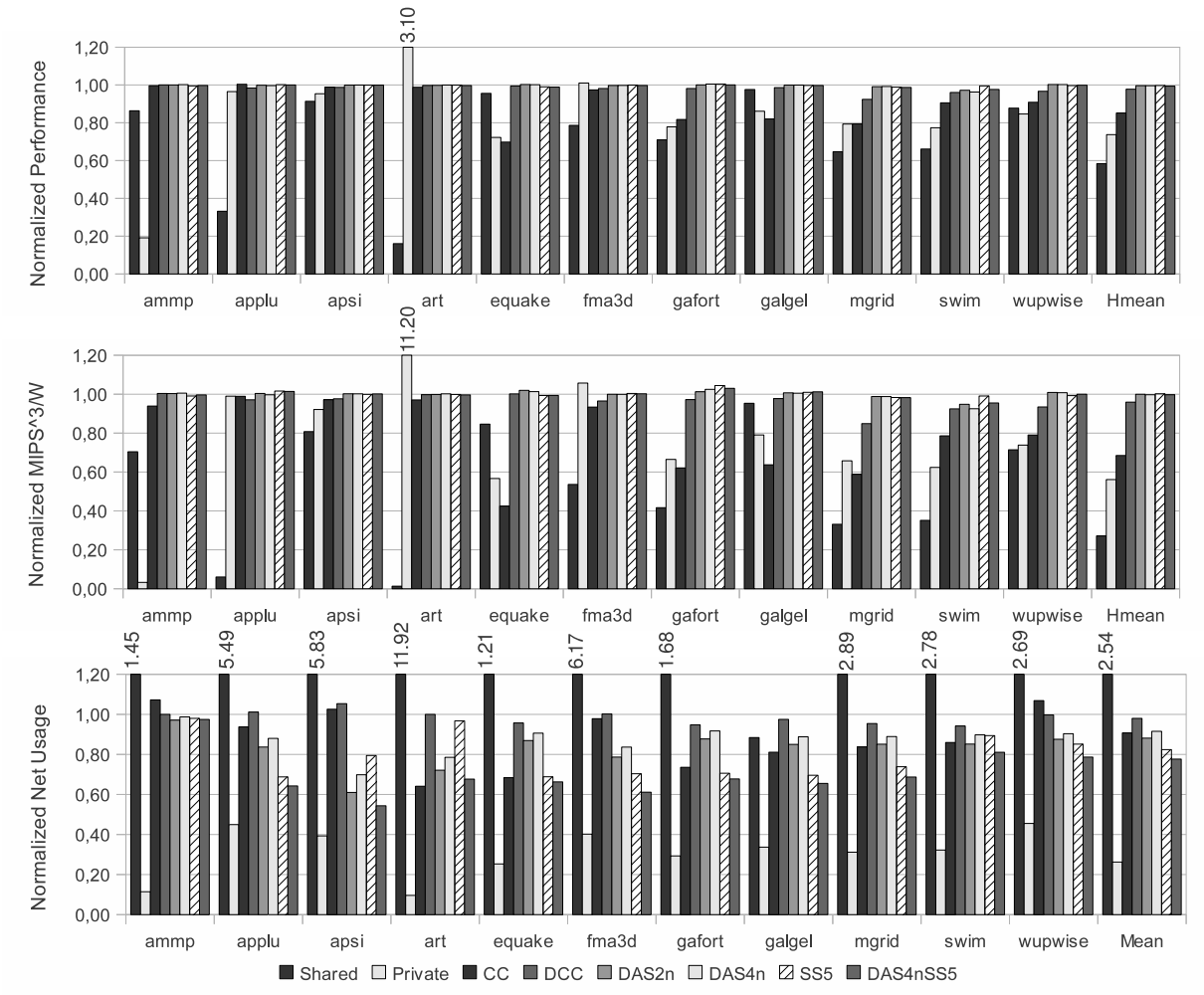


Figure 4.13: Normalized performance, energy efficiency and Network Usage over DCC Random Mesh.

Distributed Cooperative Caching outperforms the other configurations for all benchmarks except for art where private caches provide the best performance. It can be seen that while some benchmarks get the best performance of a big shared cache, others benefit from private caches. DCC is able to get the benefits of both configurations and get an overall improvement of 71% over the shared cache, 36% over private caches and 17% over Cooperative Caching.

The second graph in Figure 4.13 shows the power/performance relation of the evaluated techniques. The performance improvement mixed with a reduced energy consumption increases the benefits of the Distributed Cooperative Caching compared to the other techniques. When compared with Cooperative Caching, the reduction in the number of

compared tags reduces the energy required for each request significantly, giving a 46% improvement in energy efficiency.

Finally, the third graph shows the on-chip network traffic. It can be seen, as expected, that private caches make a very small usage of the interconnection network. This behavior is good both for performance and power consumption but may not be desirable if it also implies an increase in off-chip misses due to a less optimized usage of the cache space. The shared cache on the other hand makes a very intensive usage of the network due to the distribution of data in all the on-chip cache space. In the middle, we can see all the cooperative caching configurations that try to get the benefits of both techniques. Distributed Cooperative Caching with random spilling makes a slightly higher use of the network than Cooperative Caching due to the distribution of Coherence Engines. However, power-efficient spilling mechanisms show to be able to reduce this traffic by 28% while keeping the performance benefits.

Ring Network

In this section we present the results obtained for a network with the same configuration parameters stated in Chapter 3 but changing the network topology to a bidirectional ring (as in the Larrabee chip multiprocessor [111]).

Figure 4.14 shows the performance, energy efficiency and network usage in this network topology. Ring networks have a smaller bandwidth, which penalizes configurations with high network traffic such as the shared cache configuration. It can be seen that in this case private caches outperform the shared cache in all cases except for the ammp benchmark which is known to have a high amount of sharing between threads. DCC configurations also provide the best performance, especially the ones with distance aware spilling.

The second plot of the figure also shows that the Private configuration increases its efficiency for this type of network due to the bandwidth limitations of the ring. Local allocation of data in this network topology becomes critical to reduce access latency. Therefore, this technique is able to outperform in some cases the DCC. Applications with shared data like ammp, however, do not benefit from this organization. DCC, on the other hand, is able to optimize cache usage in these cases and get a good performance for all benchmarks. Therefore, on average it remains as the most efficient solution with a 24% performance improvement and a 38% higher efficiency when compared to the private configuration.

Finally, for this network topology Cooperative Caching is penalized by its centralized tag structure. Network usage (on the third plot) decreases significantly for this configuration

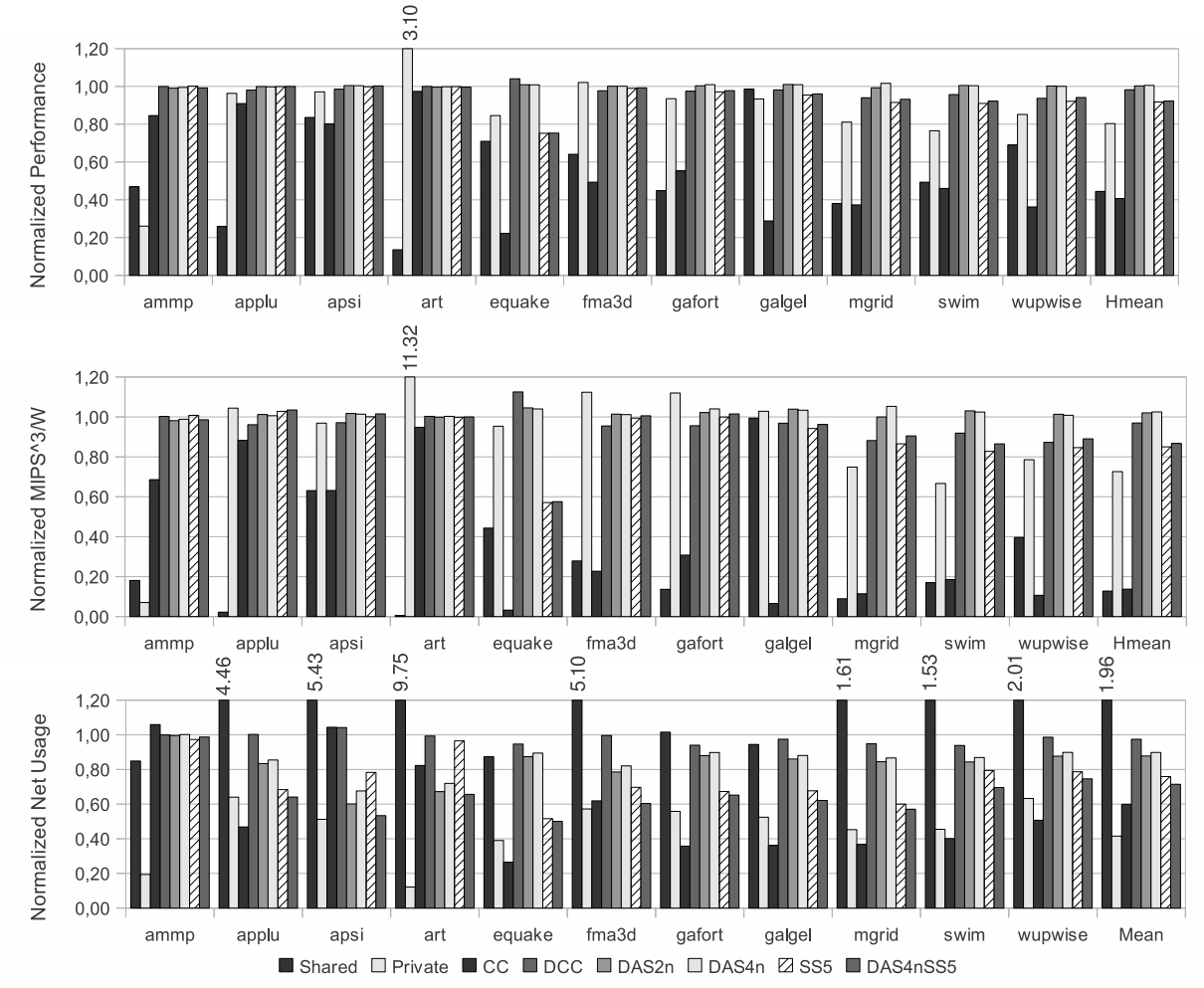


Figure 4.14: Normalized performance, energy efficiency and Network Usage over DCC Random Ring.

due to the bottleneck produced in the Coherence Engine. This translates to a significant performance and energy-efficiency reduction that can be seen in the other two plots.

Two interesting parameters for evaluating the memory hierarchy of chip multiprocessors are shown in Figures 4.15 and 4.16. These are the number of off-chip misses and the average L1 miss latency. Traditional configurations such as shared or private caches only achieve good results in one of them. The best results regarding off-chip misses are obtained by the shared cache configuration while the private configuration shows a better average latency.

On the other hand, hybrid proposals like the Centralized and the Distributed Cooperative Caching improve both parameters by locating cache blocks in the local nodes but also making use of all the on-chip cache space.

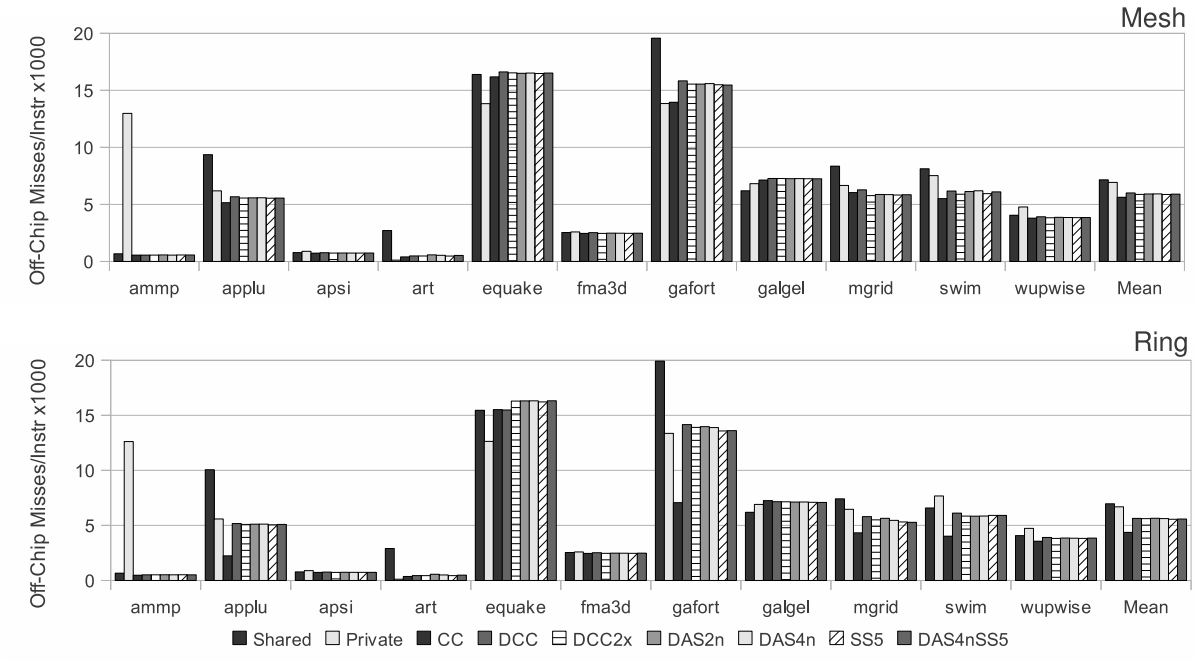


Figure 4.15: Off-Chip Misses per thousand instructions.

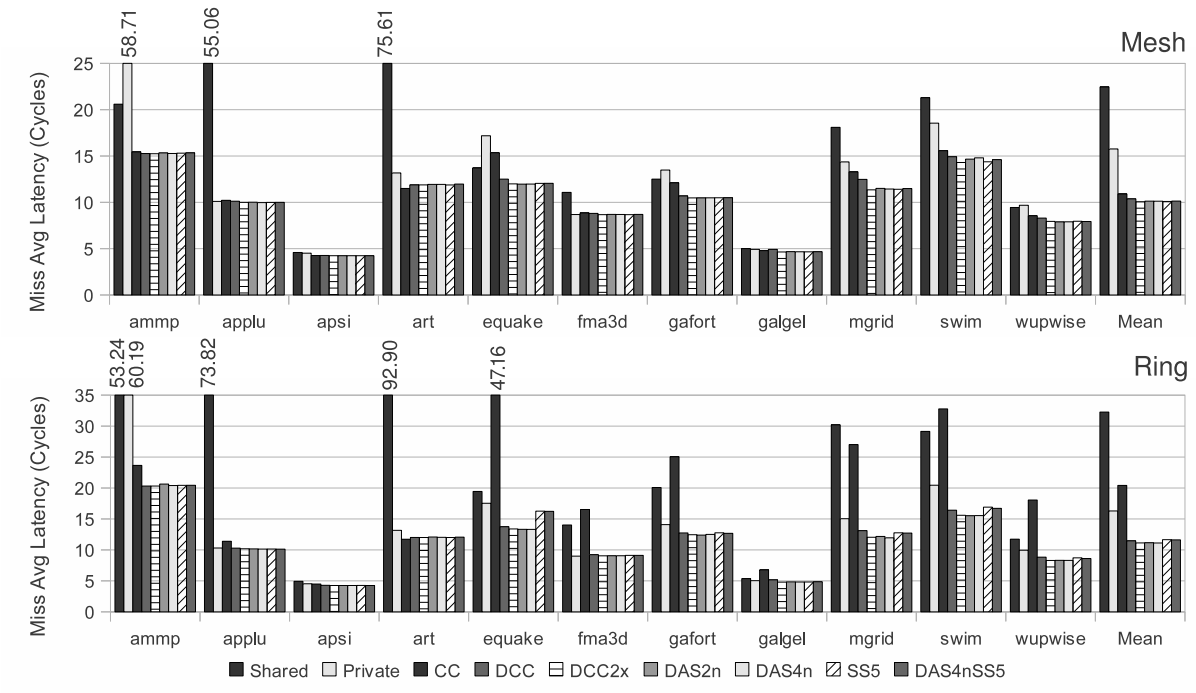


Figure 4.16: Average L1 Miss latency.

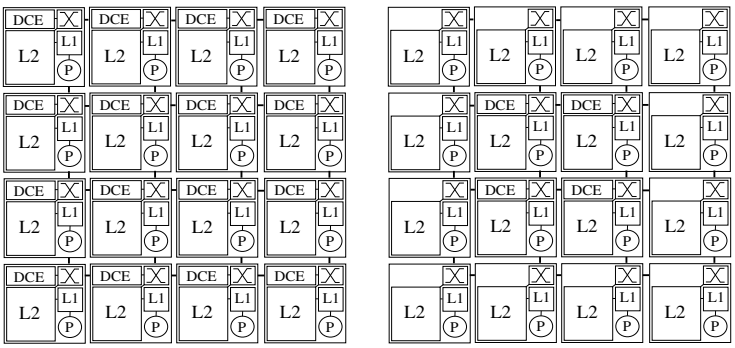


Figure 4.17: DCC16CE and DCC4CE organization.

The Distributed Cooperative Caching has a number of off-chip misses close to the shared cache configuration due to its replacement policy. The shared tag structure of the DCEs invalidates blocks on a replacement through a Least Recently Used (LRU) policy. On the other hand, in the centralized organization, replacement in caches is done independently with the least recently used block of each cache. With the latter, since spilled blocks are usually newer in the local node, they generate evictions of local data, as seen in the example of section 2. The number of off-chip misses, however, remains very similar to the Distributed Cooperative Caching.

The average L1 miss latency, depicted in figure 4.16, shows that CC and DCC achieve the best results. This is because both configurations behave similarly to a private cache configuration and improve it further by adding a sharing mechanism that reduces off-chip accesses. We can also see that the network congestion produced in some benchmarks for the shared configuration leads to a very high miss latency, which explains the degradation in performance.

Sensitivity Studies

We have also conducted a study to exploit the configuration flexibility of the DCE. The behavior of a system with a DCE for each node and 16 processors (DCC16CE) has been compared with a system with 4 DCEs and 16 processors (DCC4CE). Figure 4.17 shows the DCE distribution of both configurations. In the 4 DCE version, the number of tags has been increased so both configurations have the same number of entries. In addition, three different associativities for the DCEs have been evaluated with our framework, 4-way (4A), 8-way (8A) and 16-way (16A) DCEs. Figure 4.18 shows the speedups and power/performance relation of all these configurations over the base DCC configuration.

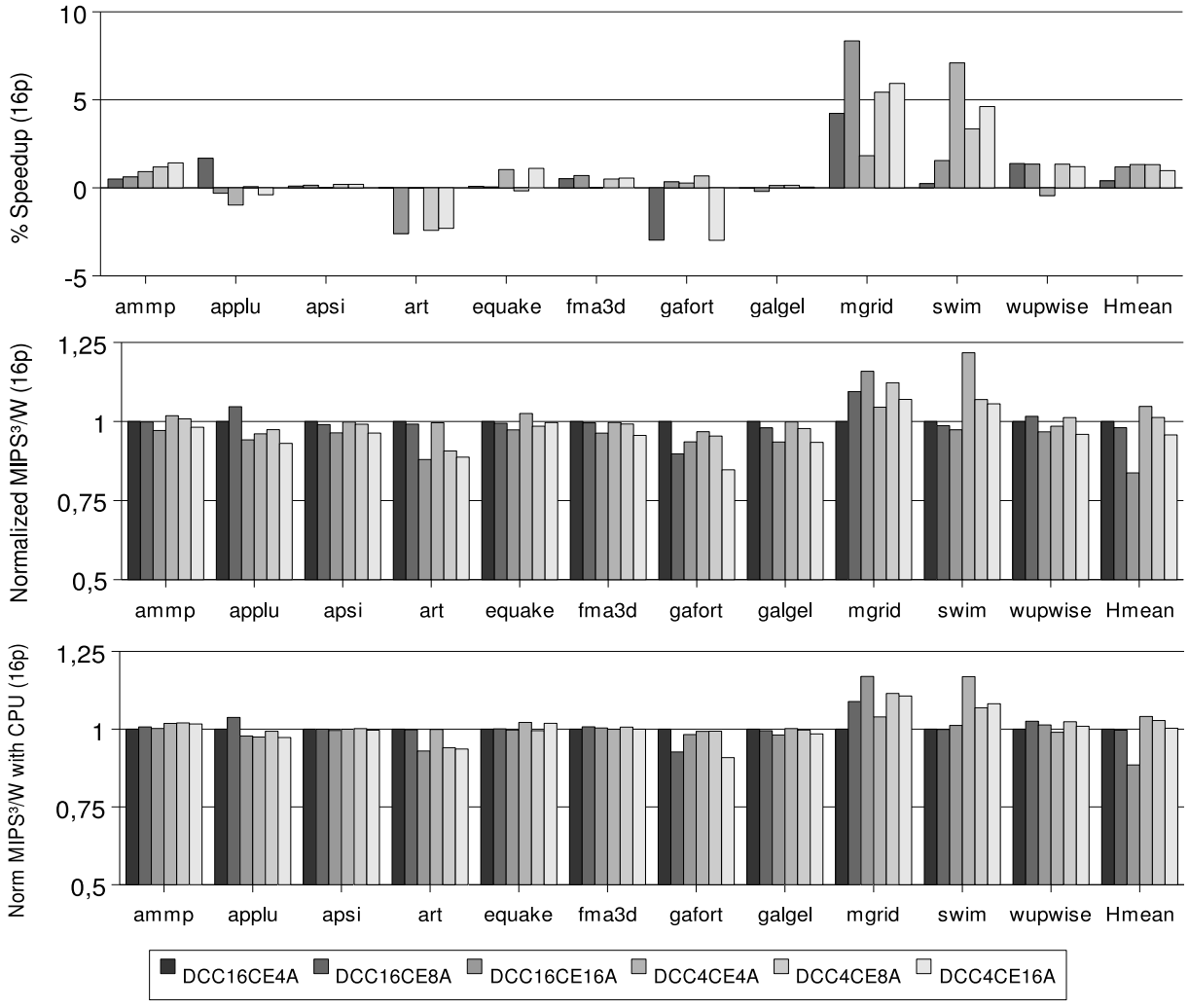


Figure 4.18: DCC Optimal Configuration Study.

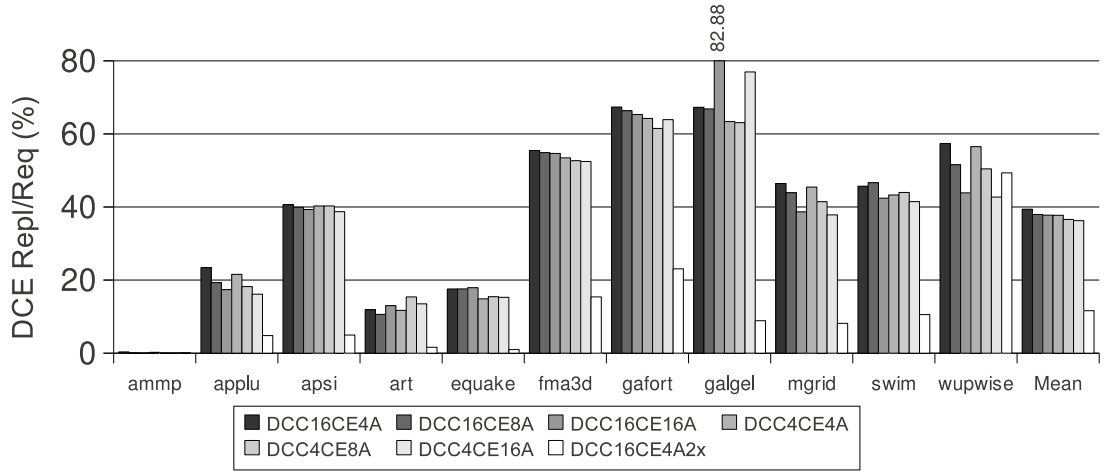


Figure 4.19: DCE replacements per request.

Results show that configurations with higher associativity achieve a slightly better performance. This is because the number of replacements per request is reduced. However, the number of tags compared in the DCEs depends on its associativity. The power/performance relation shows that the speedup obtained for a higher associativity is not enough to compensate the additional power requirements. On the other hand, the usage of less DCEs than nodes increases the power/performance relation by 5%. This improvement is explained by the reduction in the average distance to the DCE. A balanced solution between distance and request distribution across the network needs to be chosen for every case. The DCC scheme, however, provides a flexible framework to find the optimal configuration.

Figure 4.19 shows the percentage of requests that end up in an invalidation of a cache block due to the lack of tags. For the base configuration almost half of the requests to the DCEs end up with an invalidation, except for the ammp benchmark that is very cpu intensive and does not stress the memory system. These invalidations may cause a degradation of the overall performance, so we have evaluated our Distributed Cooperative Caching framework with twice the initial number of entries (labeled 2x in the figure). This configuration may be to expensive in hardware for a real implementation but it allows us to see how much performance is lost. We can see that the number of replacements per request is highly reduced and this is translated in a performance improvement. This improvement, however, is not very high since evicted blocks of the original configuration are always the least recently used from that set.

Finally, in order to evaluate the scalability of Distributed Cooperative Caching against Cooperative Caching we have evaluated both techniques with a varying number of processors. In addition, we have added an additional configuration of CC able to process 4

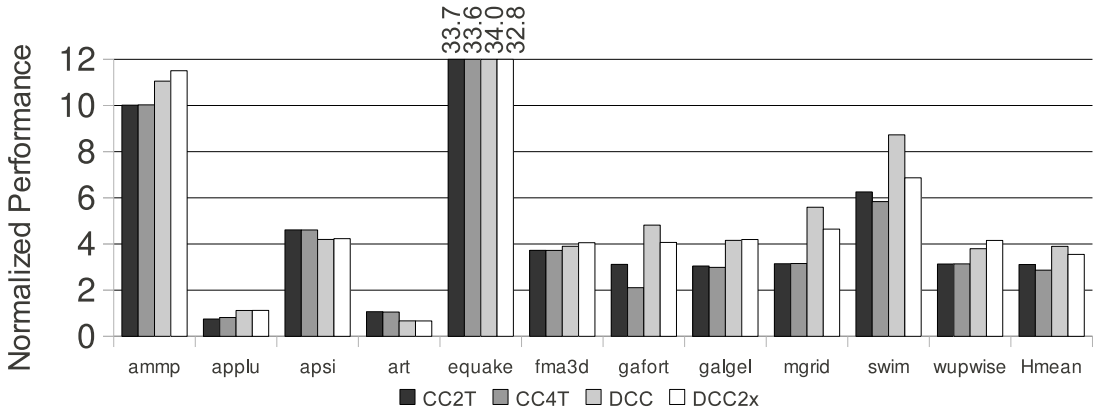


Figure 4.20: DCC scalability. Performance for 32p normalized over 8p.

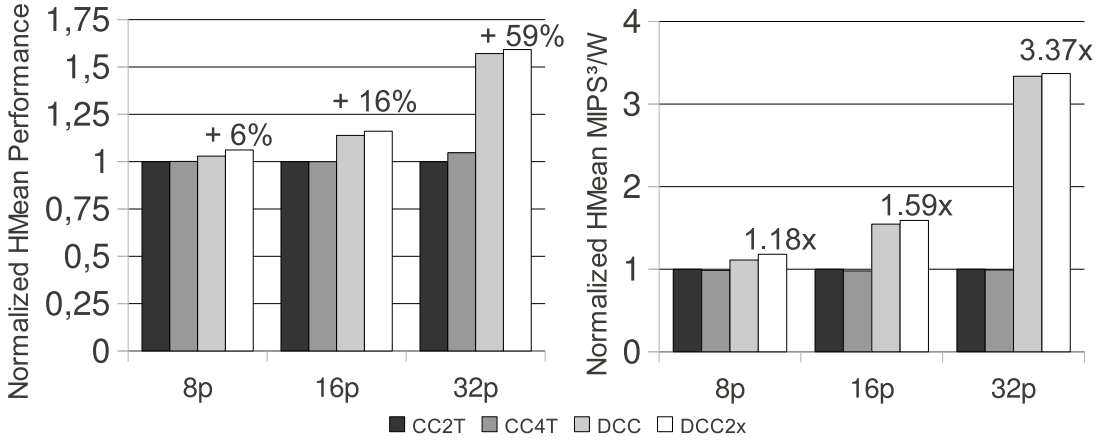


Figure 4.21: DCC scalability. Performance normalized over CC2T.

transactions per cycle (CC4T) for the sake of a fair comparison. We have also evaluated two DCC configurations, the default one (DCC) uses as many tags as the L2, requiring 128k entries for a total L2 of 4 MB. The second configuration (DCC2x) uses twice as many entries. This configuration is used to reduce the effect of invalidations and check the degradation in performance they induce.

Figure 4.20 shows the performance improvement of each configuration with 32 processors and with its results normalized over the 8 processor configuration results. It can be seen that DCC is able to improve performance more significantly for all applications except for apsi and art. The performance improvements shown in this figure, however, can be influenced by the addition of more cache capacity since the overall L2 cache space is increased from 2MB to 8MB (256KB per core). Therefore, improvements are not necessarily due to a better scalability.

On the other hand, Figure 4.21 shows the performance of each configuration for 8, 16

and 32 processors and normalized over the CC2T configuration with the same number of processors. Since results are normalized over a configuration with the same number of processors/cache, we consider this figure more accurate in order to evaluate the scalability of DCC. It can be seen that the behavior of both configurations is similar for a reduced number of cores but when increased, the Distributed Cooperative Caching shows a better response improving performance by 59% over CC. In terms of energy efficiency the difference is even bigger due to the tag structure implemented in the CC that forces to check the tags of all caches. In this case the energy efficiency of DCC is 3.37x better than CC.

4.4.3 Benchmark Set 1 Evaluation

To see the interaction between applications and the possible inter-thread interferences in this section we have evaluated benchmark set 1.

Mesh Network

Figure 4.22 shows the performance, energy efficiency and network activity for the studied multiprogrammed set of benchmarks. The presented energy-aware techniques effectively cut down network usage without degrading performance. Distance Aware Spilling achieves a reduction of 14.2% for the DAS2n and 10.8% for the DAS4n. In the case of Selective Spilling, network usage is further reduced up to 20.8% since the amount of spilling is limited to reused data. The combined solution shows a reduction of 26.6% of the network traffic. All these improvements are achieved while keeping the same performance of the DCC with random spilling (the best performing configuration).

The second plot of Figure 4.22 shows both the benefits of the performance increase and the reduction in network usage (power). Distributed Cooperative Caching also outperforms other configurations by a 22-31% in energy efficiency. The addition of Distance-aware spilling, pushes up the savings an extra 4% (DAS2n). This improvement is achieved although network power consumption is only a small part of the total power. Networks of next generation tiled microarchitectures, however, are expected to have a greater importance in the overall performance and power, leading to greater impact of these techniques.

Ring Network

In ring networks, as it was shown before, the penalty of accessing far nodes is higher. As it will be shown, this limitation is reduced in the Distance-Aware Spilling techniques which

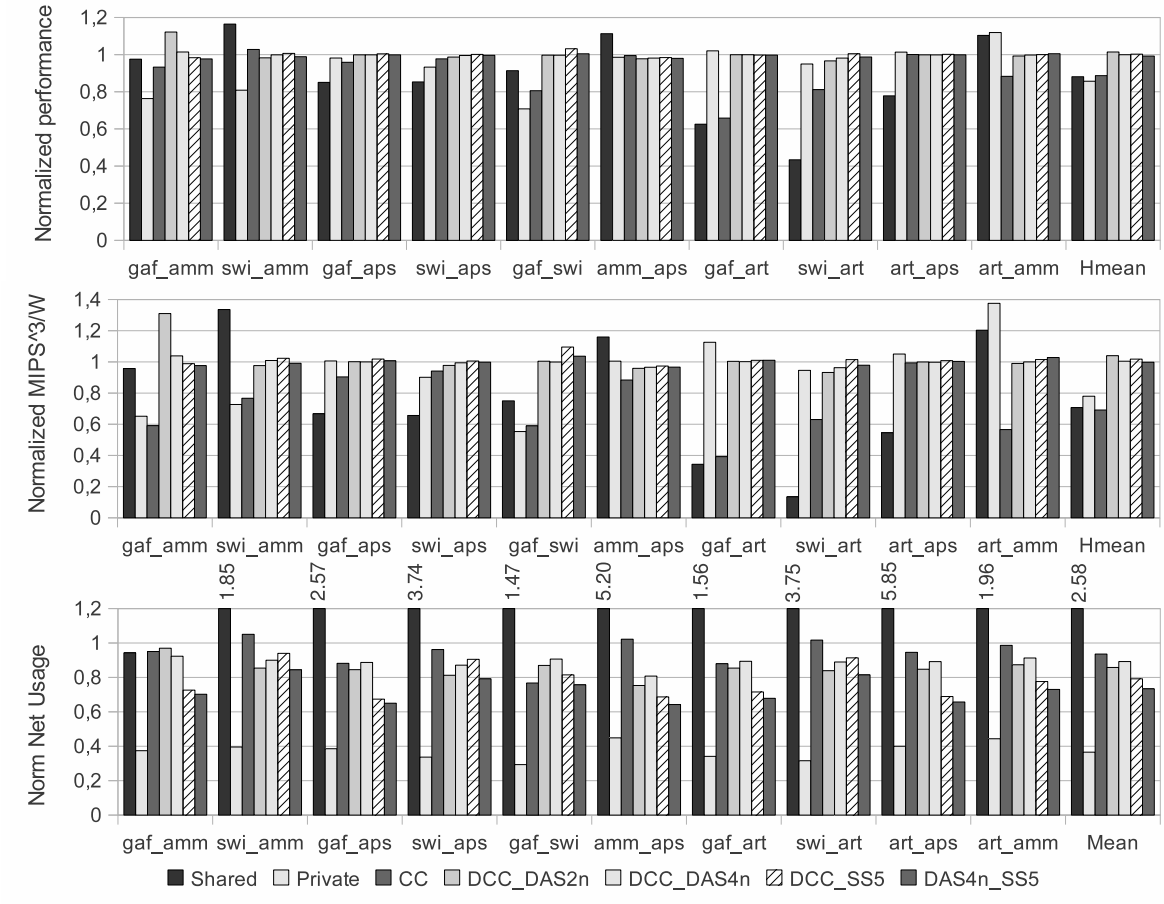


Figure 4.22: Normalized performance, energy efficiency and Network Usage over DCC Random Mesh.

directly translates to better performance. Results are normalized to the same DCC organization running on top of the ring network. Therefore, Distance-Aware spilling is more suitable for this kind of configurations. This can be seen in the first graph of Figure 4.23 where DAS2n achieves a performance improvement of 4.24% and DAS4n of 6.26%. Configurations with the Selective Spilling technique keep the same performance as the random configuration but, as it can be seen in the third graph, reduce the network usage by 21.7%. On the other hand, Distance-Aware (DAS) configurations achieve a reduction of 16.7% for the DAS2n and 14.1% for the DAS4n while increasing the overall performance. If the benefits of both configurations are combined, the reduction of the network traffic is even higher, reaching a 30%.

Finally, the energy efficiency of the evaluated configurations can be seen in the second graph of Figure 4.23. The performance increase and network activity reduction of the Distance-Aware techniques is translated in an improvement of the energy efficiency of a

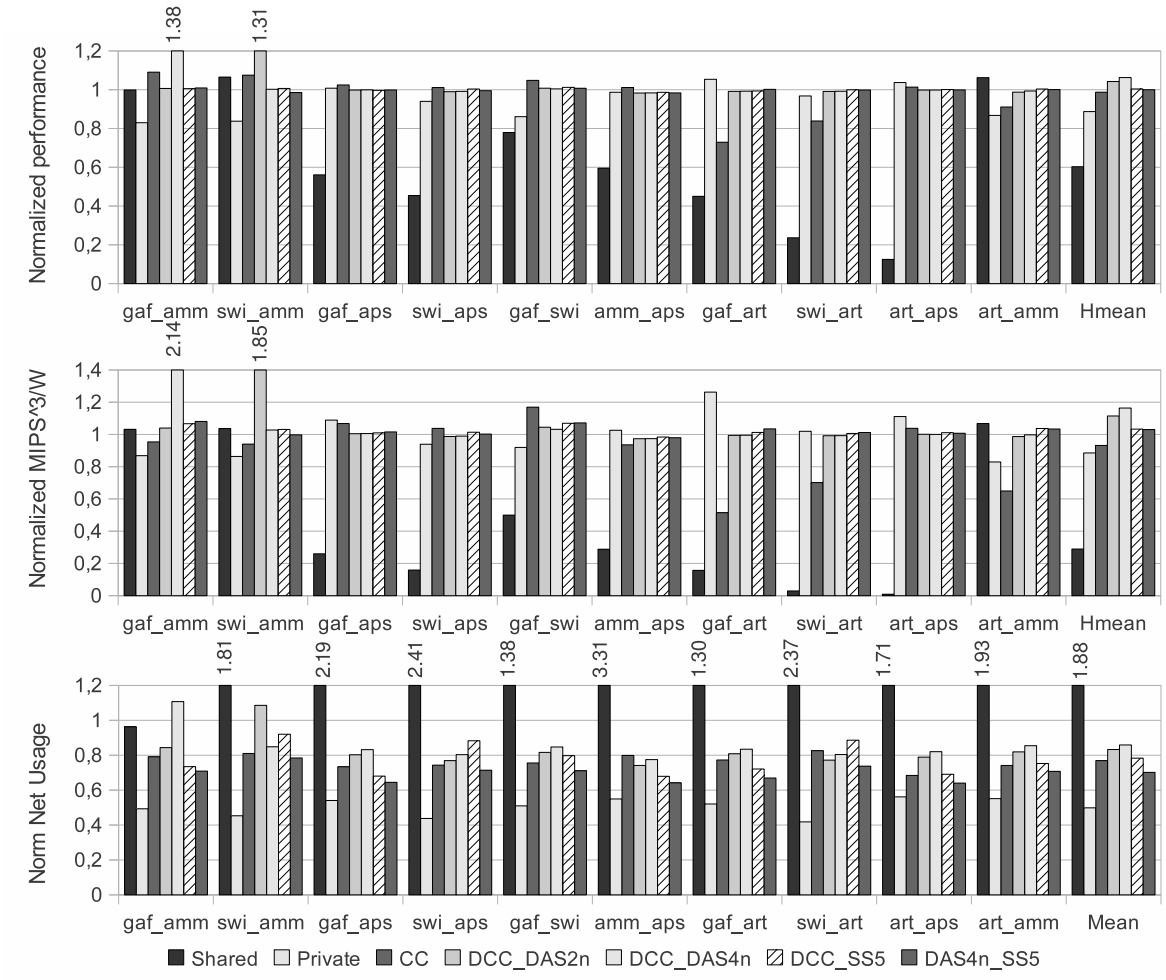


Figure 4.23: Normalized performance, energy efficiency and network usage over DCC Random Ring.

11.5% for the DAS2n and a 16.4% for the DAS4n. Once more, the Selective Spilling configurations, do not show a big variation with respect to the random spilling since the power contribution of the on-chip network in the overall power is not very high. The energy efficiency of the shared cache configuration is very low due to the intensive use of the on-chip network, that in this case is aggravated by the lower capacity of the network (when compared to a mesh). Private caches, on the other hand, show a good energy efficiency due to the low network usage but have a much higher number of off-chip misses that reduce its performance. This would probably increase the overall power had we considered the memory controller energy consumption.

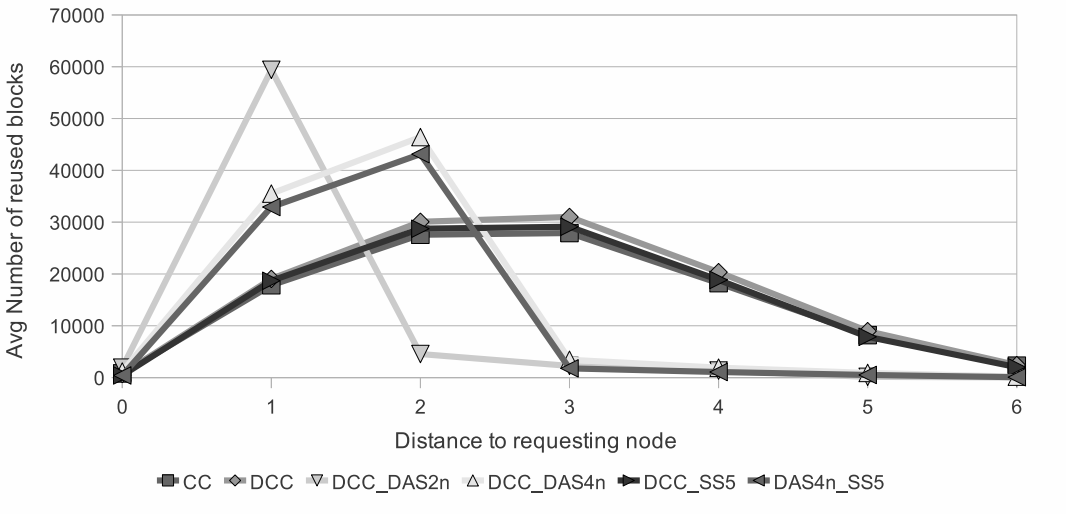


Figure 4.24: Average distance of reused blocks

Spilling Reuse

Figure 4.24 shows the average distance of reused blocks. As expected, the average reuse distance is significantly reduced for distance aware spilling techniques which explains the network usage reduction. This distance, however, is slightly higher than the spilling distance since nodes can reuse data spilled by other processors.

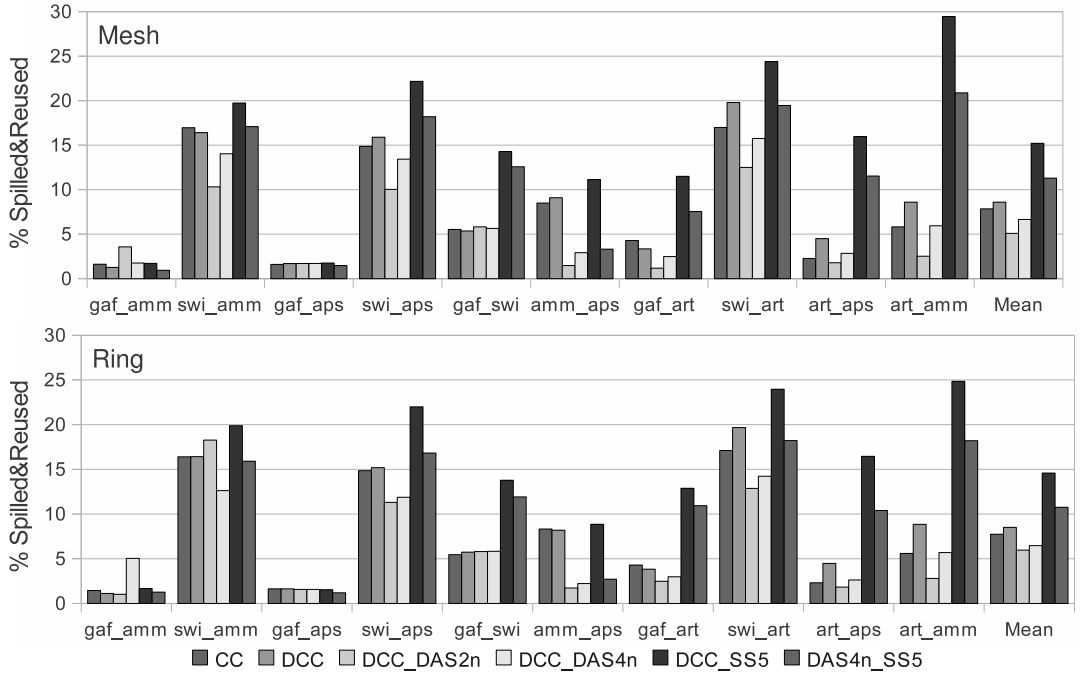


Figure 4.25: Percentage of spilled blocks being reused

Finally, Figure 4.25 shows the percentage of spilled blocks that are reused later. It can be seen that random spilling makes very small reuse since all evicted blocks are spilled. On the other hand, Selective Spilling is able to double the average reuse in both networks by spilling only data from nodes with reuse and achieves up to 29.5% reuse in the mesh for the art_ammp benchmarks. The combined configuration (DAS+SS) also improves the random configuration in both cases while reducing the distance to the destination nodes. DAS techniques reduce the distance to destination nodes in exchange of available cache space. Therefore, spilled blocks are evicted earlier when the cache usage is unbalanced and reuse is reduced.

4.4.4 Benchmark Set 2 Evaluation

In this last section, we can see the behavior of the evaluated techniques with benchmark set 2. To complement the evaluation of DCC in this section we have also modeled the Adaptive Selective Replication (ASR) and compared the performance of both configurations. The evaluated version of ASR also makes use of DCEs to provide coherence in order to have a decentralized structure.

Mesh Network

Figure 4.26 shows the behavior of Distributed Cooperative Caching compared to other techniques. It can be seen that Cooperative Caching behaves like traditional shared and private configurations. This is due to the high usage of the memory hierarchy which saturates the centralized directory (CCE). ASR, on the other hand, has a good performance which in some cases outperforms DCC but in the most memory intensive configurations (those using ammp and 456_Hmmer) does not perform well. This is explained because most of the applications do not share data and, therefore, in such cases private caches are desirable. ASR is implemented as a distributed shared cache which under low utilization all cores can replicate data in the neighboring caches. Under high utilization, however, data cannot be replicated to reduce off-chip misses and, therefore, must be accessed in far caches. This also explains the high network usage of the ASR configuration in the third plot of Figure 4.26. DCC in such cases behaves as a system with private caches with lower latency and lower network usage. This allows to get an average performance improvement of 43% over shared caches, 47% over private caches, 58% over Cooperative Caching and 12% over ASR. The non-sharing behavior of single-threaded applications benefits the local data allocation of DCC. The power-efficient techniques, however, do not improve performance

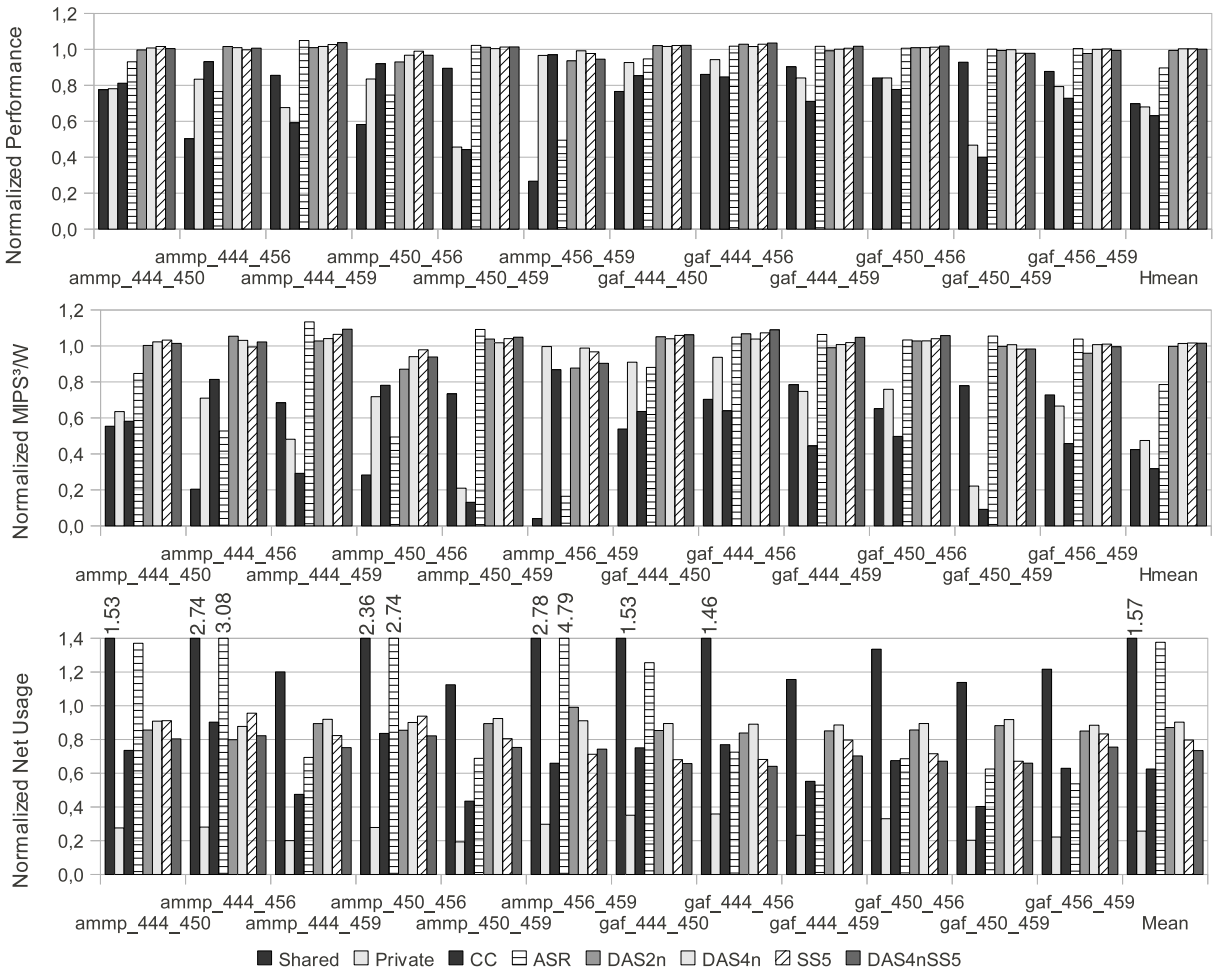


Figure 4.26: Normalized performance, energy efficiency and Network Usage over DCC Random Mesh.

over random spilling, although they are able to reduce network usage by 27%. In this case, the benefits of a reduced distance o spilled blocks are compensated by the reduction in available cache space.

Ring Network

With the ring topology, as with the previous benchmarks evaluations, the more limited network bandwidth increases the impact of using centralized structures or improper allocations. This can be clearly seen in Figure 4.27 in the case of Cooperative Caching which behaves worse than traditional configurations due to the high cost of accessing the CCE. In the case of ASR, its behavior as a distributed shared cache in high usage configurations combined with a limited bandwidth reduces its performance significantly. DCC, thanks to its distributed

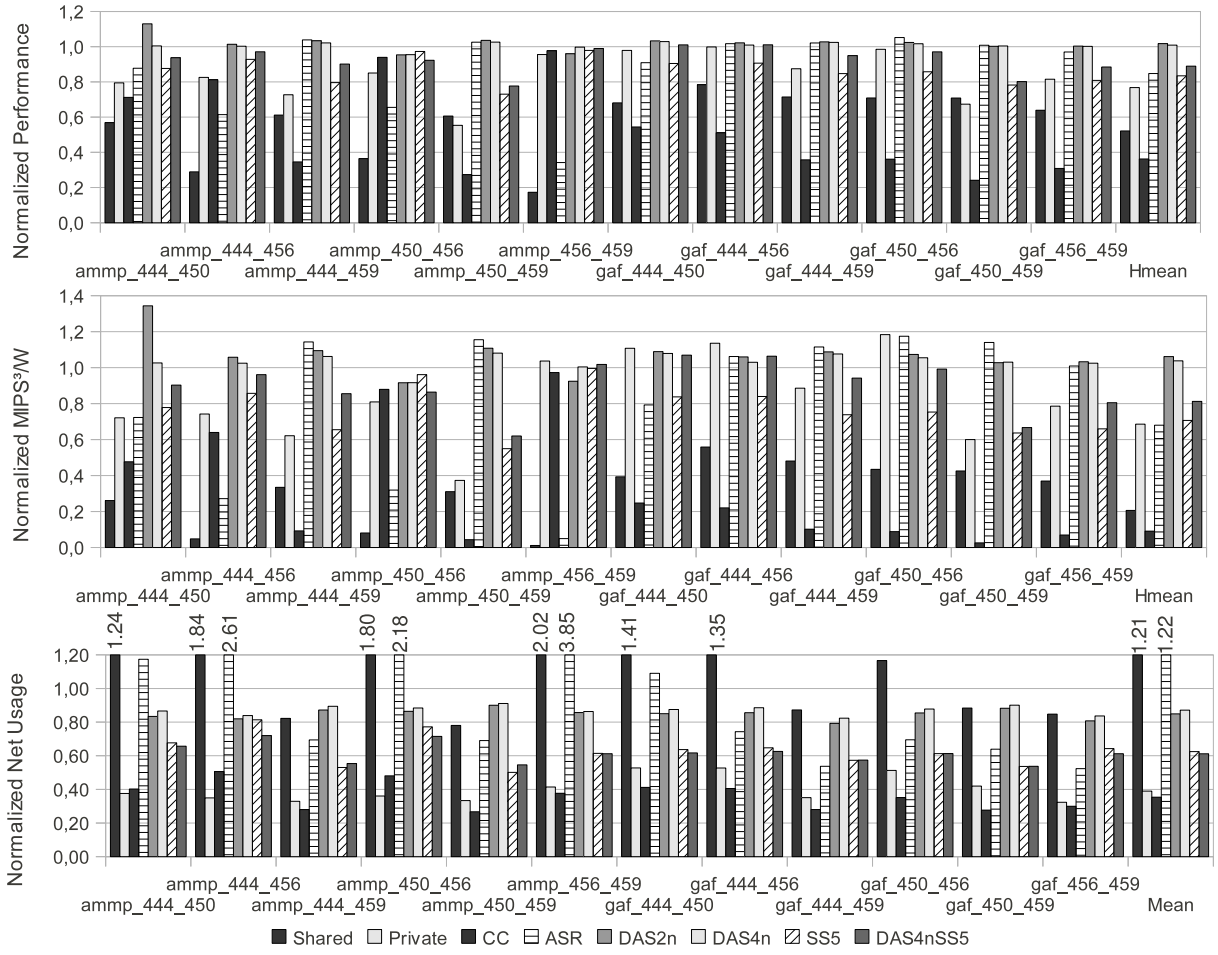


Figure 4.27: Normalized performance, energy efficiency and network usage over DCC Random Ring.

organization, also outperforms existing configurations. In this case, since on-chip network traffic is more limited, Distance-aware spilling techniques outperform random spilling. As in previous configurations, private caches, show a good performance for this network topology but DCC with Distance-aware spilling to 2 nodes outperforms them by 33% and improves energy-efficiency 55%. Selective Spilling, in this case, significantly reduces the amount of network traffic but in exchange of penalizing the overall performance. This behavior shows that in some cases a too aggressive limitation of the spilling ability can harm the overall performance and, therefore, threshold selection must be done carefully. We have kept the same thresholds for all the evaluations in order be able to compare results. For these type of applications, however, it is clear that the threshold value should be reduced in order to benefit from the Selective Spilling.

4.5 Conclusions

In next generation many core architectures on-chip and off-chip communications are going to have a greater influence in the overall performance and energy efficiency. While for a reduced number of nodes traditional configurations like shared or private caches provide the best power/performance relation, in the advent of the many core era it is essential to devise a more efficient solution. We have seen how the Distributed Cooperative Caching framework provides a scalable and energy efficient organization for large multicore architectures using less hardware resources than the centralized version. For a 16-core CMP, DCC with random spilling improves execution time by 71% over the shared cache, 36% over private caches and 17% over Cooperative Caching. These results are even higher when a ring network is used since configurations with centralized structures are limited by the reduced on-chip bandwidth.

We have also shown that the N-chance forwarding mechanism can be improved in terms of power and network usage while retaining its performance advantages for chip multiprocessors. Compared to Random Spilling, Distance-Aware Spilling provides an energy efficiency increase in mesh and, specially, in ring networks (with even a small performance speed-up). In the latter, ED2P ($MIPS^3$ /W) is increased by 16% on average, network usage is reduced by 14% while performance increases 6%. At the same time, the Selective Spilling technique is able to avoid most of the unnecessary spilling in applications with low reuse reducing network traffic by an average of 22%. A combination of both techniques reduces network usage by 30% on average without degrading performance, allowing a 9% increase in energy efficiency. The benefits of these improvements, however, are highly dependent on the interconnection network and application characteristics. Therefore, it must be evaluated if the benefits compensate the additional complexity in each case.

In conclusion, we have seen that Distributed Cooperative Caching is a promising memory organization for chip multiprocessors able to provide cache coherence in a distributed way. Its distributed nature, efficient cache allocation and energy-efficiency make it very suitable for next generation chip multiprocessors with a high number of cores.

Chapter 5

Elastic Cooperative Caching

5.1 Background and Motivation

In the previous Chapter, we have presented a distributed technique to provide coherence in an energy efficient way to caches in a chip multiprocessor. In chip multiprocessor environments, however, due to the limited parallelism of most commercial applications, are expected to run multiple heterogeneous workloads simultaneously. The behavior of each of these applications can be very different and therefore lead to different cache requirements. Recent studies [76] show that cache partitioning has a significant performance impact in runtime execution and that dynamic configurations can adapt to the program's time-varying phase behavior and improve performance.

One example of the optimization opportunities that adaptive configurations can exploit is the simultaneous execution of streaming and cache bound applications. For instance, audio and video streaming are commonly executed simultaneously with text editors, web browsers or antivirus in desktop computers. In general, streaming applications do not take advantage of the upper levels of the memory hierarchy and introduce a high amount of data in the caches that is not going to be reused. This inefficiency is aggravated in shared caches by the eviction of blocks from other applications that could be eventually reused. Therefore, it is interesting to have a memory hierarchy that provides some kind of intelligent control to distribute resources fairly and take advantage of the differences among applications. Ideally, the reallocation of resources should be managed autonomously through hardware arbiters to avoid adding extra complexity to the software layer. In addition, next generation memory hierarchies should provide the elasticity to offer the low latency advantages of private caches and the low off-chip miss rate of shared caches. Future tiled microarchitectures also need scalable structures to allow increased levels of parallelism.

As we have seen in Chapter 2 there is extensive prior research on the memory hierarchy of chip multiprocessors. This work can be divided between static and dynamic resource partition mechanisms.

In static resource partition mechanisms [9, 17, 23, 56, 87, 119, 138] all threads have the same priority and the amount of cache assigned to each thread is changed through the coherence protocol and replacement mechanisms. These organizations either have inter-thread interferences (e.g. Cache-intensive threads may degrade performance of other applications by replacing their blocks) or are not able to give all the cache space to a single thread if the others are not using the cache. This is logical if we consider that the cache space is statically mapped to threads, and can lead to a non-optimal usage of resources in unbalanced workloads.

If we want to optimize cache allocation to reduce off-chip misses, a repartitioning mechanism is desirable to be able to change the cache size assigned to every thread. Dynamic resource partition mechanisms dynamically modify the amount of memory that is assigned to every node and eliminate inter-thread cache conflicts by allocating independent partitions of resources. These resources can be divided in banks, sets or ways and require an arbitration mechanism that can be software or hardware based.

Software-based dynamic configurations delegate resource allocation to the OS. Most of these organizations divide resources in independent sections to be able to provide QoS [18, 38, 54, 81, 100]. These configurations are limited by centralized structures in some cases and by a software-based arbitration that increases programming complexity.

Hardware-based dynamic organizations [30, 50, 104, 115], on the other hand, are able to implement the repartitioning policy in hardware, reducing the programming complexity. They are based on performance counters to measure the benefit of increasing the cache size for each thread. Several recent works [30, 104] adapt the cache size through the column caching technique [21]. Column caching is a cache partitioning mechanism that restricts the available number of sets when allocating a block, therefore enabling the cache to be partitioned. In the case of Utility-Based Cache Partitioning (Utility) [104] a single last-level cache is partitioned and assigned among threads. On the other hand, in the Adaptive Shared/Private NUCA (ASP-NUCA) [30] a cache for each node is divided into shared and private cache space according to thread requirements.

Cache partitioning is controlled by the repartitioning unit, depicted in Figure 5.1 for ASP-NUCA. Repartitioning in ASP-NUCA is based on the expected reduction in misses for each thread. This technique, also used in the Utility-Based Cache Partitioning [104], stores the tags of the evicted blocks on a replacement in the private region. These tags are known as

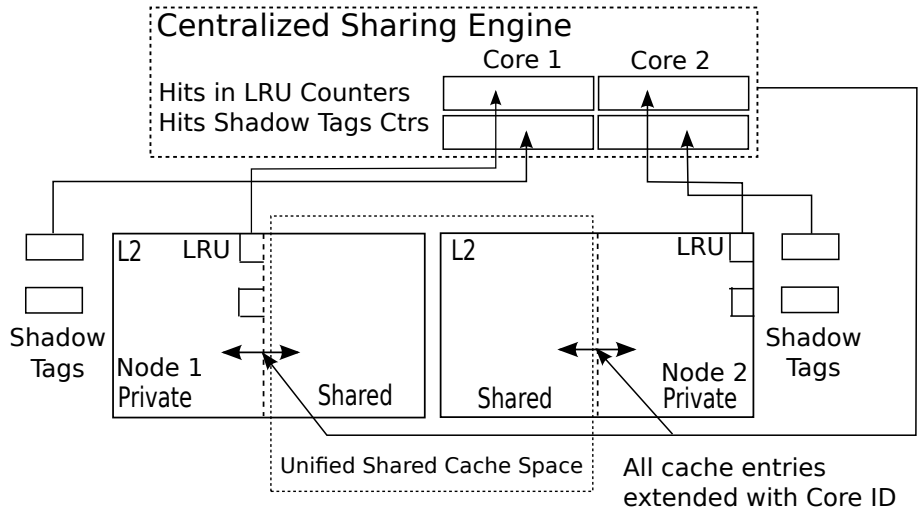


Figure 5.1: Adaptive Shared/Private NUCA Repartitioning Unit.

shadow tags. On a cache miss, the shadow tag is checked to detect the potential benefit of increasing the cache size. On the other hand, on a hit in the LRU block the second counter is updated to detect the potential degradation of reducing the cache size.

ASP-NUCA uses all the shared cache space as a centralized shared cache where all nodes can store as many blocks as indicated by a variable set for each of them. To be able to track the number of blocks from each core, ASP-NUCA also requires a CoreID field for each cache entry. When a block needs to be allocated in the shared region all shared partitions are checked to count the number of allocated blocks for that core. If there are less than allowed, the LRU block from all shared partitions is evicted. If not, the LRU block from that core is evicted.

The usage of shadow tags in all sets implies a significant hardware overhead and, therefore, only a few sets are monitored (6%). Even after doing this reduction the hardware overhead of ASP-NUCA is: a CoreID field in all cache entries, a shadow tag for the 6% of all cache sets, and two counters per node.

In addition to that, ASP-NUCA never changes the amount of private and shared cache space since every time that a private region in one node is increased, it is also decreased in another node. Therefore, this method is not optimal when all nodes execute independent tasks and big private caches would be desirable for each of them or when all threads share the data and a big shared cache would be better.

As we have seen, none of these techniques is suitable for large tiled microarchitectures because they require either a centralized cache or a centralized repartitioning unit that limits the scalability.

5.2 Elastic Cooperative Caching

In this section, we describe the Elastic Cooperative Caching (ElasticCC), a distributed and dynamic memory hierarchy that adapts autonomously to application behavior. We have focused on designing a scalable solution suited for next generation tiled microarchitectures.

To be able to have a scalable memory hierarchy organization it is important that all parts can be distributed to avoid bottlenecks. Therefore, ElasticCC and the evaluated version of Adaptive Selective Replication use Distributed Coherence Engines (DCEs) presented in the previous Chapter to grant coherence. Other coherence enforcement mechanisms could be used if the usage of a distributed structure is not necessary. ElasticCC also uses the spilling technique [27] to reduce off-chip misses. This mechanism, as seen in the previous Chapter, reallocates evicted cache blocks in neighboring caches when the block is the last on-chip copy. Spilling is handled by the DCE since it stores sharers information and also is able to avoid coherence races during the transmission.

To achieve the separation between private and shared cache space we have used the column caching technique proposed by Chiou [21] that allows separation and dynamic repartitioning without having to invalidate any block. Column caching allows software to map specific data to specific regions of the cache. Careful mapping can reduce or eliminate some replacement errors or, as we are going to see, assign cache resources more efficiently in a multiprogrammed environment, resulting in improved performance. The simplest implementation of column caching is derived from a set-associative cache where lower-order bits are used to select a set of cache-lines which are then associatively searched for the desired data. During lookup, a column cache behaves exactly as a standard set-associative cache and thus incurs no performance penalty on a cache hit.

The main difference of column caching with respect to a common set-associative cache is that the replacement algorithm instead of selecting from any cache-line in the set, is restricted to certain columns. Each column is one way of the n-way set-associative cache. A bit vector specifying the permissible set of columns for each type of data is used by the replacement unit. By aggregating columns into partitions, we can provide set-associativity within partitions as well as modify these partitions dynamically. If the cache is repartitioned, data stays in the same cache entry regardless of not being the same partition anymore. However, data still can be found since all cache lines in the set are searched during every access, providing a graceful repartitioning. These entries are only going to be replaced when other blocks are allocated in the new partition through the regular replacement mechanism.

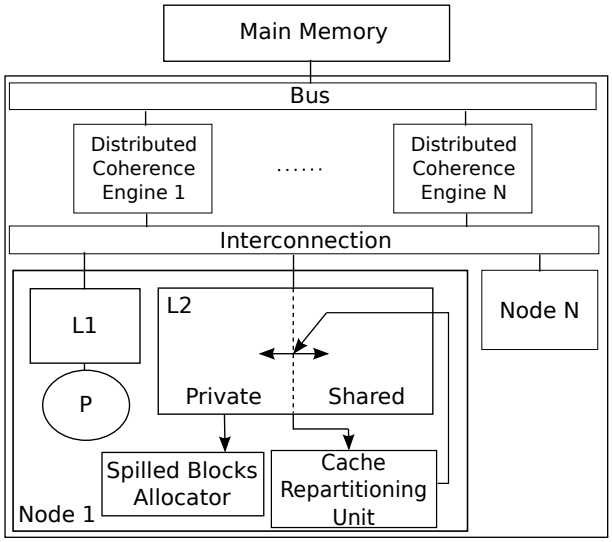


Figure 5.2: ElasticCC Node Structure.

5.2.1 ElasticCC Structure

The ElasticCC framework consists of several independent L2 cache memories that are logically divided into a shared and a private region that compete for the cache space. Private regions store all the evicted blocks from the local L1 and shared regions store spilled blocks from neighboring caches. This allows the creation of big local private caches if all applications have similar cache requirements and a big shared cache if only a few take advantage of extra cache space. ElasticCC also adjusts the level of replication by repartitioning caches. Shared data is replicated when requested in the corresponding private regions but is never replicated in the shared region since it stores only unique blocks. Therefore, bigger private regions allow a higher replication and bigger shared regions limit it.

The extra hardware required for each node is a Repartitioning Unit and a Spilled Block Allocator, described in detail in the next subsections. The Cache Repartitioning Unit is responsible for dynamically adjusting the amount of cache that is going to be private or devoted to spilled blocks. Since not all nodes have the same shared cache space, the Spilled Block Allocator is responsible for deciding to which node a locally evicted block is spilled. This part is also important because more blocks should be spilled to the nodes with more shared space and less to the ones highly used by the local node. Figure 5.2 shows the structure of the proposed configuration.

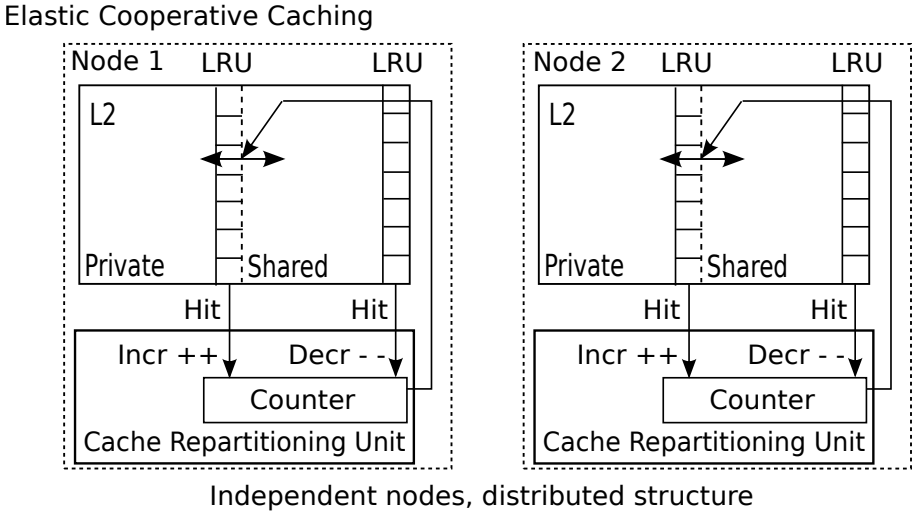


Figure 5.3: ElasticCC Repartitioning Unit.

5.2.2 Cache Repartitioning Unit

Our Cache Repartitioning Unit, depicted in Figure 5.3, adjusts the proportion of private and shared space locally for every L2 cache, avoiding centralized structures that limit the scalability. As can be seen, the hardware overhead of the repartitioning unit is minimal. The Repartitioning Unit only needs one counter per node that is incremented with Private LRU block hits and decremented with Shared LRU block hits.

Our cache repartitioning is done every fixed number of cycles and the decision is based on the number of hits on the Least Recently Used (LRU) blocks of the shared and the private parts of the cache. If the resulting value of the counter exceeds an Upper Threshold (UT) then the size of the private region is increased. If the value does not reach a Lower Threshold (LT) then the size of the shared region is increased. For all the middle values the size is not changed to avoid oscillatory states. Repartitioning is done at a given number of cycles to match program phases behavior and thresholds are control registers set at boot time.

Figure 5.4 shows the cache repartitioning algorithm, where hits in the LRU blocks of the private region increase a counter and hits in the LRU blocks of the shared region decrease it. Because we use LRU hits to decide to change the cache size, we can increase the private size of the cache even when the working set already fits in it. However, this is only going to happen when the shared region is not used since in the other cases the LRU hits in the shared region will compensate the counter value.

As it was shown before cache repartitioning does not require the eviction of all cache

```
If Private_LRU_Hit then
   Increase Counter
EndIf
If Shared_LRU_Hit then
   Decrease Counter
EndIf
If Repartition_Cycle then
   If Counter > Upper_Threshold then
      Add_Private_Way
      Send Repartition_Info_Msg
   ElseIf Counter < Lower_Threshold then
      Add_Shared_Way
      Send Repartition_Info_Msg
   EndIf
   Clear Counter
EndIf
```

Figure 5.4: Cache Repartitioning Algorithm.

blocks of the reassigned way since this would degrade performance unnecessarily. Once a cache is repartitioned partition sizes are updated and this information is used by the replacement mechanism. Therefore, new blocks will gradually replace the ones belonging to the other partition. Private and Shared portions must always have at least 1 way to simplify the coherence protocol and avoid race conditions. Therefore, partitioning only affects the replacement and allocation mechanisms. For accesses, the L2 cache can be seen as a normal shared cache, directly accessed by the local processor or indirectly accessed by other nodes through the DCEs. This organization also means that in-flight coherence messages of blocks being shifted among regions are not going to be affected.

5.2.3 Spilled Block Allocator

The second important part of the Elastic Cooperative Caching is the distribution of spilled blocks across the chip. Due to the dynamic behavior of caches some nodes can be mostly private and without cache space for spilled blocks and other nodes can be completely shared. Therefore it is important to have a Spilled Block Allocator in each node to distribute spilled blocks efficiently.

Every time that a cache is repartitioned, a message is broadcast to all nodes with the partitioning information. This information is later used by the Spilled Block Allocator to distribute data among caches in a more efficient way: sending more evicted blocks to caches with more shared space. Since this mechanism is only used to balance the amount of spilled blocks among nodes, and the maximum size variation on each repartitioning is going to be a single way, we allow the use of stale information when caches are repartitioned.

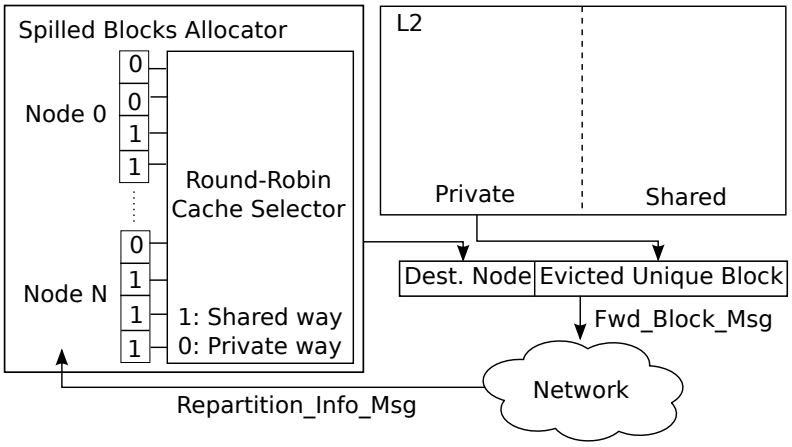


Figure 5.5: Spilled Block Allocator.

Thus, it is possible to separate the repartitioning information from the critical paths and broadcast it in a low priority network channel.

The proposed Spilled Block Allocator depicted in Figure 5.5 uses a Round-Robin arbiter with a bit vector containing the cache partitioning information of each way of each cache. For a 16-core processor with 8-way L2 caches this represents an overhead of 15 bytes which is negligible. However, processors with higher number of cores could just include the neighboring nodes if necessary, as in distance-aware spilling. The bit vector is updated every time that a message with partitioning information is received, with a bit representing the shared or private state of each way. When a block is evicted from the local private partition the arbiter selects the next shared way and spills the block to the corresponding node. Therefore, all shared ways are used equally and in circular order and nodes with more shared space receive more spilled blocks.

5.2.4 Adaptive Spilling mechanism

Elastic Cooperative Caching, as shown in previous sections, distributes available cache space to reduce the number of off-chip misses. Spilling, however, is done regardless of application properties. This may cause interferences between applications in the shared cache space. We propose an extension to the Elastic Cooperative Caching that takes advantage of its fully distributed organization and decides locally (i.e. autonomously) whether spilling is needed or not depending on the type of application.

Table 5.1 shows the most important characteristics of each of the application categories defined in Chapter 3 and the last two columns show the desired behavior of our system. Low

Type	Working set size	Sharing	Local Reuse	Private Cache size	Spilling
Saturating Utility	Small	H/L	H/L	Small	No
Low Utility	Big	Low	Low	Small	No
Shared High Utility	Big	High	H/L	Small	Yes
Private High Utility	Big	Low	High	Big	Yes

Table 5.1: Application Types Behavior

Utility applications, for example, minimize private regions due to their low reuse but still are able to use the shared region. Since this type of applications may have a very high number of misses, the shared cache is going to be filled by blocks that are not going to be reused. The desired behavior in this case would be to forbid spilling. It can be seen that only High Utility applications benefit from the shared cache space. Therefore, our Adaptive Spilling mechanism detects high utility applications and allows them to use the shared cache space while making it unavailable to others.

We will allow spilling, then, for both Private and Shared High Utility applications, but our mechanisms detect each of these applications differently. Private High Utility applications are detected through block reuse. Since applications with high reuse will have a high number of private ways, spilling is allowed when 75% of the cache (6 ways) is private. For Shared High Utility applications, sharing is detected by monitoring cache-to-cache transfers. The spilling decision is done on a per-block basis, allowing us to spill only the truly shared blocks. To track the sharing history of data, one bit is added to each entry of the DCE and is set when there is a cache-to-cache transfer (1= Block was shared, 0= Block was never shared). This bit is later used to detect the corresponding utility type together with the size of the private and shared regions. Our studies have shown that if a block was shared only once it is worth while to spill that block. This information is stored in the DCEs because it provides a global view of the sharing history of the block and avoids the influence of variations in the L2 cache size. The hardware overhead of this improvement is 64 Bytes of extra memory per DCE, which is negligible.

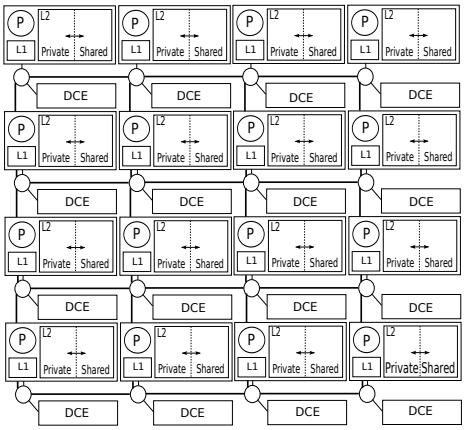


Figure 5.6: ElasticCC Memory Structure.

5.3 Evaluation

5.3.1 Simulated Configurations

The Elastic Cooperative Caching framework has been compared against traditional organizations such as shared or private last level cache. As we have seen, existing cache partitioning techniques rely on a centralized cache or a centralized repartitioning unit with information from all caches that works very well for a small number of nodes. However, for a high number of cores, the scalability of these techniques is very limited. For instance, ASP-NUCA has a max number of blocks in a set for the private and the shared partition. This means that for replacements in the shared region, all shared partitions are considered as a unique cache. Therefore a distributed version of this technique would require snooping all nodes on every cache replacement, potentially saturating the network. Therefore we consider it more fair to compare ElasticCC to scalable state-of-the-art cache organizations like the Adaptive Selective Replication [9] and the previously presented Distributed Cooperative Caching.

All these configurations have been compared through the execution of the first benchmark set presented in Chapter 3. Previously selected benchmarks of the SPECOMP2001 workload set have been simulated with the reference input sets and in pairs, each of them executing 8 threads.

Since our work is intended for large tiled microarchitectures we have used Distributed Coherence Engines (DCEs) presented in the previous Chapter to grant coherence in the ASR technique and in the ElasticCC to avoid centralized directories or snoop based protocols. To the best of the authors knowledge, this is the first time a realistic distributed version

of the ASR technique is implemented and evaluated. Network interconnect also plays an important role in the overall performance [66]. Therefore, we have used a mesh interconnect to have a scalable solution. Figure 5.6 shows the memory structure of ElasticCC with one DCE per node. In all the tested configurations two levels of cache are used, as well as a MOESI protocol to grant coherence between nodes. All simulations use a local and private L1 cache and a shared/private L2 cache for every processor. Evaluated configurations are:

Shared Memory. This configuration assumes a Non-Uniform Cache Access (NUCA) architecture. L2 cache is physically distributed across the nodes and logically unified. Addresses are mapped to cache banks in an interleaved way to try to distribute requests in the network. L1 and L2 caches are inclusive and the L2 also includes the directory information for the allocated entries. On an L1 miss, the L2 bank corresponding to the address is accessed. If the block is located in another L1 in read-only mode, then it is replicated in the requesting node L1. Otherwise, the owner is invalidated without having to access the off-chip directory. This configuration tries to optimize cache usage and reduce off-chip accesses.

Private Memory. In this design, an L2 cache bank is assigned to every processor. On an L2 cache miss, memory must be accessed to check if the block is shared and to retrieve the data. This configuration makes little usage of the on-chip network and tries to optimize the access latency by placing all cache blocks in the local L2.

Distributed Cooperative Caching. (DCC) This configuration, presented in Chapter 4, has been used as a baseline, therefore, results are normalized over it. With this configuration it is possible to see the influence of interferences produced by the usage of a common cache space for private and spilled blocks. It uses 1 DCE for each node/processor with 2 R/W ports and 8-way associativity.

Adaptive Selective Replication. (ASR) [9] We have evaluated a distributed version of the ASR technique where coherence is granted through DCEs. This configuration uses 8 K entry 8 way NLHBs and 512 entry 8 way VTBs for each node.

Elastic Cooperative Caching. (ElasticCC) We have evaluated our dynamic proposal with a Cache Repartitioning Unit updated every 100k cycles and with a high threshold of 5 and a low threshold of 0. Thresholds are determined empirically. Since there is no previous information available on the behavior of applications, caches are initialized with 4 private and 4 shared ways.

ElasticCC + Adaptive Spilling. (ElasticCC+AS) This configuration evaluates the Elastic Cooperative Caching extended with the Adaptive Spilling mechanism, which only allows spilling when the private region occupies 75% of the cache (6 ways) or when the evicted

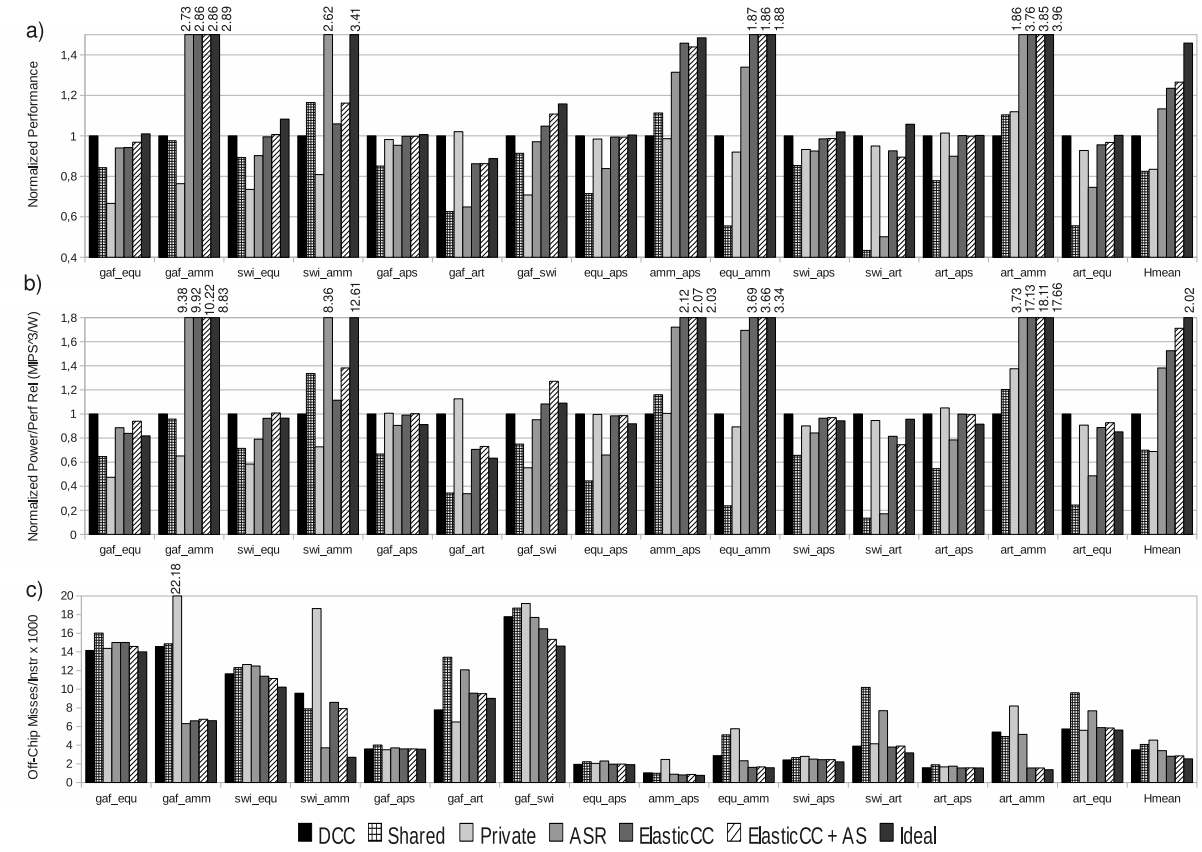


Figure 5.7: Normalized performance, Normalized energy efficiency and Off-Chip misses per Instr.

block was shared. Thresholds for the Repartitioning Unit are the same as the previous configuration.

Ideal. Finally, a configuration that shows how much performance is possible to extract by increasing the cache size is evaluated. This configuration doubles the cache space in each node by using half private half shared 16 way 512kb L2 caches. In this case the shared/private division is static and grants 8 ways to each region. This situation emulates a configuration where there is -at the same moment- the maximum private and maximum shared space (i.e. twice the capacity of the evaluated cache).

5.3.2 Benchmark Set 1 Evaluation

In this section, the Elastic Cooperative Caching is evaluated and compared to existing memory hierarchy configurations. Figure 5.7(a) shows the performance of the studied configurations normalized to Distributed Cooperative Caching, the power/performance relation

that measures the energy efficiency of the proposed solutions, and the number of off-chip misses per instruction.

The performance graph shows that the Elastic Cooperative Caching outperforms the Distributed Cooperative Caching by an average of 27%, by 12% over the distributed version of Adaptive Selective Replication, by 52% over private caches, and by 53% over a distributed shared cache. Performance improvement in dynamic configurations is highly dependent on the characteristics of all the applications being executed simultaneously. Performance improvements can only come from High Utility benchmarks and in the other cases the adaptive mechanism must find the lowest amount of dedicated resources that does not degrade performance.

It can be seen that the Elastic Cooperative Caching is able to improve the performance of High Utility benchmarks when they are able to benefit from the extra cache space of neighboring applications. In some cases, it achieves almost the same performance as the Ideal configuration that has twice the cache space. The only configuration where performance is not similar to the ideal configuration is the one executing two High Utility benchmarks (Swim-Ammp), where ASR gets better performance. In this case both benchmarks benefit from the larger cache space and none of them leaves unused cache space. ElasticCC, however, outperforms ASR in all other cases thanks to its ability to repartition caches.

From the High Utility applications, Ammp is the one that improves its performance in a more significant way. This application is known for sharing a lock variable that leads to increased invalidations and cache misses[5]. Elastic Cooperative Caching is able to keep this data in the caches by avoiding replication of shared blocks and this leads to a high reduction in the number of off-chip cache misses. On the other hand, Swim requires much more cache space to improve its performance. Therefore, in this case, performance improvements depend much more on the other application that is executing simultaneously. However, when executing with a Low Utility benchmark (Gafort-Swim), it is able to increase performance by 10%.

The energy efficiency of Elastic Cooperative Caching is showed in Figure 5.7(b). Results in $MIPS^3/W$ are normalized over the DCC configuration. ElasticCC+AS shows a 71% improvement over DCC and 24% over ASR. The more effective usage of shared regions with Adaptive Spilling can be seen in this graph. ElasticCC+AS improves the energy efficiency by 12% over ElasticCC without Adaptive Spilling. This improvement is produced by more effective spilling that reduces the network traffic and the avoids unnecessary reallocations of blocks without reuse.

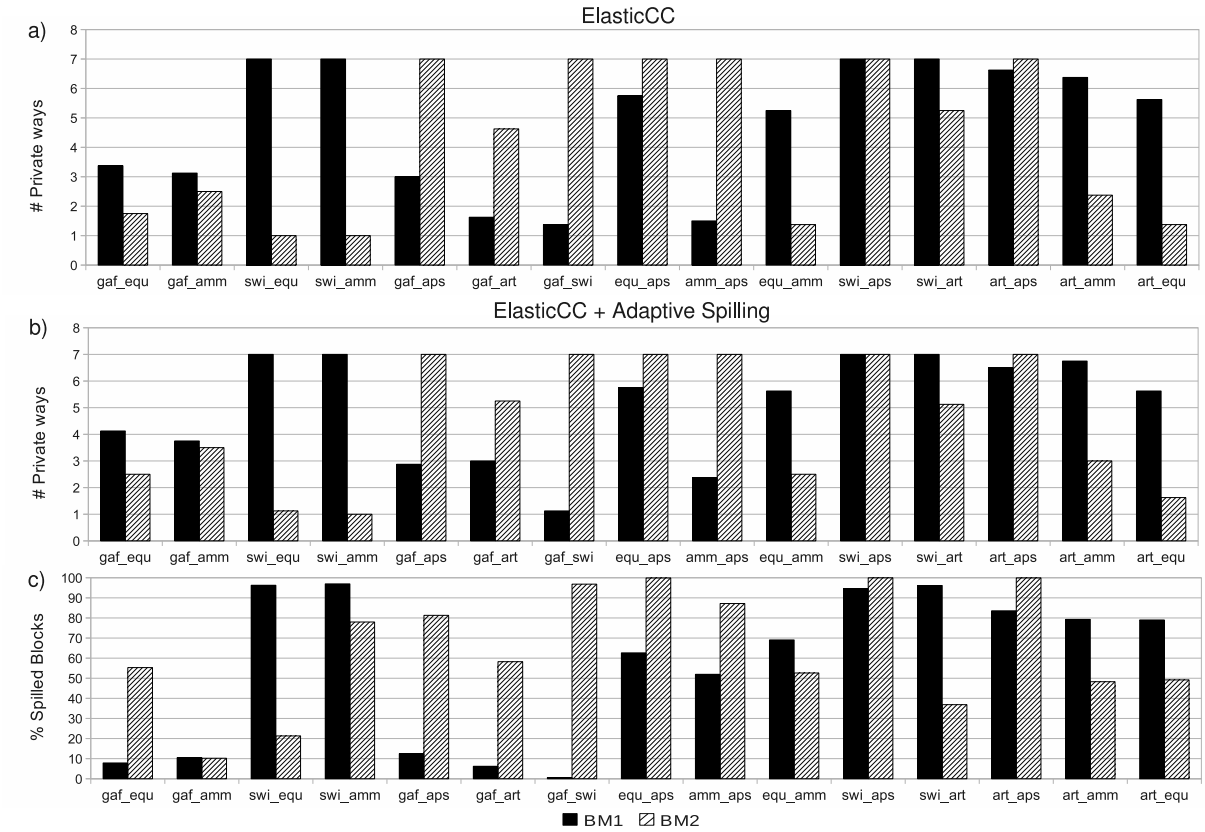


Figure 5.8: Average number of private ways per benchmark in ElasticCC and ElasticCC + AS and percentage of spilled blocks per benchmark in ElasticCC + AS compared to ElasticCC.

Finally, Figure 5.7(c) shows the number of off-chip misses per instruction for each configuration. As expected the more efficient use of caches of ElasticCC brings an average reduction of 18.6% over DCC and 16.4% over ASR.

Dynamic behavior of ElasticCC

To see how the Elastic Cooperative Caching adapts to application behavior, Figure 5.8 shows the average partitioning of the caches through the execution of the applications. Results show that the Repartitioning Unit is able to detect the application behavior and adapt cache sizes accordingly. The Low Utility application (Gafort) is granted in all cases less than 4 ways. The number of private ways assigned depends on the neighbor application. Therefore, when the second application is a Private High Utility one (Gafort-Swim), private ways are even reduced to less than 2. On the other hand, the Private High Utility application (Swim) always uses all the available cache space and ends with 7 private ways. Finally, in the case of the Shared High Utility application (Ammp), it is possible to see that the amount

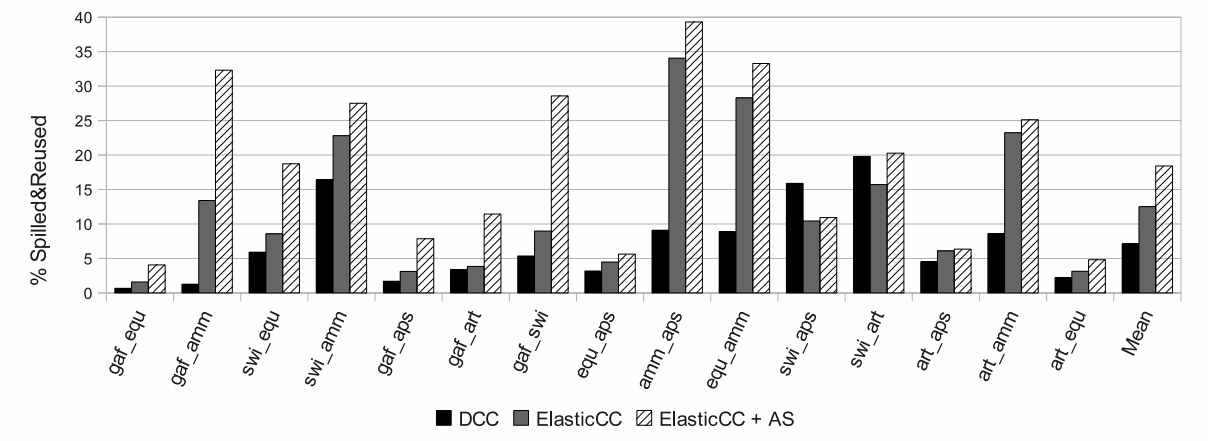


Figure 5.9: Percentage of spilled blocks that are reused in Benchmark Set 1.

of private cache is reduced in order to decrease the amount of replication and keep a bigger number of different cache blocks on-chip.

Although cache repartitioning is done correctly, the usage of the shared space is not optimized since Low Utility applications are going to corrupt it. Therefore a configuration with Adaptive Spilling also has been tested. Figure 5.8(b) shows the average final state for this configuration and also the percentage of evicted blocks that have been spilled to the shared cache. Caches are adapted to application behavior as well as without Adaptive Spilling but in this case private regions are slightly larger. This increase is compensated by the much more restrictive usage of shared regions. It is interesting to observe how the Adaptive Spilling is able to filter data that is not going to be reused. Figure 5.8(c) shows the percentage of spilled blocks. Spilling for the Low Utility application (Gafort) is significantly cut down in all cases compared with the ElasticCC alone where all evicted blocks are spilled. This reduction increases the available cache space for other applications.

The effectiveness of Adaptive Spilling can bee seen in Figure 5.9 where the percentage of spilled blocks that are reused is shown. Elastic Cooperative Caching clearly increases the efficiency of spilling compared to Distributed Cooperative Caching. Reuse is increased on average from 7.1% to 12.5%, avoiding unnecessary reallocations of blocks without reuse. Adaptive Spilling is able to further increase this reuse up to 18.4%. This translates to fewer unnecessary messages across the network and improves the energy efficiency of the proposed configuration.

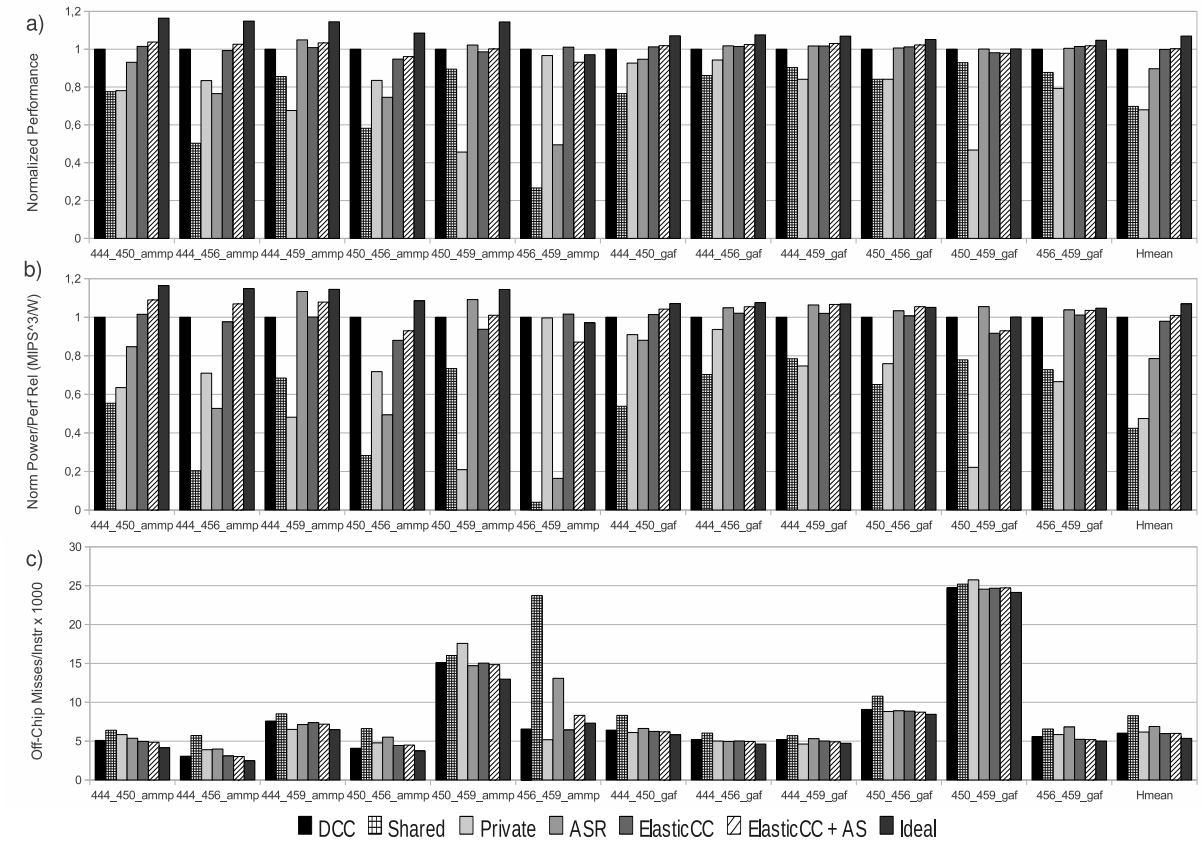


Figure 5.10: Normalized performance, Normalized energy efficiency and Off-Chip misses per Instr.

5.3.3 Benchmark Set 2 Evaluation

The main goal of Elastic Cooperative Caching is to redistribute unused cache resources to minimize off-chip misses. However, it may be the case that all nodes use their resources intensively. In this case we would like our system to behave at least as well as with private caches and minimize the inter-core interference. Therefore, we have also evaluated our technique with the second benchmark set which executes several copies of the SPEC CPU benchmarks. These applications do not share any data and make an intensive use of the memory hierarchy.

Figure 5.10(a) shows the performance of the evaluated combinations. It can be seen that ElasticCC outperforms a private cache configuration in all cases. The ideal configuration shows that performance could be improved by only 7% with a cache of twice the capacity. In such environment, ElasticCC does not have idle resources to redistribute and, therefore, behaves as the Distributed Cooperative Caching.

In terms of energy-efficiency, however, ElasticCC with Adaptive Spilling outperforms the

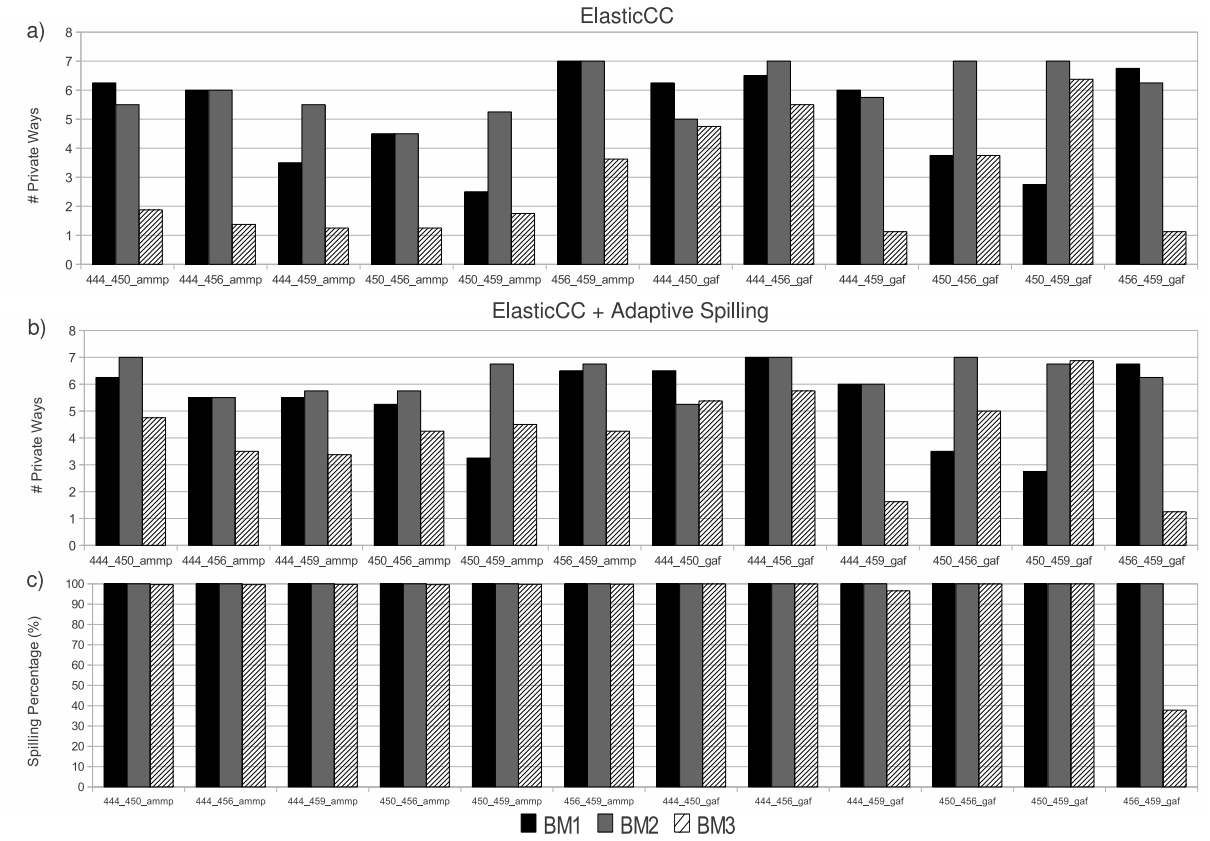


Figure 5.11: Average number of private ways per benchmark in ElasticCC and ElasticCC + AS and percentage of spilled blocks per benchmark in ElasticCC + AS compared to ElasticCC.

Distributed Cooperative Caching due to a more efficient spilling that reduces the number of unnecessary messages and reallocations. Finally, Figure 5.10(c) shows the number of off-chip misses, which is similar for all configurations but for the shared cache. The shared cache configuration does not perform well with this benchmark set. This is especially true in the 456_459_ammp configuration where the interference between applications significantly increases the miss rate.

Dynamic behavior of ElasticCC

Figure 5.11 shows the dynamic behavior with the second benchmark set. In this case the applications 444_namd, 456_hmmer and 459_GemsFDTD are the ones with more local reuse and therefore more cache is granted to them. 450_soplex, on the other hand, has a big working set that does not fit in the local cache and therefore does not take advantage of it.

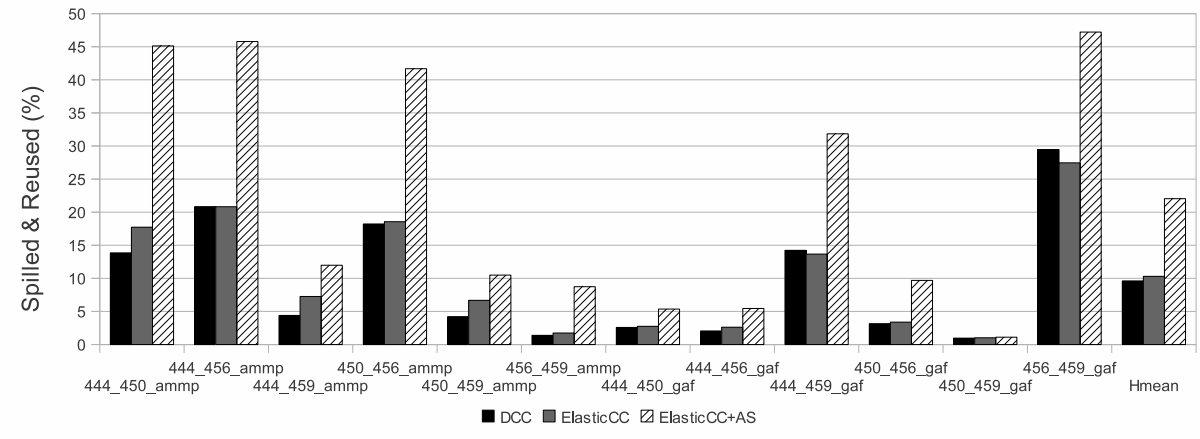


Figure 5.12: Percentage of spilled blocks that are reused in Benchmark Set 2.

Finally, Figure 5.12 shows the percentage of spilled blocks being reused for the configurations with spilling. It can be seen that also in this case ElasticCC+AS is able to increase significantly the utility of reallocated blocks, more than doubling the reuse of spilled blocks compared to DCC. This efficient usage of spilling reduces the number of unnecessary reallocations and explains the better energy efficiency of ElasticCC when compared with DCC.

5.3.4 Temporal behavior of ElasticCC

In order to evaluate the temporal behavior of ElasticCC and its adaptivity to execution phases, detailed statistics have been collected during 80 million cycles. Figure 5.13 shows the temporal behavior of the elastic Shared/private cache of the node executing thread 1 of Equake for the Gafort-Equake combination. The plot shows the miss rate of the private region for a fixed version with half private-half shared cache and for ElasticCC. When the local thread makes no reuse at all, the shared cache space is increased so other nodes can take advantage. On the other hand, when reuse is high, the cache behaves as private to reduce cache misses. An finally, when reuse is small or medium, the behavior depends on other threads requirements. Since Gafort is an application with low reuse, very few blocks are going to be reallocated in the shared space and the cache is kept mostly private. In addition, Figure 5.13 shows that when reuse is small the cache is more likely to repartition than when we have a medium reuse. It can be seen that ElasticCC is not only capable of detecting differences among applications but also variations within the same application, adjusting cache size accordingly.

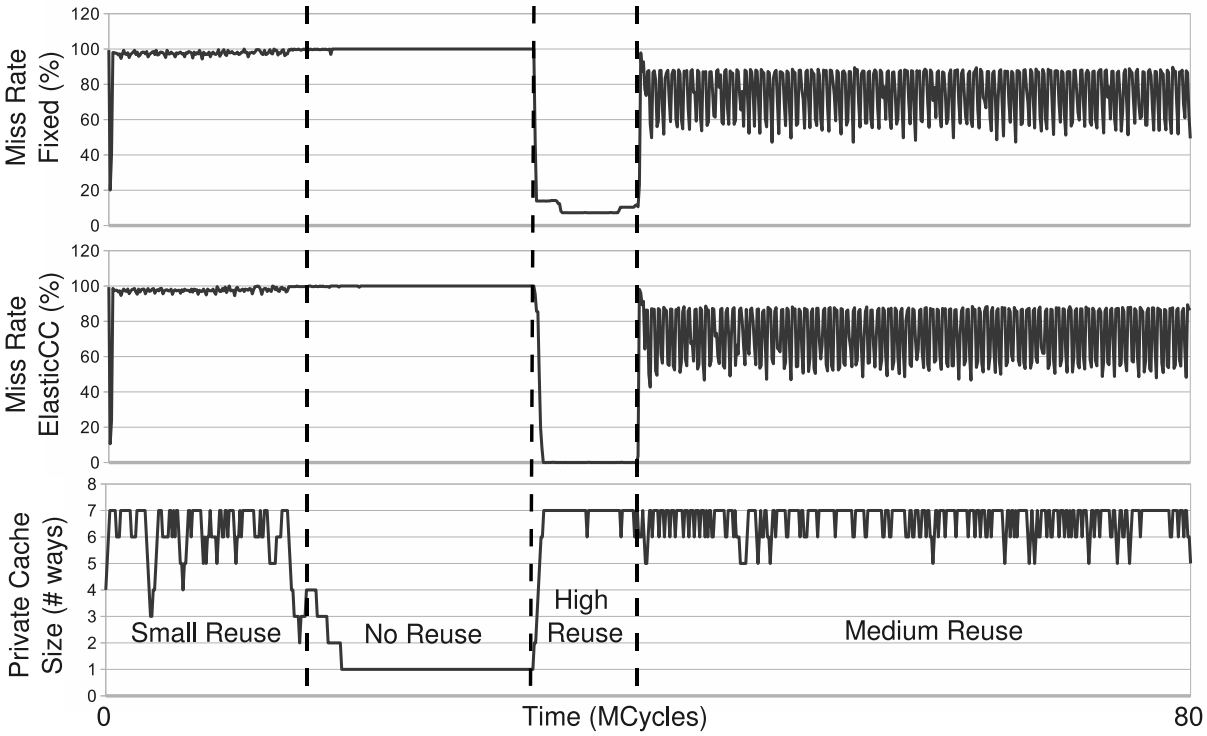


Figure 5.13: Cache behavior for thread 1 of Equake.

5.4 Conclusions

A dynamic and scalable memory hierarchy is necessary for next generation tiled microarchitectures. Elastic Cooperative Caching is shown to be a good candidate due to its autonomy, distributed organization and adaptivity to applications. Its repartitioning unit is the first to allow independent resource allocation based on local information, making it suitable for tiles with a high number of cores. The proposed scheme is able to detect the different cache requirements among applications and distribute cache resources accordingly without software support. Furthermore, the sensitivity of the repartitioning unit is able to detect changes in the execution phase of an application and adapt the system to the new situation. ElasticCC achieves a speedup of 27% over Distributed Cooperative Caching, 12% over Adaptive Selective Replication, 52% over private caches, and 53% over a distributed shared cache. More effective cache allocation is responsible for this improvement since it leads to a 19% reduction of off-chip misses compared to DCC and 16% compared to ASR. Furthermore, the dynamic management of cache resources avoids the energy overhead of reallocating not-reused cache blocks and consequently increases energy efficiency by 71% over the DCC configuration and 24% over the ASR configuration.

Chapter 6

Thread Row Buffers

6.1 Background and Motivation

In the previous chapters, new organizations for the on-chip cache memories have been proposed to optimize its performance and energy-efficiency in chip multiprocessors. Cache memories, however, have a limited size due to chip area constraints and applications eventually need to access DRAM memories or disks to retrieve data. Therefore, memory access is also an important topic that we have studied in this thesis.

During the last decade memories have greatly evolved in terms of capacity and integration but still remain one of the main limiting factors of current processor performance due to its long access latency and bandwidth limitations. This problem has been exacerbated with the introduction of chip multiprocessors, which require much larger amounts of data and have different access patterns. Such changes suggest that the memory hierarchy must be adapted to deal with the new requirements.

With the expected increase in multiprogrammed environments comes a loss in memory locality, resulting in a reduction in the row hit rate of memories. Due to the large size of memory arrays, memories use row buffers (typically of 8kB) which store a whole page to allow faster reads and writes. This buffer needs to be updated every time a different row is read or written, consuming time and energy. Therefore, it is critical that memory systems make as much use as possible of row locality to both increase performance and reduce energy consumption.

As it has been shown in chapter 2, a First-Ready First-Come-First-Serve (FR-FCFS) policy [108, 142] is usually implemented in the memory controller of uniprocessors since it reaches reasonable hit rates with a simple reordering mechanism. The execution of several simultaneous applications, however, has led to different memory access patterns which

often alternate between a limited number of rows. This behavior significantly reduces the row hit rate for traditional configurations, and therefore, reduces overall performance.

On the other hand, the usage of a shared resource like DRAM memory by different threads makes it necessary for the system to provide some kind of fairness or performance isolation control. Existing solutions [63, 97, 98, 99] however rely on traditional DRAM memories which penalize heavily row misses and limit the adoption of more aggressive scheduling policies to avoid hurting throughput.

6.2 Thread Row Buffers

The main problem in all the existing reordering techniques is that the important influence of row buffer locality in the memory system throughput generates a trade-off between throughput and fairness or performance isolation.

In order to solve the limitations of existing configurations, we propose Thread Row Buffers (TRBs). TRBs are extra storage added into the DRAM memory to keep active several rows at the same time. Every row buffer is assigned to a thread, so that every time that a row is activated in the DRAM, the data is copied to the entry of the requesting thread. This overcomes the problem with alternating accesses in multiprogrammed environments and, as we will see, improves the efficiency of fairness or performance isolation oriented memory schedulers.

The addition of an extra storage area to keep active rows was previously proposed in the DRAM Cache [46]. The DRAM Cache, however, proposed a unique cache for all the DRAM memory to store all the rows being activated in the different banks. More recently Loh [82] proposed to divide this cache and distribute it among the different banks. This technique, however, was proposed considering that the memory controller was kept out of the chip and the cache tags could be checked before issuing a request to the DRAM banks. Most current processors, however, have integrated memory controllers on-chip which makes it more difficult to implement this technique. A possible implementation would be to insert all the cache tags and the replacement mechanism with the memory controller. Then the reordering logic of each bank would need to compare the corresponding cache tags for each entry of the bank queue. Depending on the associativity of the cache this would incur a significant delay and energy consumption.

Our proposed mechanism is much simpler and only requires the reordering logic to store the row addresses of the TRBs. Since row buffers are assigned to threads, the replacement mechanism is trivial. Figure 6.1 shows the structure of the DRAM memory with TRBs.

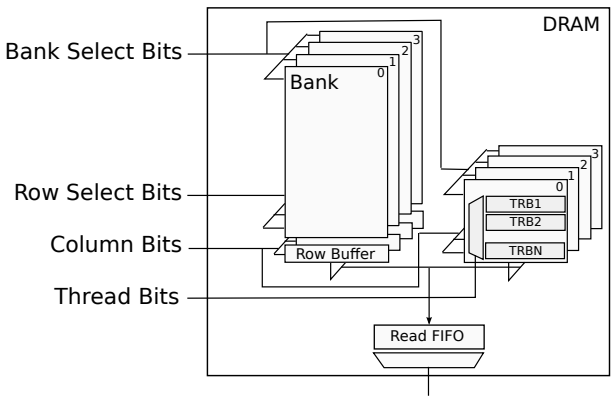


Figure 6.1: DRAM with TRBs structure.

The implementation of Thread Row Buffers requires extra storage in each bank and the corresponding multiplexers to access them. The area overhead of the Thread Row Buffers for the evaluated configuration would be 2.04% of the total memory die (assuming a 56.7 mm^2 die size [124]). Memory arrays would not need to be modified, thus maintaining the same high level of integration that characterizes current DRAM memories.

The implementation of Thread Row Buffers in DRAM chips can be done in several ways, using sense amplifiers like existing row buffers [**?**], using 3t1d memory cells [75] or using SRAM, like on-chip caches, mixed with the DRAM cells [6, 47]. Our implementation assumes the same number of threads as TRB entries in order to allow performance isolation between threads. Nevertheless, TRBs can be used in other contexts with a different number of threads and TRB entries. The implementation of TRBs enables the memory controller to actively decide the replacement policy and management of the TRBs unlike previous proposals (i.e. DRAM Cache) and to manage memory accesses in ways that have not traditionally been possible due to the limit of a single active row. These alternatives are left for future study. For instance, TRBs which could be switched off when not used to reduce the static power consumption or TRBs could be assigned on an application basis.

6.3 Service Partitioning Scheduler

Traditional memory schedulers have been limited by the trade-off of providing the maximum throughput and some kind of fairness or performance isolation. Reordering in the bank queues has been done selecting the request with highest priority. Priorities are calculated concatenating different parameters depending on the type of scheduler. Figure 6.2 shows an example of how the request priority is calculated for a traditional FR-FCFS scheduler

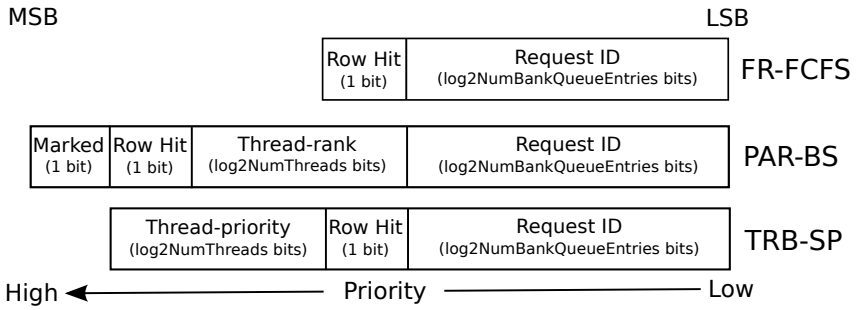


Figure 6.2: Priority calculation for FR-FCS, PAR-BS and TRB-SP.

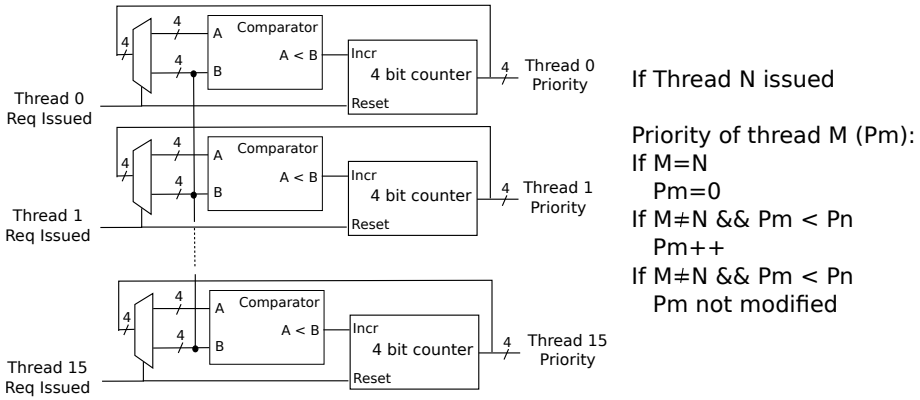


Figure 6.3: Thread priority calculation hardware.

and for the PAR-BS [98] scheduler. Since the memory row hit rate has a very high influence in the overall memory throughput these techniques give more importance to it, potentially reducing the final fairness or isolation. The usage of TRBs, on the other hand, has the advantage that the last row accessed by each thread is always going to be loaded in the TRBs, allowing the memory controller to apply any scheduling policy without hurting throughput.

As an example to show how easy is to provide fairness and performance isolation with TRBs we have prioritized requests with the method depicted in Figure 6.2 as TRB-SP. In order to isolate the performance of the different threads we have used a system which maintains a thread prioritization and uses these priorities to reorder requests, giving more importance to this ordering than the row hit information.

In order to calculate the thread priority for service partitioning we have used a very simple mechanism that uses the most recent history and requires very little hardware. The working principle of our priority calculator is that every time that a request is issued, the owner thread changes its priority to the minimum (zero). The remaining threads that used to have lower priority than that thread increase their priority by one. Therefore, the thread that has been longer without issuing a request is going to have the maximum priority. The

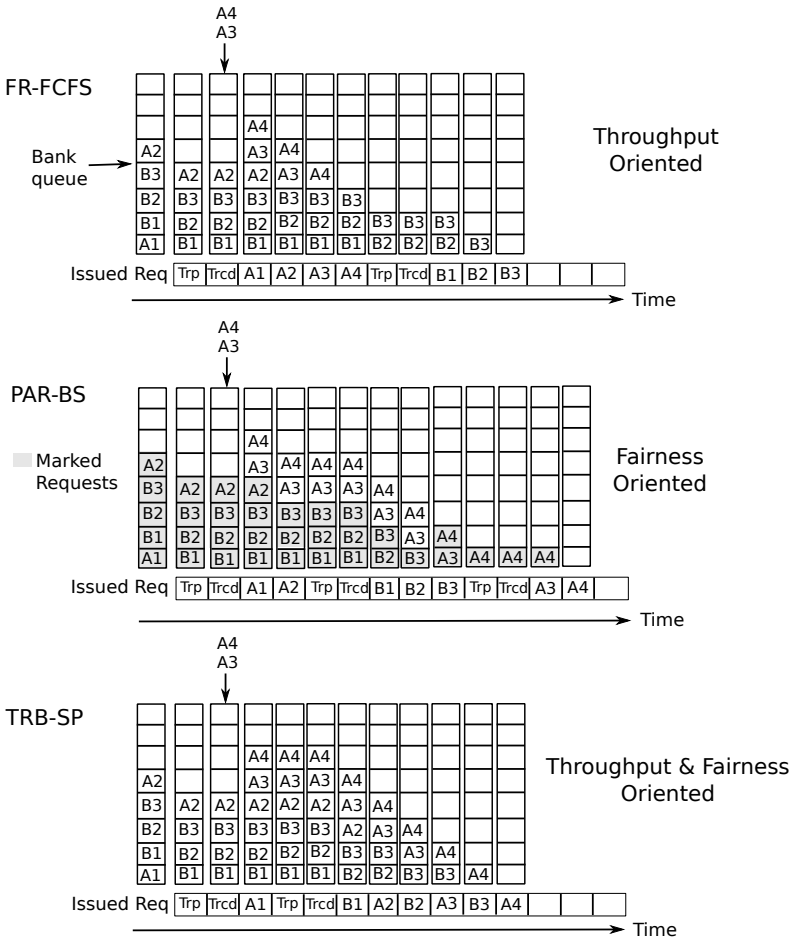


Figure 6.4: Scheduling example.

extra hardware required for priority calculation is very low and can be seen in Figure 6.3. If needed, more complex scheduling mechanisms could be implemented in conjunction with TRBs without having to deal with a performance degradation due to a reduction of the row hit rate. As we will see, the proposed service partitioning mechanism is able to enforce a strong performance isolation without requiring a large hardware overhead.

Figure 6.4 shows an example of how the different scheduling priorities work. We assume two threads (A and B), each of them always accessing the same row. For each configuration we can see on the top the requests stored in the bank queue and on the bottom the requests issued to the DRAM. It is possible to see how FR-FCFS is able to finish very fast. This is because it prioritizes requests if there is a row hit, saving time in precharges and activations. This technique however is not fair and in some cases can stall a request from a different thread for a long time if multiple requests to the active row arrive. On the other, hand PAR-BS prioritizes fairness. The example shows how requests are grouped in a batch at the

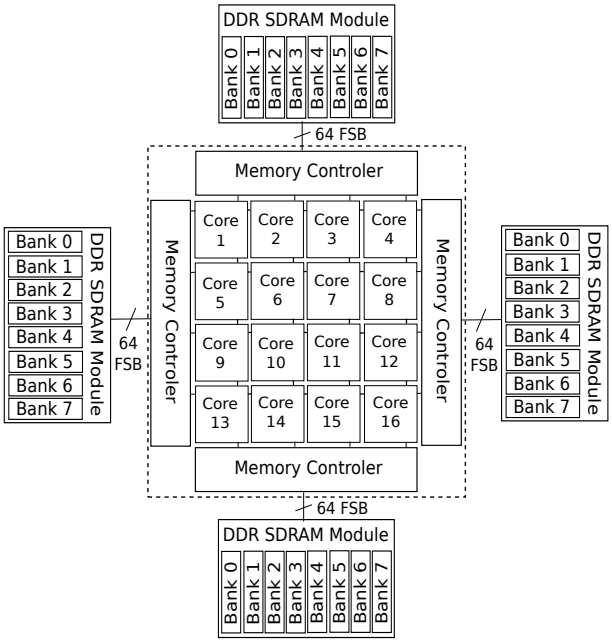


Figure 6.5: Simulated CMP Structure.

beginning and how requests from threads with fewer requests are given priority. Batching, however, does not allow new requests to advance those within a batch even if they are to the active row. Therefore, this reduces the row hit rate and the overall throughput. TRB-SP is able to combine the benefits of both techniques by providing the lowest completion time and, in addition, servicing requests in a fair way. We can see in Figure 6.4 how requests are alternated and this does not penalize the hit rate due to the ability to store several active rows.

6.4 Evaluation

6.4.1 Simulated Configurations

Figure 6.5 shows the evaluated CMP, using 16 processors and 4 memory controllers with a dedicated bus. The overall chip bandwidth is similar to recent processors like IBM Power7 [58] which has a sustained memory bandwidth of more than 100GB/s with 8 channels operating at 6.4 GHz. In all the tested configurations two levels of cache are used; as well as a MOESI protocol to grant coherence between nodes. On-chip coherence is granted through the Distributed Coherence Engines presented in chapter 4. DCEs are distributed across the chip and connected through a mesh network. Local and private L1 and

L2 caches are used in every processor allowing sharing through the DCEs. We have also assumed that the access time of TRBs or the Cache DRAM is the same as for the DRAM, although these parts could have lower access time if implemented with SRAM technology. To emulate the multiprogrammed execution environment of current chip multiprocessors we have used the second benchmark set.

We have evaluated the following configurations to compare the proposed techniques:

FR-FCFS: Our baseline configuration uses 8 bank memories, a FR-FCFS scheduler and an on-chip prefetcher with 32 stream buffers of 8 entries each.

PAR-BS: We have compared to a state of the art scheduling technique, the Parallelism-Aware Batch Scheduling [98]. We have used a Marking-Cap of 5, which is said to be the best compromise between system throughput and fairness. We also evaluated a version with a Marking-Cap of 16 to ensure that prefetch streams were kept within a batch and results were very similar.

Cache DRAM: This configuration evaluates a Cache DRAM [82] with 16 row entries per bank, a LRU replacement policy and a FR-FCFS bank scheduler.

TRB: This organization shows the results obtained for the Thread Row Buffers, using a row buffer per thread in each bank. This configuration also uses a FR-FCFS bank scheduler that only reorders if the first request produces a row miss.

TRB-SP: In this configuration TRBs are evaluated with the Service Partitioning mechanism in the memory controller to grant performance isolation.

6.4.2 Performance and Energy Efficiency

In this section we present the evaluation of the TRB compared to existing configurations described previously. Figure 6.6 shows the weighted Speedup, normalized throughput and row hit rate of the evaluated organizations. It can be seen in the third plot that Thread Row Buffers are able to increase the row hit rate significantly, raising from the 54.4% of FR-FCFS to 75%. This increase is explained by the ability of TRBs to keep several active rows at the same time. Cache DRAM also has this ability and is able to increase the hit rate to 67.6%. The row hit rate improvement results in a lower latency and, therefore, an increase of the overall throughput. Weighted speedup is increased on average by 19.7% over FR-FCFS, 23.9% over PAR-BS and 9.1% over Cache DRAM. In terms of aggregated throughput TRBs also show the best results, improving it on average by 17.1% over FR-FCFS, 21.9% over PAR-BS and 6.9% over Cache DRAM.

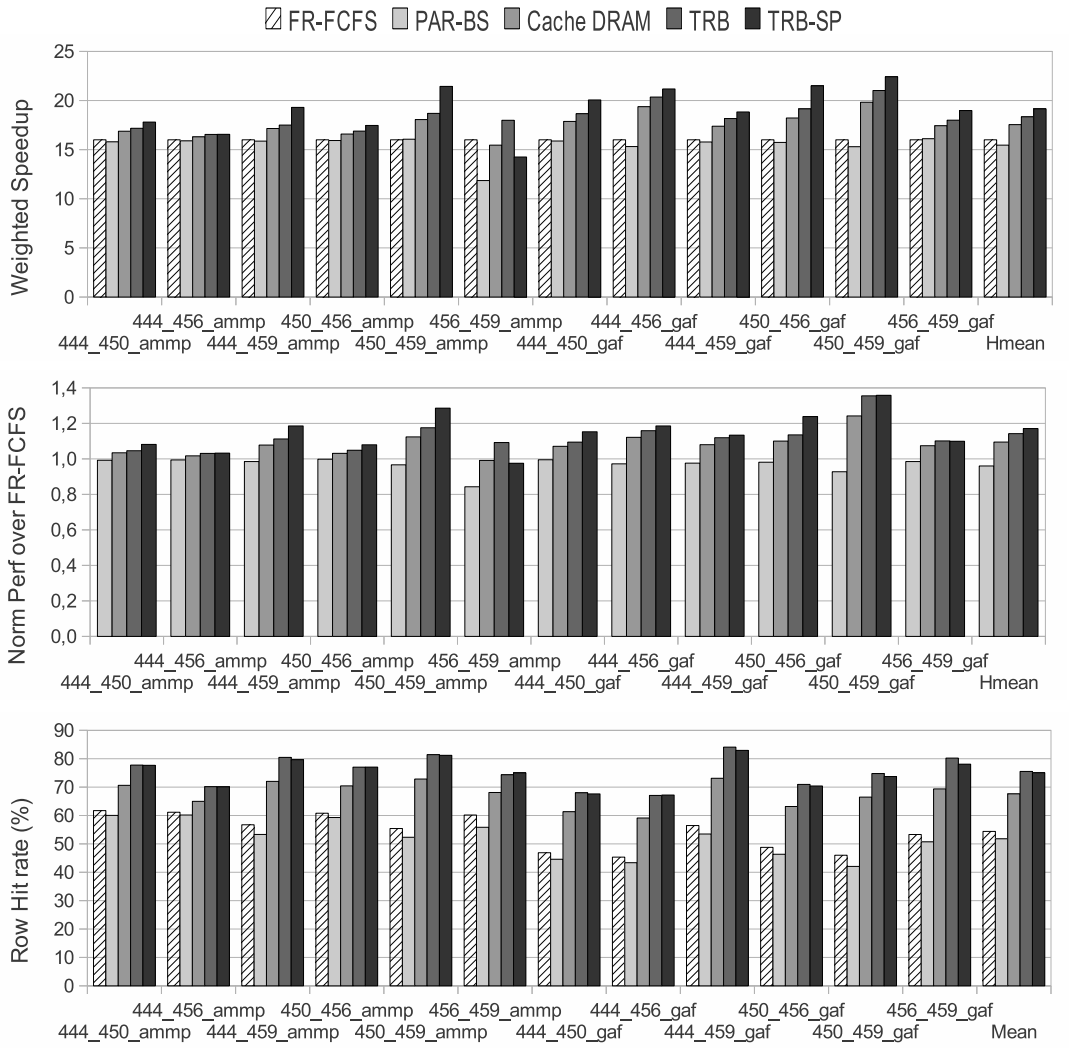


Figure 6.6: Weighted Speedup, normalized throughput and row hit rate.

The performance of PAR-BS is similar to FR-FCFS. This is explained by the low hit rate that this configuration achieves. Previous studies evaluated this technique without prefetching, and therefore lower memory level parallelism (MLP). In that environment a FR-FCFS scheduler is not able to get a reasonable hit rate and, therefore, the penalization of grouping requests into batches is small. The addition of stream prefetchers, however, increases the MLP and shows that under the presence of prefetching batch grouping may reduce the overall throughput.

On the other hand, the TRB configuration has a slightly higher row hit rate than TRB-SP and its performance is lower. This is explained by the performance isolation capability of TRB-SP which is able to reduce the memory latency of threads with lower number of requests. This results in most of the cases in a performance improvement of most of the

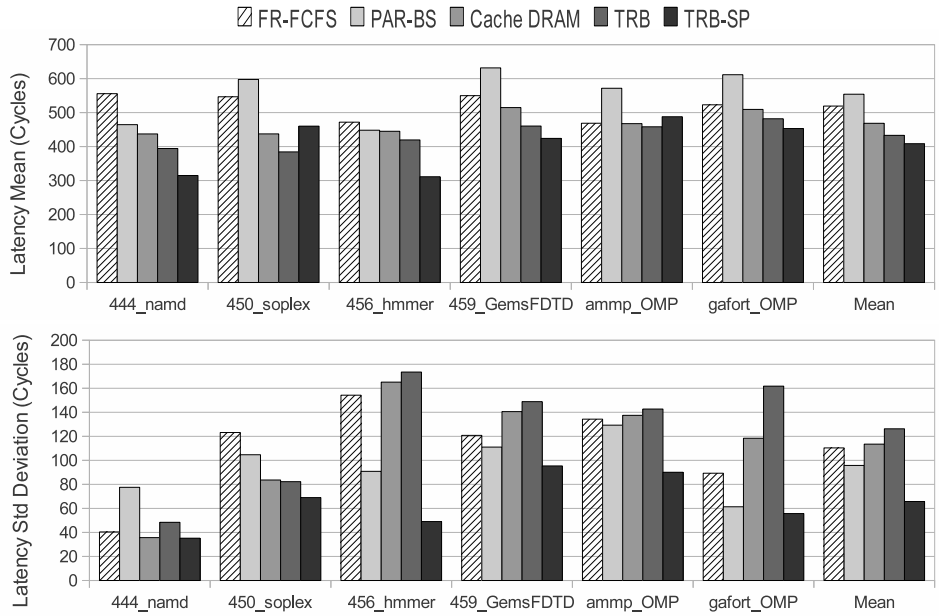


Figure 6.7: Average BM latency and standard deviation.

threads. In the case of 456_459_ammp, however, configurations that enforce performance isolation show a worse performance. This application combination has the highest average number of simultaneous requests and the impact of the scheduling policy is more clear. In this case the Service Partitioning scheduler stalls some requests from ammp to grant a fair access to the other threads, specially those from 456_hmmer. Since ammp represents half of the threads the overall performance is reduced. This reduction, however, is in exchange of a more fair resource allocation and does not imply a reduction in memory throughput.

To measure the performance isolation we have used two different metrics, the average memory latency of each application and its standard deviation, shown in Figure 6.7. The second plot, shows how TRB-SP enforces performance isolation. The performance reduction of ammp in favor of 456_hmmer seen in the 456_459_ammp combination reduces the latency deviation of this application by 68% over FR-FCFS.

PAR-BS also shows a good standard deviation due to the usage of a fair scheduler and is able to reduce it on average by 13.2% over FR-FCFS. The average latency, however, is high due to the low hit rate. The configuration with Service Partitioning, however, shows the best average results. It is able to reduce the average latency by 21.3% over FR-FCFS, 26.3% over PAR-BS and 12.8% over Cache DRAM. Standard deviation is also reduced by 40.4% over FR-FCFS, 31.4% over PAR-BS and 42.1% over Cache DRAM. TRBs, on the other hand, grant that the row hit rate is not affected by thread alternation but applications with a higher number of requests are going to penalize those with less memory

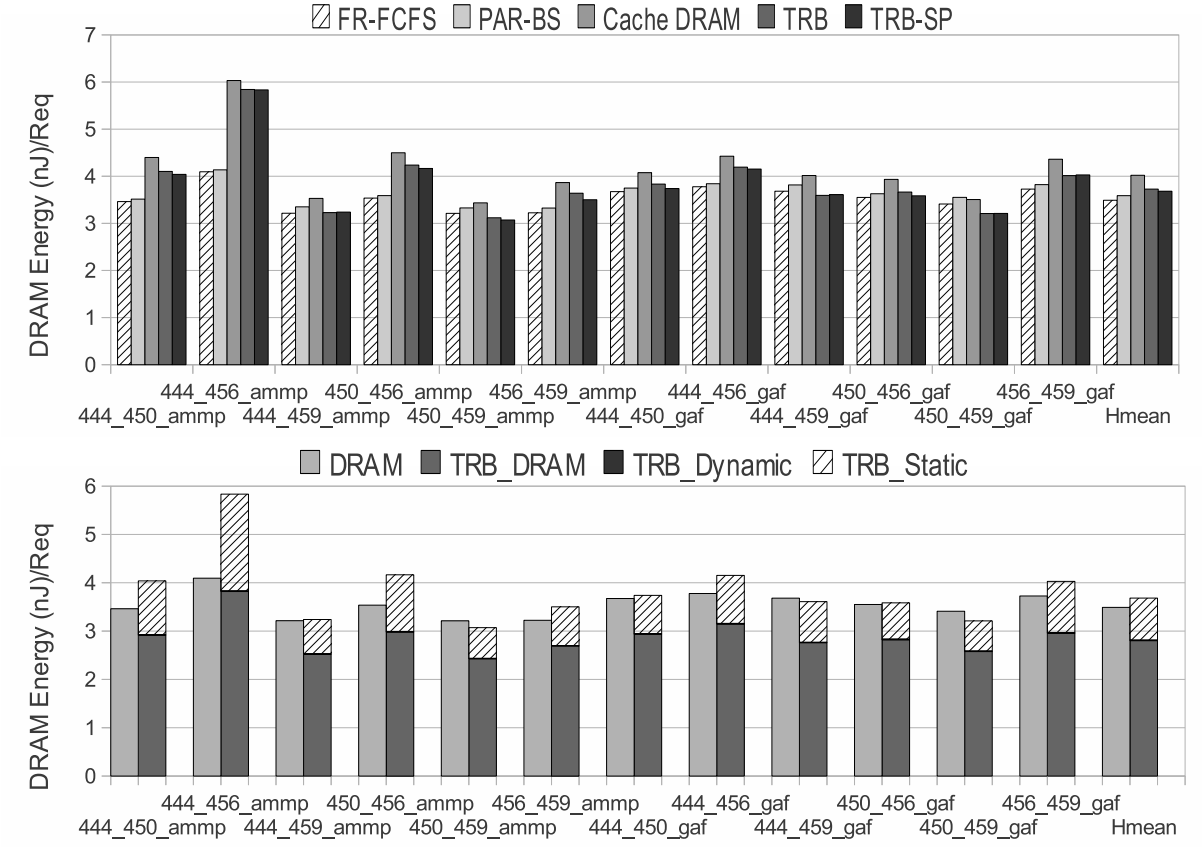


Figure 6.8: TRB-SP memory power decomposition.

accesses. Therefore, the addition of TRBs is not enough to enforce performance isolation Cache DRAM and TRB show the worst variability since they do not have any kind of fair scheduler and are throughput oriented. They increase the standard deviation by 13.2% and 14.4% respectively compared to FR-FCFS.

Furthermore, DRAM power in modern server systems can account for 30% of total system power [8]. And from this power around one third is spent in the precharge and activation of rows. Hence, the aforementioned reduction in row hits also is useful from an energy consumption point of view. In order to evaluate TRB power we have assumed the usage of SRAM TRBs and, therefore, their static power and area has been evaluated with Cacti [123]. Figure 6.8 shows that the power consumption per request of the TRB is similar to the power consumed in the other configurations except for the 444_456_ammp configuration. The extra power in this case is caused by a very small number of requests that do not take advantage of the hit rate improvement. The second plot of Figure 6.8 shows a decomposition of the memory power consumed in the TRB-SP configuration. It can be seen that in all cases the DRAM power consumption is reduced due to a higher hit rate that

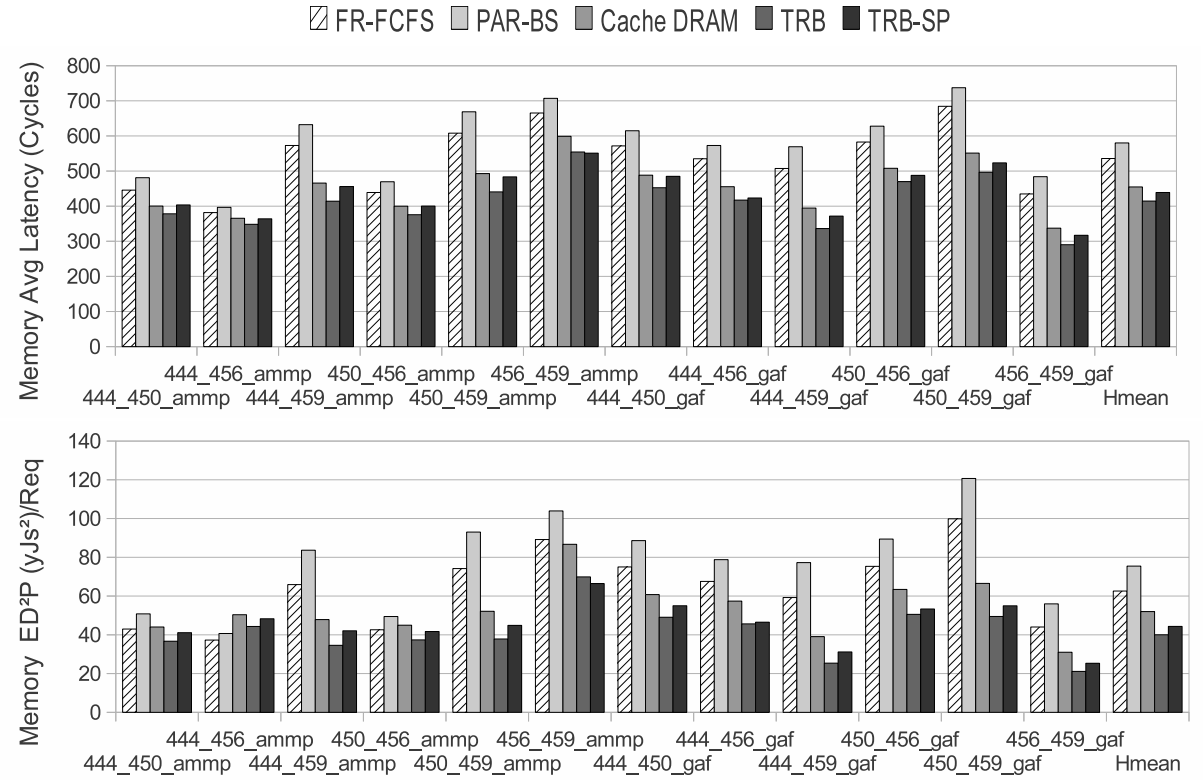


Figure 6.9: Power, Avg latency and Energy-efficiency.

avoids row activations which consume a high amount of energy. However, the extra storage adds a significant amount of static power. On average, TRBs increase the required power per request by 1.8% compared to FR-FCFS but reduce it by 8.3% compared to Cache DRAM.

If we look at the energy-efficiency, however, results are much better for TRBs due to the reduction of the average memory latency (first plot of Figure 6.9). Results show that TRB-SP reduces latency by 18.1% compared to FR-FCFS and by 3.5% compared to Cache DRAM. The second plot shows the energy-delay squared product (ET^2) which is a good measure of energy-efficiency (the lower the better). As expected, a reduced latency with similar power requirements leads to a much higher energy-efficiency for TRB configurations. TRB-SP is able to increase the energy-efficiency reducing the ET^2 by 29.1% over FR-FCFS, 41.2% over PAR-BS and 14.7% over Cache DRAM.

6.4.3 Addition of extra banks

A different alternative to increase the memory access parallelism and increase the overall hit rate would be to use a higher number of banks, each of them with its corresponding row

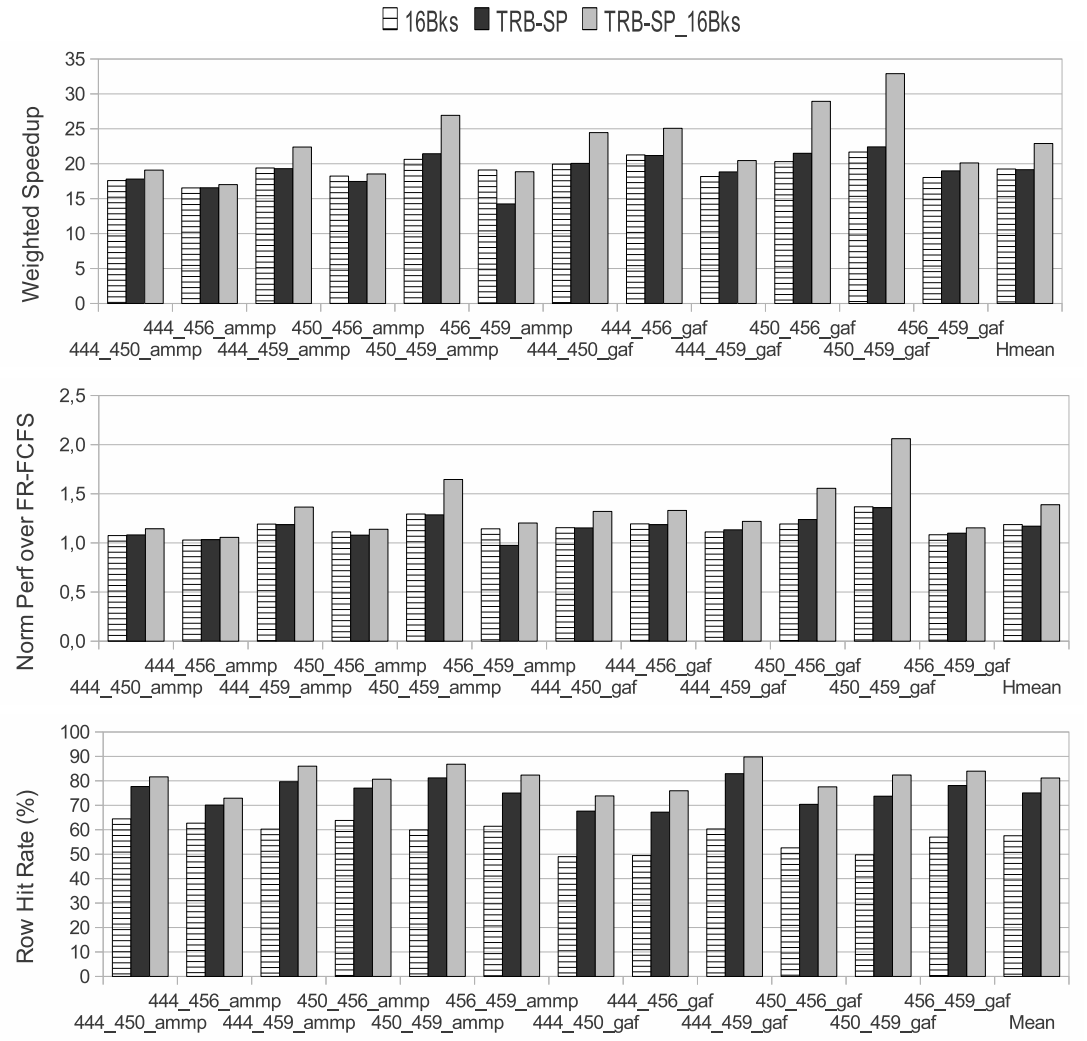


Figure 6.10: Weighted Speedup, normalized throughput and Row Hit rate with extra banks.

buffer. This solution is orthogonal to the proposed technique and could be combined with it.

It is important to note that the addition of extra banks has a high cost in terms of hardware, especially in the memory controller. It requires adding buffer queues and arbiters for each new bank, therefore doubling the required area when doubling the number of banks. In addition, since the extra banks are not allocated to threads, this solution still has to deal with the throughput-isolation trade-off. This configuration, however, also has the benefit of being able to activate or precharge more rows simultaneously. Since it is interesting to see the trade-offs between these alternatives, we have evaluated a configuration with extra banks (16Bks) and a configuration that combines both techniques (TRB-SP_16Bks).

Figure 6.10 shows the performance improvements and row hit rate of both configura-

tions. It can be seen that Thread Row Buffers are able to improve the overall throughput as much as the configuration that doubles the number of banks with a smaller complexity. If we look at the third plot we can see that in terms of row hit rate TRBs are much more efficient than adding extra banks and reaches and average of 75% compared to the 58% of the 16Bks configuration. Finally, we can see that a combination of both techniques, although having a high cost in terms of hardware, can double the overall throughput in some cases and increase it by 39% on average over FR-FCFS.

6.5 Conclusions

In this Chapter we have seen that current chip multiprocessors require that we deal with an increasingly heterogeneous set of applications which change the memory access patterns and encourage the adoption of new organizations. Thread Row Buffers are able to deal with alternate row access with the usage of multiple active rows. It has been shown that they are able to increase the row hit-rate by 38% with respect to FR-FCFS and by 11% with respect to Cache DRAM. This, in turn, increases the overall performance by 17% and 7% respectively.

Furthermore, the increasing number of virtualized environments show a demand for performance isolation between threads. Isolation in existing memory schedulers has been granted by sacrificing throughput due to the usage of traditional DRAM memories. The addition of TRBs eliminates this trade-off allowing the implementation of more aggressive schedulers. The configuration with TRBs and Service Partitioning is able to reduce the standard deviation of an application latency by 40% over FR-FCFS, 31% over PAR-BS and 42% over Cache DRAM.

Overall, we have shown that Thread Row Buffers are an energy-efficient mechanism which allows to avoid the trade-off throughput-isolation and the implementation of simple fairness or Quality-of-Service oriented schedulers without hurting memory throughput. Therefore, TRBs show to be a good alternative to be incorporated in future memory designs.

Chapter 7

Conclusions

Several memory hierarchy organizations have been proposed and analyzed in this thesis for next generation tiled CMPs. We have presented the state of the art in organizations for the memory hierarchy and the challenges that are going to arise in next generation chip multiprocessors. From there, we have evaluated current solutions and found their limitations in order to propose new approaches. In particular, we have presented several techniques for the memory hierarchy of chip multiprocessors that we summarize in the following paragraphs.

7.1 Thesis Contributions

In Chapter 4, we showed that traditional cache organizations such as shared or private caches behave well only for some applications and that an adaptive system would be desirable. Cooperative Caching is a framework that takes advantage of the benefits of both worlds. This technique, however, requires the usage of a centralized coherence structure and has a high energy consumption. We presented the Distributed Cooperative Caching, a mechanism to provide coherence to chip multiprocessors and apply the concept of cooperative caching in a distributed way. Furthermore, we showed that our tag allocation method is much more flexible and energy-efficient.

In addition, we presented a study of the effectiveness and reuse patterns of the spilling technique, showing that not always is necessary and that most of the reuse is done by the evicting node. We proposed a Distance-Aware and a Selective Spilling mechanisms to solve these issues. Distance-Aware Spilling allows to reduce the spilling distance and, therefore, the network traffic and energy consumption. Selective Spilling on the other hand,

evaluates dynamically the effectiveness of spilling and deactivates it when not effective, also reducing the overall energy consumption.

In Chapter 5, we showed that applications make different uses of cache and that an efficient allocation can take advantage of unused resources. We have proposed Elastic Cooperative Caching, an adaptive cache organization able to redistribute cache resources dynamically depending on application requirements. One of the most important contributions of this technique is that adaptivity is fully managed by hardware and that all repartitioning mechanisms are based on distributed structures, allowing a better scalability. ElasticCC not only is able to repartition cache sizes to application requirements, but also is able to dynamically adapt to the different execution phases of each thread. Our experimental evaluation also has shown that the cache partitioning provided by ElasticCC is efficient and is almost able to match the off-chip miss rate of a configuration that doubles the cache space.

Finally, in Chapter 6 we have studied the behavior of current memory controllers. A detailed model has been implemented and it has been evaluated in a multiprogrammed environment. Although traditional memory schedulers work well for uniprocessors, we showed that new access patterns advocate for a redesign of some parts of DRAM memories. Several new organizations exist for the DRAM schedulers, however, all of them must tradeoff between memory throughput and fairness. We proposed Thread Row Buffers, an extended storage area in DRAM memories able to store a data row for each thread. This mechanism enables a fair memory access scheduling without hurting memory throughput. This is specially important in multiprogrammed systems where performance isolation and Quality-of-Service is desired. We showed that Thread Row Buffers enable the implementation of very simple bank schedulers that outperform existing state of the art techniques.

7.2 Future Work

In this thesis, we have shown that the implementation of application-aware caches can improve significantly their performance in chip multiprocessors. Therefore, in the future, it is going to be important to study the characteristics of common applications to detect their characteristics and design adaptive cache systems able to maximize performance and power efficiency. ElasticCC provides a good starting point due to its simplicity and adaptivity to current applications. An interesting research direction would be to study hardware-software co-design approaches which could do a finner grain classification. This approach can increase the programming complexity but, on the other hand, it can improve the cache behavior and the processor performance if correctly managed by the compiler.

In the memory controller part, it has been shown that arbitration between threads is going to be critical in the overall system performance. The proposed Thread Row Buffers, open a new research path in the design of memory schedulers. The disappearance of the throughput-fairness trade-off in the DRAM scheduling allows to design new memory schedulers with more aggressive prioritization mechanisms. Depending on the system requirements in terms of isolation or QoS, specifically suited schedulers can be proposed.

Glossary

Activate Action of loading a row of data in the row buffer of a DRAM memory.

Bank Region of a DRAM memory. All banks can be accessed in parallel

Cache Storage component that keeps a portion of the data close to the consumer. It allows to take advantage of program locality and reduce the number of memory requests. Typically implemented with SRAM memory.

CMP Chip multiprocessor. Chip containing more than one computing element.

Coherence Engine Directory responsible of granting on-chip coherence.

DRAM Dynamic Random Access Memory. Storage component slower than SRAM but with a much higher integration density. It only requires one transistor and one capacitor per bit. Typically is used as an intermediate storage component between caches and disk.

Low Utility Application Application with low temporal locality that makes an intensive use of the memory hierarchy without reusing the data.

Memory Controller Scheduling mechanism responsible of managing DRAM memories and arbitrate among memory requests.

N-Chance forwarding Technique for sharing cache space with private caches. On an eviction a block is forwarded N times to other caches before being evicted from the chip.

Precharge Action of writing back the row buffer data to the DRAM storage.

Private High Utility Application Application that benefits from larger levels of memory hierarchy but does not share data between threads

Row Also known as page. Portion of data in a DRAM memory, typically consisting of around 128 blocks.

Row Buffer Storage area in DRAM memories where data rows are allocated. All data in DRAM memories must be read from the row buffers.

Saturating Utility Application Application characterized by having a small working set that fits in the cache.

Shared High Utility Application Application that benefits from larger levels of memory hierarchy and his data is shared between threads

Spilling See N-Chance forwarding.

SRAM Static Random Access Memory. Storage component usually composed of six transistors per bit. Four transistors conform the storage cell through two cross-coupled inverters and the other two are used to control the access to data. It is faster and less power hungry than DRAM but also less dense.

References

[1] J. ABELLA AND A. GONZÁLEZ. **Heterogeneous way-size cache**. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 239–248, New York, NY, USA, 2006. ACM. 13

[2] M.E. ACACIO, J. GONZALEZ, J.M. GARCIA, AND J. DUATO. **A new scalable directory architecture for large-scale multiprocessors**. *HPCA '01: 7th International Symposium on High-Performance Computer Architecture*, pages 97–106, 2001. 11

[3] A. AGARWAL, R. SIMONI, J. HENNESSY, AND M. HOROWITZ. **An evaluation of directory schemes for cache coherence**. In *Proceedings of the 15th Annual International Symposium on Computer architecture*, ISCA '88, pages 280–298, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press. 9

[4] J.K. ARCHIBALD. **A cache coherence approach for large multiprocessor systems**. In *Proceedings of the 2nd international conference on Supercomputing*, ICS '88, pages 337–345, New York, NY, USA, 1988. ACM. 10

[5] V. ASLOT AND R. EIGENMANN. **Quantitative performance analysis of the SPEC OMPM2001 benchmarks**. *Scientific Programming*, **11**(2):105–124, 2003. 46, 103

[6] F. ASSADERAGHI, L.L-C. HSU, AND J.A. MANDELMAN. **Mixed memory integration with NVRAM, DRAM and SRAM cell structures on same substrate**. *US Patent 6,424,011*, 23 Jul. 2002. 113

[7] SEMICONDUCTOR INDUSTRIES ASSOCIATION. **International Technology Roadmap for Semiconductors**. Technical report, http://www.itrs.net/reports.html, 2005. 3, 41

[8] L. BARROSO AND H. HOLZLE. *The Datacenter as a Computer: An introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2009. 19, 28, 120

[9] B.M. BECKMANN, M.R. MARTY, AND D.A. WOOD. **ASR: Adaptive Selective Replication for CMP Caches**. *MICRO-39: 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2006. 11, 13, 14, 92, 100, 101

[10] B.M. BECKMANN AND D.A. WOOD. **Managing Wire Delay in Large Chip-Multiprocessor Caches**. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 319–330, Washington, DC, USA, 2004. IEEE Computer Society. 13

[11] C. GORDON BELL, JOHN E. MCNAMARA, AND J. CRAIG. MUDGE. *Computer Engineering; A DEC View of Hardware Systems Design*. Butterworth-Heinemann, Newton, MA, USA, 1978. 7

[12] C.G. BELL, R.C. CHEN, S.H. FULLER, J. GRASON, S. REGE, AND D.P. SIEWIOREK. **The Architecture and Applications of Computer Modules: A Set of Components for Digital Design**. *IEEE Compcon.*, **73**:177–180, March 1973. 7

[13] L. BLOOM, M. COHEN, AND S. PORTER. **"Considerations" in the Design of a Computer with High Logic-to-Memory Speed Ratio**. In *Proceedings of Gigacycle Computing Systems*, **5–136** of *AIEE Special Publ.*, pages 53–63, 1962. 1

[14] F. BRIGGS, M. CEKLEOV, K. CRETA, M. KHARE, S. KULICK, A. KUMAR, L.P. LOOI, C. NATARAJAN, S. RADHAKRISHNAN, AND L. RANKIN. **Intel 870: A Building Block for Cost-Effective, Scalable Servers**. *IEEE Micro*, **22**:36–47, March 2002. 23, 24

[15] D.M. BROOKS, P. BOSE, S.E. SCHUSTER, H. JACOBSON, P.N. KUDVA, A. BUYUKTOSUNOGLU, J-D. WELLMAN, V. ZYUBAN, M. GUPTA, AND P.W. COOK. **Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors**. *IEEE Micro*, **20**(6):26–44, 2000. 38

[16] J. CHANG, E. HERRERO, R. CANAL, AND G. SOHI. *Cooperative Caching for Chip Multiprocessors. In Cooperative Networking. ISBN: 978-0-470-74915-9*. Wiley, 2011. 5

[17] J. CHANG AND G.S. SOHI. **Cooperative Caching for Chip Multiprocessors**. In *ISCA '06: 33rd Annual International Symposium on Computer Architecture*, pages 264–276, 2006. 11, 13, 15, 60, 61, 92

[18] J. CHANG AND G.S. SOHI. **Cooperative cache partitioning for chip multiprocessors**. In *ICS '07: 21st Annual International Conference on Supercomputing*, pages 242–252, 2007. 13, 16, 92

[19] A. CHARLESWORTH, N. ANESHANSLEY, M. HAAKMEESTER, D. DROGICHEN, G. GILBERT, R. WILLIAMS, AND A. PHELPS. **The Starfire SMP interconnect**. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, Supercomputing '97, pages 1–20, New York, NY, USA, 1997. ACM. 11

[20] M. CHAUDHURI. **PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches**. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 227 –238, 2009. 13

[21] D. CHIOU. **Extending the Reach of Microprocessors: Column and Curious Caching**. *PhD Thesis, Massachusets Institute of Technology*, 1999. 17, 49, 92, 94

[22] Z. CHISHTI, M.D. POWELL, AND T.N. VIJAYKUMAR. **Optimizing Replication, Communication, and Capacity Allocation in CMPs**. *International Symposium on Computer Architecture*, **0**:357–368, 2005. 11, 13, 14

[23] Z. CHISHTI, M.D. POWELL, AND T.N. VIJAYKUMAR. **Distance associativity for high-performance energy-efficient non-uniform cache architectures**. *MICRO-36: 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 55–66, 3-5 Dec. 2003. 60, 92

[24] BURROUGHS CORPORATION. **The operational characteristics of the processors for the Burroughs B5000**. Technical report, Burroughs Corporation, 1962. 7

[25] A.L. COX AND R.J. FOWLER. **Adaptive cache coherency for detecting migratory shared data**. In *Proceedings of the 20th annual international symposium on Computer architecture*, ISCA '93, pages 98–108, New York, NY, USA, 1993. ACM. 13

[26] D.E. CULLER, A. GUPTA, AND J.P. SINGH. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997. 7

[27] M.D. DAHLIN, R.Y. WANG, T.E. ANDERSON, AND D.A. PATTERSON. **Cooperative Caching: Using remote client memory to improve file system performance**. *OSDI '94: 1st Conference on Operating Systems Design and Implementation*, pages 267–280, Nov. 1994. 94

[28] J.D. DAVIS, J. LAUDON, AND K. OLUKOTUN. **Maximizing CMP throughput with mediocre cores**. *PACT '05: 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 51–62, 17-21 Sept. 2005. 36

[29] PRADEEP DUBEY. **A Platform 2015 Workload Model: Recognition, Mining and Synthesis Moves Computers to the Era of Tera**. *Intel White Paper ftp://download.intel.com/technology/computing/archinnov/platform2015/download/RMS.pdf Intel Corporation*, 2005. 32

[30] H. DYBDAHL AND P. STENSTROM. **An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors**. *HPCA '07: 13th International Symposium on High Performance Computer Architecture*, pages 2–12, 10-14 Feb. 2007. 11, 13, 16, 17, 92

[31] E. EBRAHIMI, O. MUTLU, C.J. LEE, AND Y.N. PATT. **Coordinated control of multiple prefetchers in multi-core systems**. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 316–326, New York, NY, USA, 2009. ACM. 19, 23, 27

[32] B. FALSAFI AND D.A. WOOD. **Reactive NUMA: a design for unifying S-COMA and CC-NUMA**. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 229–240, New York, NY, USA, 1997. ACM. 11

[33] K. GHARACHORLOO, A. NOWATZYK, R. MCNAMARA, R. STETS, S. SMITH, S. QADEER, B. SANO, B. VERGHESE, AND L.A. BARROSO. **Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing**. *Computer Architecture, International Symposium on*, **0**:282, 2000. 11

[34] A. GONZÁLEZ, C. ALIAGAS, AND M. VALERO. **A data cache with multiple caching strategies tuned to different types of locality**. In *Proceedings of the 9th international conference on Supercomputing*, ICS '95, pages 338–347, New York, NY, USA, 1995. ACM. 13

[35] J.R. GOODMAN. **Using cache memory to reduce processor-memory traffic**. In *Proceedings of the 10th annual international symposium on Computer architecture*, ISCA '83, pages 124–131, New York, NY, USA, 1983. ACM. 10

[36] E. HAGERSTEN, A. LANDIN, AND S. HARIDI. **DDM: A Cache-Only Memory Architecture**. *Computer*, **25**:44–54, September 1992. 11

[37] E.G. HALLNOR AND S.K. REINHARDT. **A fully associative software-managed cache design**. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 107–116, New York, NY, USA, 2000. ACM. 13

[38] N. HARDAVELLAS, M. FERDMAN, B. FALSAFI, AND A. AILAMAKI. **Reactive NUCA: near-optimal block placement and replication in distributed caches**. In *ISCA '09: 36th annual international symposium on Computer architecture*, pages 184–195, New York, NY, USA, 2009. ACM. 10, 11, 13, 16, 92

[39] J.L. HENNESSY AND D.A. PATTERSON. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003. 2, 9

[40] J.L. HENNING. **SPEC CPU2006 benchmark descriptions**. *SIGARCH Comput. Archit. News*, **34**(4):1–17, 2006. 46

[41] M. HERLIHY AND J.E.B. MOSS. **Transactional memory: architectural support for lock-free data structures**. In *Proceedings of the 20th annual international symposium on*

Computer architecture, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM. 12

[42] E. HERRERO, J. GONZÁLEZ, AND R. CANAL. **Distributed cooperative caching**. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 134–143, New York, NY, USA, 2008. ACM. 4

[43] E. HERRERO, J. GONZÁLEZ, AND R. CANAL. **Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors**. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, pages 419–428, New York, NY, USA, 2010. ACM. 5

[44] E. HERRERO, J. GONZÁLEZ, AND R. CANAL. **Power-Efficient Spilling Techniques for Chip Multiprocessors**. In PASQUA D'AMBRA, MARIO GUARRACINO, AND DOMENICO TALIA, editors, *Euro-Par 2010 - Parallel Processing*, **6271** of *Lecture Notes in Computer Science*, pages 256–267. Springer Berlin / Heidelberg, 2010. 4, 60

[45] E. HERRERO, J. GONZÁLEZ, AND R. CANAL. **Distributed Cooperative Caching: An Energy Efficient Memory Scheme for Chip Multiprocessors.** *IEEE Transactions on Parallel and Distributed Systems (To Appear)*, 2011. 5

[46] H. HIDAKA, Y. MATSUDA, M. ASAKURA, AND K. FUJISHIMA. **The Cache DRAM Architecture: A DRAM with an On-Chip Cache Memory**. *IEEE Micro*, **10**(2):14–25, 1990. 112

[47] L.L. HSU, C. RADENS, AND L-K. WANG. **Integrated chip having SRAM, DRAM and Flash memory and method for fabricating the same**. *US Patent 6,556,477*, 29 Apr. 2003. 113

[48] L.R. HSU, S.K. REINHARDT, R. IYER, AND S. MAKINENI. **Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource**. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 13–22, New York, NY, USA, 2006. ACM. 12

[49] J. HUH, D. BURGER, AND S.W. KECKLER. **Exploring the Design Space of Future CMPs**. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 199–210, Washington, DC, USA, 2001. IEEE Computer Society. 2

[50] J. HUH, C. KIM, H. SHAFI, L. ZHANG, D. BURGER, AND S.W. KECKLER. **A NUCA substrate for flexible CMP cache sharing**. In *ICS '05: 19th Annual International Conference on Supercomputing*, pages 31–40, 2005. 13, 15, 16, 60, 92

[51] I. HUR AND C. LIN. **Adaptive History-Based Memory Schedulers**. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–354, Washington, DC, USA, 2004. IEEE Computer Society. 23, 24

[52] INTEL. **The SCC Platform Overview**. Technical report, Intel Labs, May 2010. 3, 10, 32

[53] E. IPEK, O. MUTLU, J.F. MARTÍNEZ, AND R. CARUANA. **Self-Optimizing Memory Controllers: A Reinforcement Learning Approach**. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 39–50, Washington, DC, USA, 2008. IEEE Computer Society. 23, 24

[54] R. IYER. **CQoS: a framework for enabling QoS in shared caches of CMP platforms**. In *ICS '04: Proceedings of the 18th annual*

international conference on Supercomputing, pages 257–266, New York, NY, USA, 2004. ACM. 11, 13, 16, 92

[55] J.A. GREGORIO J. MERINO, V. PUENTE. **ESP-NUCA: A Low-cost Adaptive Non-Uniform Cache Architecture**. *HPCA '10: 16th International Symposium on High Performance Computer Architecture*, 9-14 Jan. 2010. 11, 13

[56] N.D.E. JERGER, L-S. PEH, AND M.H. LIPASTI. **Virtual Tree Coherence: Leveraging Regions and In-Network Multicast Trees for Scalable Cache Coherence**. *MICRO-41: 41st Annual IEEE/ACM International Symposium on Microarchitecture*, 8-12 Nov. 2008. 92

[57] N.P. JOUPPI. **Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers**. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 364–373, New York, NY, USA, 1990. ACM. 53

[58] R. KALLA AND B. SINHAROY. **POWER7: IBM's Next Generation Server Processor**. In *Hot Chips'09: 21st Symposium on High Performance Chips*, Standford University, CA, USA, 23-25 Aug. 2009. 116

[59] M. KANDEMIR, F. LI, M.J. IRWIN, AND S.W. SON. **A novel migration-based NUCA design for chip multiprocessors**. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 28:1–28:12, Piscataway, NJ, USA, 2008. IEEE Press. 13

[60] C. KIM, D. BURGER, AND S.W. KECKLER. **An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches**. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 211–222, New York, NY, USA, 2002. ACM. 11, 13, 15

[61] J.S. KIM, M.B. TAYLOR, J. MILLER, AND D. WENTZLAFF. **Energy characterization of a tiled architecture processor with on-chip networks**. In *ISLPED '03: International symposium on Low power electronics and design*, pages 424–427, 2003. 39

[62] S. KIM, D. CHANDRA, AND Y. SOLIHIN. **Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture**. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society. 11, 13

[63] Y. KIM, D. HAN, O. MUTLU, AND M. HARCHOL-BALTER. **ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers**. In *HPCA '10: Proceedings of the 2010 IEEE 16th International Symposium on High Performance Computer Architecture*, Washington, DC, USA, 2010. IEEE Computer Society. 18, 23, 26, 29, 112

[64] Y. KIM, M. PAPAMICHAEL, O. MUTLU, AND M. HARCHOL-BALTER. **Prefetch-Aware DRAM Controllers**. In *MICRO 43: Proceedings of the 43th annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2010. IEEE Computer Society. 18, 23, 27, 29

[65] P. KONGETIRA, K. AINGARAN, AND K. OLUKOTUN. **Niagara: a 32-way multithreaded Sparc processor**. *Micro, IEEE*, **25**(2):21 – 29, 2005. 11

[66] R. KUMAR, V. ZYUBAN, AND D.M. TULLSEN. **Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling**. In *ISCA '05: 32nd An-*

nual International Symposium on Computer Architecture, pages 408–419, Washington, DC, USA, 2005. IEEE Computer Society. 101

[67] C-C. KUO, J.B. CARTER, R. KURAMKOTE, AND M.R. SWANSON. **ASCOMA: An Adaptive Hybrid Shared Memory Architecture**. In *Proceedings of the 1998 International Conference on Parallel Processing*, ICPP '98, pages 207–216, Washington, DC, USA, 1998. IEEE Computer Society. 11

[68] J. LAUDON AND D. LENOSKI. **The SGI Origin: a ccNUMA highly scalable server**. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 241–251, New York, NY, USA, 1997. ACM. 11

[69] A.R. LEBECK, X. FAN, H. ZENG, AND C. ELLIS. **Power aware page allocation**. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 105–116, New York, NY, USA, 2000. ACM. 19, 23, 29

[70] C.J. LEE, O. MUTLU, V. NARASIMAN, AND Y.N. PATT. **Prefetch-Aware DRAM Controllers**. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 200–209, Washington, DC, USA, 2008. IEEE Computer Society. 19, 23, 28

[71] C.J. LEE, V. NARASIMAN, O. MUTLU, AND Y.N. PATT. **Improving memory bank-level parallelism in the presence of prefetching**. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 327–336, New York, NY, USA, 2009. ACM. 19, 23, 28

[72] H-H.S. LEE, G.S. TYSON, AND M.K. FARRENS. **Eager writeback - a technique for improving bandwidth utilization**. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 11–21, New York, NY, USA, 2000. ACM. 24

[73] D. LENOSKI, J. LAUDON, K. GHARACHORLOO, A. GUPTA, AND J. HENNESSY. **The directory-based cache coherence protocol for the DASH multiprocessor**. *ISCA '90: 17th Annual International Symposium on Computer Architecture*, pages 148–159, 28-31 May 1990. 11

[74] D. LENOSKI, J. LAUDON, K. GHARACHORLOO, W-D. WEBER, A. GUPTA, J. HENNESSY, M. HOROWITZ, AND M.S. LAM. **The Stanford Dash Multiprocessor**. *Computer*, **25**:63–79, March 1992. 11

[75] XIAOYAO LIANG, RAMON CANAL, GU-YEON WEI, AND DAVID BROOKS. **Process Variation Tolerant 3T1D-Based Cache Architectures**. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 15–26, Washington, DC, USA, 2007. IEEE Computer Society. 113

[76] J. LIN, Q. LU, X. DING, Z. ZHANG, X. ZHANG, AND P. SADAYAPPAN. **Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems**. In *HPCA '08: Proceedings of the 14th International Symposium on High Performance Computer Architecture*, pages 367–378, Feb. 2008. 12, 91

[77] W-F. LIN, S.K. REINHARDT, AND D. BURGER. **Reducing DRAM Latencies with an Integrated Memory Hierarchy Design**. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 301, Washington, DC, USA, 2001. IEEE Computer Society. 19, 23, 28

[78] J. LIRA, C. MOLINA, AND A. GONZÁLEZ. **Last Bank: Dealing with Address Reuse in Non-Uniform Cache Architecture for CMPs**. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 297–308, Berlin, Heidelberg, 2009. Springer-Verlag. 15

[79] J. LIRA, C. MOLINA, AND A. GONZÁLEZ. **LRU-PEA: a smart replacement policy for non-uniform cache architectures on chip multiprocessors**. In *Proceedings of the 2009 IEEE international conference on Computer design*, ICCD'09, pages 275–281, Piscataway, NJ, USA, 2009. IEEE Press. 15

[80] J. LIRA, C. MOLINA, AND A. GONZÁLEZ. **The auction: optimizing banks usage in Non-Uniform Cache Architectures**. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 37–47, New York, NY, USA, 2010. ACM. 13, 15

[81] C. LIU, A. SIVASUBRAMANIAM, AND M. KANDEMIR. **Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs**. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 176, Washington, DC, USA, 2004. IEEE Computer Society. 16, 92

[82] G.H. LOH. **3D-Stacked Memory Architectures for Multi-core Processors**. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 453–464, Washington, DC, USA, 2008. IEEE Computer Society. 112, 117

[83] M. LUPON, G. MAGKLIS, AND A. GONZALEZ. **FASTM: A Log-based Hardware Transactional Memory with Fast Abort Recovery**. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 293–302, Washington, DC, USA, 2009. IEEE Computer Society. 12

[84] M. LUPON, G. MAGKLIS, AND A. GONZALEZ. **A Dynamically Adaptable Hardware Transactional Memory**. In *Proceedings of the 43th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 43, Washington, DC, USA, 2010. IEEE Computer Society. 12

[85] P.S. MAGNUSSON, M. CHRISTENSSON, J. ESKILSON, D. FORSGREN, G. HALLBERG, J. HOGBERG, F. LARSSON, A. MOESTEDT, AND B. WERNER. **Simics: A Full System Simulation Platform**. *Computer*, **35**(2):50–58, 2002. 35

[86] M. MAMIDIPAKA AND N. DUTT. **eCacti: An Enhanced Power Estimation Model for On-chip Caches**. Technical report, University of California Irvine Center for Embedded Computer Systems, September 2004. 40

[87] M.M.K. MARTIN, M.D. HILL, AND D.A. WOOD. **Token Coherence: decoupling performance and correctness**. *ISCA '03: 30th Annual International Symposium on Computer Architecture*, pages 182–193, 9-11 June 2003. 10, 92

[88] M.M.K. MARTIN, D.J. SORIN, B.M. BECKMANN, M.R. MARTY, M. XU, A.R. ALAMELDEEN, K.E. MOORE, M.D. HILL, AND D.A. WOOD. **Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset**. *SIGARCH Comput. Archit. News*, **33**(4):92–99, 2005. 35

[89] M.R. MARTY, J.D. BINGHAM, M.D. HILL, A.J. HU, M.M.K. MARTIN, AND D.A. WOOD. **Improving multiple-CMP systems using token coherence**. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 328 – 339, 2005. 10

[90] S.A. MCKEE AND W.A. WULF. **Access ordering and memory-conscious cache utilization**. In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, HPCA '95, pages 253–, Washington, DC, USA, 1995. IEEE Computer Society. 27

[91] MICRON. **DDR3 SDRAM MT41J128M8 Datassheet**. Technical report, Micron Technology Inc., 2006. xi, 37

[92] MICRON. **TN-41-01, Calculating Memory System Power for DDR3**. Technical report, Micron Technology Inc., 2007. 37, 39

[93] A. MOGA AND M. DUBOIS. **The Effectiveness of SRAM Network Caches in Clustered DSMs**. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 103–, Washington, DC, USA, 1998. IEEE Computer Society. 11

[94] M. MONCHIERO, R. CANAL, AND A. GONZALEZ. **Power/Performance/Thermal Design Space Exploration for Multicore Architectures**. *IEEE Transactions on Parallel and Distributed Systems*, **19**(5):666–681, May. 2008. 39

[95] G.E. MOORE. **Cramming more components onto integrated circuits.** *Electronics*, pages 114–117, 1965. 1

[96] R. MULLINS. **Minimising Dynamic Power Consumption in On-Chip Networks**. *International Symposium on System-on-Chip*, pages 1–4, Nov. 2006. 39

[97] O. MUTLU AND T. MOSCIBRODA. **Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors**. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 146–160, Washington, DC, USA, 2007. IEEE Computer Society. 18, 23, 25, 112

[98] O. MUTLU AND T. MOSCIBRODA. **Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems**. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 63–74, Washington, DC, USA, 2008. IEEE Computer Society. 18, 23, 25, 29, 112, 114, 117

[99] K.J. NESBIT, N. AGGARWAL, J. LAUDON, AND J.E. SMITH. **Fair Queuing Memory Systems**. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 208–222, Washington, DC, USA, 2006. IEEE Computer Society. 18, 23, 25, 112

[100] K.J. NESBIT, J. LAUDON, AND J.E. SMITH. **Virtual private caches**. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 57–68, New York, NY, USA, 2007. ACM. 16, 92

[101] A. PADEGS. **System/360 and beyond**. *IBM J. Res. Dev.*, **25**:377–390, September 1981. 7

[102] M.K. QURESHI. **Adaptive Spill-Receive for robust high-performance caching in CMPs**. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 45 –54, 2009. 17, 18

[103] M.K. QURESHI, D.N. LYNCH, O. MUTLU, AND Y.N. PATT. **A Case for MLP-Aware Cache Replacement**. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 167–178, Washington, DC, USA, 2006. IEEE Computer Society. 11, 13

[104] M.K. QURESHI AND Y.N. PATT. **Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches**. *MICRO-39: 39th*

Annual IEEE/ACM International Symposium on Microarchitecture, pages 423–432, Dec. 2006. 13, 16, 17, 50, 60, 92

[105] M.K. QURESHI, D. THOMPSON, AND Y.N. PATT. **The V-Way Cache: Demand Based Associativity via Global Replacement**. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 544–555, Washington, DC, USA, 2005. IEEE Computer Society. 13

[106] N. RAFIQUE, W-T. LIM, AND M. THOTTETHODI. **Architectural support for operating system-driven CMP cache management**. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 2–12, New York, NY, USA, 2006. ACM. 13

[107] N. RAFIQUE, W-T. LIM, AND M. THOTTETHODI. **Effective Management of DRAM Bandwidth in Multicore Processors**. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 245–258, Washington, DC, USA, 2007. IEEE Computer Society. 25

[108] S. RIXNER, W.J. DALLY, U.J. KAPASI, P. MATTSON, AND J.D. OWENS. **Memory Access Scheduling**. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*. IEEE Computer Society, 2000. 18, 23, 52, 111

[109] K. SANKARALINGAM, R. NAGARAJAN, H. LIU, C. KIM, J. HUH, D. BURGER, S.W. KECKLER, AND C.R. MOORE. **Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture**. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 422–433, New York, NY, USA, 2003. ACM. 11

[110] A. SAULSBURY, T. WILKINSON, J. CARTER, AND A. LANDIN. **An argument for simple COMA**. In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, HPCA '95, pages 276–, Washington, DC, USA, 1995. IEEE Computer Society. 11

[111] L. SEILER, D. CARMEAN, E. SPRANGLE, T. FORSYTH, M. ABRASH, P. DUBEY, S. JUNKINS, A. LAKE, J. SUGERMAN, R. CAVIN, R. ESPASA, E. GROCHOWSKI, T. JUAN, AND P. HANRAHAN. **Larrabee: a many-core x86 architecture for visual computing**. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, 2008. 68, 76

[112] A. SNAVELY AND D.M. TULLSEN. **Symbiotic jobscheduling for a simultaneous mutlithreading processor**. *SIGPLAN Not.*, **35**(11):234–244, 2000. 38

[113] V. SOUNDARARAJAN, M. HEINRICH, B. VERGHESE, K. GHARACHORLOO, A. GUPTA, AND J. HENNESSY. **Flexible use of memory for replication/migration in cache-coherent DSM multiprocessors**. In *Proceedings of the 25th annual international symposium on Computer architecture*, ISCA '98, pages 342–355, Washington, DC, USA, 1998. IEEE Computer Society. 11

[114] E. SPEIGHT, H. SHAFI, L. ZHANG, AND R. RAJAMONY. **Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors**. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 346–356, Washington, DC, USA, 2005. IEEE Computer Society. 13

[115] S. SRIKANTAIAH, M. KANDEMIR, AND M.J. IRWIN. **Adaptive set pinning: managing shared caches in chip multiprocessors**. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural sup-*

port for programming languages and operating systems, pages 135–144, New York, NY, USA, 2008. ACM. 13, 16, 17, 92

[116] S. SRINATH, O. MUTLU, H. KIM, AND Y.N. PATT. **Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers**. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 63–74, Washington, DC, USA, 2007. IEEE Computer Society. 28

[117] P. STENSTRÖM, M. BRORSSON, AND L. SANDBERG. **An adaptive cache coherence protocol optimized for migratory sharing**. In *Proceedings of the 20th annual international symposium on Computer architecture*, ISCA '93, pages 109–118, New York, NY, USA, 1993. ACM. 13

[118] H.S. STONE, J. TUREK, AND J.L. WOLF. **Optimal Partitioning of Cache Memory**. *IEEE Trans. Comput.*, **41**:1054–1068, September 1992. 12

[119] K. STRAUSS, X. SHEN, AND J. TORRELLAS. **Uncorq: Unconstrained Snoop Request Delivery in Embedded-Ring Multiprocessors**. *MICRO-40: 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2007. 10, 92

[120] J. STUECHELI, D. KASERIDIS, D. DALY, H.C. HUNTER, AND L.K. JOHN. **The virtual write queue: coordinating DRAM and last-level cache policies**. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, pages 72–82, New York, NY, USA, 2010. ACM. 23, 24

[121] R. SUBRAMANIAN, Y. SMARAGDAKIS, AND G.H. LOH. **Adaptive Caches: Effective Shaping of Cache Behavior to Workloads**. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 385–396, Washington, DC, USA, 2006. IEEE Computer Society. 13

[122] G.E. SUH, L. RUDOLPH, AND S. DEVADAS. **Dynamic Partitioning of Shared Cache Memory**. *J. Supercomputing.*, **28**(1):7–26, 2004. 13

[123] D. TARJAN, S. THOZIYOOR, AND N.P. JOUPPI. **Cacti 4.0**. Technical report, HP Labs Palo Alto, June 2006. 37, 39, 41, 120

[124] TECHINSIGHTS. **Micron Technology MT41J128M8JP-187E:F**. Technical report, http://www.ubmtechinsights.com/, 2009. 113

[125] J.M. TENDLER, J.S. DODSON, J.S. FIELDS, H. LE, AND B. SINHAROY. **POWER4 system microarchitecture**. *IBM J. Res. Dev.*, **46**:5–25, January 2002. 11

[126] R.M. TOMASULO. **An efficient algorithm for exploiting multiple arithmetic units**. *IBM J. Res. Dev.*, **11**:25–33, January 1967. 1

[127] S. VANGAL, J. HOWARD, G. RUHL, S. DIGHE, H. WILSON, J. TSCHANZ, D. FINAN, P. IYER, A. SINGH, T. JACOB, S. JAIN, S. VENKATARAMAN, Y. HOSKOTE, AND N. BORKAR. **An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS**. In *ISSCC '07: IEEE International Solid-State Circuits Conference*, 2007. 3, 32

[128] K. VARADARAJAN, S.K. NANDY, V. SHARDA, A. BHARADWAJ, R. IYER, S. MAKINENI, AND D. NEWELL. **Molecular Caches: A caching structure for dynamic creation of application-specific Heterogeneous cache regions**. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 433–442, Washington, DC, USA, 2006. IEEE Computer Society. 13

[129] B. VERGHESE, S. DEVINE, A. GUPTA, AND M. ROSENBLUM. **Operating system support for improving data locality on CC-NUMA compute servers**. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, pages 279–289, New York, NY, USA, 1996. ACM. 11

[130] B. VERGHESE, A. GUPTA, AND M. ROSENBLUM. **Performance isolation: sharing and isolation in shared-memory multiprocessors**. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VIII, pages 181–192, New York, NY, USA, 1998. ACM. 11

[131] D.T. WANG. *Modern dram memory systems: performance analysis and scheduling algorithm*. PhD thesis, College Park, MD, USA, 2005. Chair-Jacob, Bruce L. 19

[132] H-S. WANG, X. ZHU, L-S. PEH, AND S. MALIK. **Orion: a power-performance simulator for interconnection networks**. *MICRO-35: 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 294–305, 2002. 39, 42

[133] T. WATANABE, K. AYUKAWA, S. MIURA, M. TODA, T. IWAMURA, K. HOSHI, J. SATO, AND K. YANAGISAWA. **Access optimizer to overcome the ldquo;future walls of embedded DRAMs rdquo; in the era of systems on silicon**. In *Solid-State Circuits Conference, 1999. Digest of Technical Papers. ISSCC. 1999 IEEE International*, pages 370 –371, 1999. 18

[134] R.P. WEICKER AND J.L. HENNING. **Subroutine profiling results for the CPU2006 benchmarks**. *SIGARCH Comput. Archit. News*, **35**(1):102–111, 2007. 47

[135] W. WINSTON. **Optimality of the Shortest Line Discipline**. *Journal of Applied Probability*, **14**(1):181–189, 1977. 26

[136] L. YEN, J. BOBBA, M.R. MARTY, K.E. MOORE, H. VOLOS, M.D. HILL, M.M. SWIFT, AND D.A. WOOD. **LogTM-SE: Decoupling Hardware Transactional Memory from Caches**. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 261–272, Washington, DC, USA, 2007. IEEE Computer Society. 12

[137] G.L. YUAN, A. BAKHODA, AND T.M. AAMODT. **Complexity effective memory access scheduling for many-core accelerator architectures**. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 34–44, New York, NY, USA, 2009. ACM. 19, 23, 29

[138] M. ZHANG AND K. ASANOVIC. **Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors**. *ISCA '05: 32nd Annual International Symposium on Computer Architecture*, pages 336–345, June 2005. 11, 13, 17, 60, 92

[139] Y. ZHANG, D. PARIKH, K. SANKARANARAYANAN, K. SKADRON, AND M. STAN. **Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects**. Technical report, University of Virginia Dept. of Computer Science, March 2003. 40

[140] Z. ZHANG AND J. TORRELLAS. **Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA**. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, HPCA '97, pages 272–, Washington, DC, USA, 1997. IEEE Computer Society. 11

[141] Z. ZHU, Z. ZHANG, AND X. ZHANG. **Fine-grain Priority Scheduling on Multi-channel Memory Systems**. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 107–, Washington, DC, USA, 2002. IEEE Computer Society. 18, 23, 26

[142] W.K. ZURAVLEFF AND T. ROBINSON. **Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order**. In *U.S. Patent Number 5,630,096*, May 1997. 18, 23, 111