Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

Adaptive Cache Sharing for Future Many Core Architectures

by Pau Erola Cañellas

Advisors
Grigorios Magklis
Antonio González

January 26, 2009

Masters thesis submitted in partial fulfillment of the requirements for the Master of Computer Architecture, Networks and Systems degree

This work was supported by the Spanish Ministry of Education and Science and FEDER funds of the EU under contracts TIN 2004-03072, and TIN 2004-07739-C02-01, the Generalitat de Catalunya under grant 2005SGR00950, Intel Corporation and Obra Social “la Caixa”.

Index

1. Introduction

1.1. The manycore era

More than forty years ago Gordon Moore predicted that the number of transistors that can be inexpensively placed on an integrated circuit will be increasing exponentially, doubling approximately every two years. This trend is not expected to stop for a decade at least. In computer architecture, however, the correlation between this increment of transistors and the scalability of the performance (computational capacity) of single-core CPUs is limited by power and thermal issues [6], such as dissipating heat from increasingly densely packed transistors, that limit the rate at which processor frequency can be increased (see figure 1.1). A shift to new architectures is needed.

To reduce the power consumption we can use a number of interconnected smaller cores on the same chip, a *multicore* architecture, that run at lower clock frequencies. But the bandwidth request of a multicore system scales linearly to the number of cores putting high pressure on the memory system. The widening gap between the memory and the processor has motivated the study of new memory architectures.

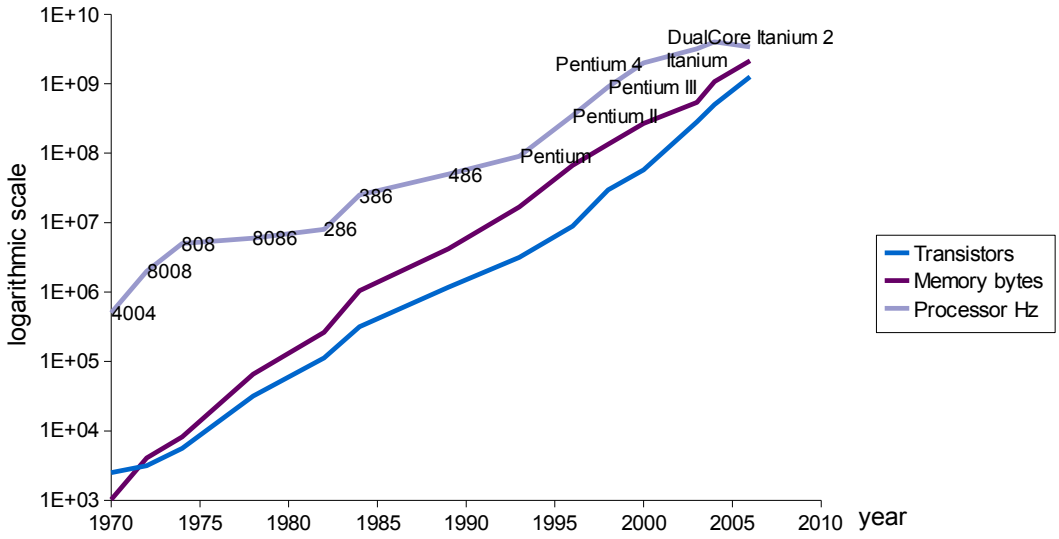


Figure 1.1: Power and thermal issues do not allow to increase more the clock frequency justifying the shift to multicore architectures. In the chart we can observe the slowdown in processors frequency during the last years. Figure elaborated with data available at *intel.com*.

Scaling multicore architectures up to hundreds of cores, *manycore* architectures, creating platforms capable of performing trillions of calculations per second on terabytes of data, will bring us to the era of *terascale* computing. For the Intel Terascale group [15] the key attributes that will be required for future terascale platforms are the programmability, the adaptability and the scalability amongst others. New multicore platforms need optimized software to live up to their potential and must be able to reconfigure themself to match varied usage and workloads. The problem of scalability will be solved with a bottleneck free architecture, this is: a good interconnection network and a workload-balanced cache hierarchy.

Designing an operating system for this new architectures is also a challenge. Where to run processes,

adding support for a new memory mechanism, or performing management of resources in software to reduce area overheads of the control hardware are only few of the currently open research problems new research interests.

In this first section we present some of the proposed solutions for these future architectures.

Memory die stacking

Computational requirements increase day after day, applications have bigger footprints and the off-chip bandwidth requests are increased. To this, we have to add the increasing pressure on the memory hierarchy due multicore architectures. How to provide a power-efficient high bandwidth DRAM access is an important challenge.

	Off-Package	**3D Stacked Die**
I/O Speed	4 Gb/s	10+ Gb/s
# of Connections	4-6 DDR	10,000+
Bandwidth	<100 GB/s	>1 TB/s
Capacity	>1 GB	<512 MB
Interconnect length	>100 mm	1 mm

Table 1.1: 3D Stacking capabilities compared with current off-package technology which has more limited pins and bus-speed. Data from Intel Technology Journal [31].

In traditional planar technology the transistors are in a single 2-D plane. New stacked integration technologies allow us to have several planes one over other. Emerging 3D technology provides a higher integration density up to the number of planes with the reduction in the clock distribution and wires length. Table 1.1 resumes the capabilities of 3D stacked memory compared with current off-package technology. Moreover, as we connect stacked planes with vertical vias, the technology used in each plane may differ, making possible to use the better technology for each plane. On the other hand, thermal dissipation is a problem associated with 3D stacking, stacked hot spots scales power and heat density [30].

Exploration of 3D-stacking capabilities can be seen in Black [5][4] and Mysore [30] works. They divide a commercial processor floorplan into smaller plans that are stacked. Results of Black for two plans show a performance improvement of 15%, due to the reduction of pipeline cycles, and a similar power reduction. If we also stack a memory plane we can achieve a 15% reduction in memory access time. Mysore results also show the dissipation problems associated with stacked hot spots.

Puttaswamy and Loh [32] have presented a study in cache partitioning. This study has demostrated that 3D stacking may be used to implement faster and low power on-chip caches due to the reduction of interconnections. If we divide the cache into banks we can increase the memory size improving the processor performance when it runs memory intensive applications. If we split the cache into arrays we reduce the wire length.

Interconnection networks

The next generation of interconnets among cores and caches must take into account many interacting factors. Each interconnection topology has its own tradeoffs in throughput, latency, resource occupation, and ease of implementation, making it suitable at a specific level in the cache hierarchy.

Four are the basic interconnection topologies: buses, crossbars, tiny-networks, and rings. The buses have a high bandwidth but suffer from a high number of collisions. The crossbars have very low latencies but have important costs in area occupancy. The tiny-networks and rings are simple and have a low area occupancy at the expense of a high latency. Moreover, when dealing with cache coherence

we must consider transactions within and across chips.

Kumar, Zyuban and Tullsen [22] studied the overhead and scalability of interconnection topologies in CMPs. Their results show that the architecture of the interconnect interacts with the design and architecture of the cores and caches, and in bad designs the interconnection can take important amounts of area and power. While shared last-level caches improve cache hit rates, the implications on the interconnect can be extreme, requiring importants amounts of area just for the network.

In the next generation architectures we can not use a bus snoop broadcast protocol that can not be scaled and have a big area and power consumption. The Intel Corporate Technology Group [2] identified that some of the basic requirements for the on-die interconnections are scalability, partitionability, regularity and flexibility. Candidate topologies will be determined by the wiring density and router complexity, interconnection must be power efficient and its area negligible. The 2D torus/mesh networks are the best candidates for the terascale computing. The 2D mesh perfectly matches the silicon surface and is easy to implement, but a number of limitations have been proved especially for long distance traffic.

Cache memory systems

Jacob, Ng and Wang [20] recognize that research in cache design is by no means at a stand still, and the future holds numerous avenues of interesting work. New architectures, with multiple cores and on-die last-level caches, open the interest on exploring significantly more sophisticated algorithms for content and consistency management.

In future manycore architectures, due to the increasing pressure on the memory hierarchy and the variability of the workload demands, efficient cache hierarchies and coherence protocols are demanded. Their efficiency will be determined by their flexibility and scalability. In a cache hierarchy several key factors must be take into account: the organization of the cache structure, the number of cache levels, the size of each cache level, the associativity, the latency, the allocation, and the eviction protocol. The main goal of tuning all these factors is to reduce the access latency. Also important in current systems is the energy-efficiency. Setting the cache hierarchy and the number of cores a priori in a CMP architecture will result in poor performance across many application classes.

A space exploration of tradeoffs of future CMPs has been done by Huh, Burger, and Keckler [19]. In this study they analyze the CMP organization in accordance to area and performance trade-offs. They model in-order and out-of-order processor organizations with off-chip limited bandwidth and a limited pin count according to the SIA Roadmap for different integration technologies with a fixed area. The results show that as technology factor scales down the amount of memory needed is bigger due to the increasing memory gap.

As multicore processors become pervasive, a key design issue facing processor architects will be the hierarchy and policies for the last-level cache. This forces us to take into account if cores have to share their cache memory space or not. Sharing the caches among all cores can provide poor isolation, but cores view a bigger cache memory.

In accordance with [7], when we are sharing a cache memory amog cores, consider we have N processors, each processor x emits circular sequences of memory accesses (a sequence of memory accesses where the first and the last accesses are to the same line address) to the same set denoted by $cseq_x(d_x,n_x)$ where

n_x is the number of cache accesses mapped to the same set, and

d_x the number of distinct lines addressed mapped to the same set.

Let consider A the cache associativity. Running alone, in a *A-way* associative cache with LRU replacement policy, we will have a miss if $d_x > A$, this is, if we have a circular sequence longer than the cache associativity. If we have multiple co-runners sharing an *A-way* cache we have that processor

x will have a miss if $d_x + \sum_{i=0; i \neq x}^{N} d_i > A$, where $\sum_{i=0; i \neq x}^{N} d_i$ always is a non negative value. Due this, sharing a same-sized cache memory we will have equal or more miss probability, this is, equal or bigger miss rate.

If we decide to share our cache memory space some workloads could perform better, but we have co-runners and the pollution between applications can degrade our performance. As our final objective is reduce the accesses latency in all the possible workloads, an adaptive technique will be the best decision.

Cooperative Caching is the technique proposed by Chang and Sohi [8] that tries to maximize the L2 usage to reduce the off-chip accesses creating a globally-managed logical shared L2 with private caches. A distributed version of these scheme was proposed by Herrero *et al.* [18]. Dybdahl and Stenstrom [11] designed an adaptive NUCA cache partitioning scheme that tries to maximize the last-level cache performance sharing caches with a dynamically controlled quota mechanism.

Different workloads performs better with different replacement policies. Adaptive techniques have been presented by Subramanian and Loh [42] and Qureshi and Patt [34]. Their idea is to use two cache policies and switch between them. They achieve up to 12% performance improvement respect the LRU policy.

Traditionally the cache has been a transparent resource for the OS and no much improvements can be done. Some new techniques to manage the cache memory via the operating system have been proposed. Rafique, Lim and Thottethodi [35] worked on an OS cache management technique. Their idea is to manage shared caches with variety of policies using a hardware cache quota mechanism controled by the OS. The Quota Enforcement Mechanism keeps the ownership of each cache block over where we have priority and the OS manages a Sharer Quota Table which stores quota of each sharer.

Cho and Jin [10] presented a new page allocation mechanism. As the OS can determine the virtual page number of a set of data or code, with a modified architecture that assign the resources according with the virtual page number we can manage the placement of our data or code. This allows us to make virtual multicores and use the cache of idle processors.

Fair Scheduling [13] is an OS scheduling developed by Fedorova *et al.* that tries to reduce co-runner-dependent variability in the performance of the applications. It estimates the L2 cache miss rate and assigns bigger time slices to applications with performance degradation due poor isolation.

Transactional memory

Since the first multiprocessors appeared, guaranteeing the memory coherence and consistency [24] has been a recurring problem. A parallel programming model must take into account the work balance, the dependency control, and the communication, and provide an easy-to-use framework. But most models have problems associated with conventional locking techniques or they need an expert programmer.

In 1993 Herlihy and Moss [17] presented a new mechanism that achieves a high abstraction for concurrent accesses to shared memory, the *transactional memory,* that uses a new set of instructions allowing easy-to-use non-blocking synchronization. Accesses to shared memory regions are performed as transactions (a sequence of machine instructions that preserves the atomicity of the operations) that protect this data until the transaction commits.

Two years later, Shavit and Touitou [39] presented a new implementation of transactional memory fully base on software. Using `Load_Linked` and `Store_Conditional` operations it supports full true system execution with scalability but had bad performance.

McDonalt *et al.* [27] developed a new hardware transactional memory with full support for modern OS and programming models, supporting nested transactions. Their model can achieve higher speedups

than the first transactional memory model. Since then, hybrid hardware/software transactional memory mechanisms have been proposed that try to achieve the best of both implementations. Kumar *et al.* [23] developed a new scheme that uses hardware structures until these are exhausted, then it switches to a software based transactional memory model maintaining the scalability of the model. Moore *et al.* [29] worked on a fast version of transactional memory, LogTM, based on an extension of the MOESI directory protocol. This accelerates transactions by hardware, but keeping the aborts management in software. To reduce the control traffic Hammond *et al.* [14] presented Stanford TCC that combines all writes from each transaction region into a single packet reducing the amount of bandwidth required.

Saha *et al.* [38] implemented a scalable software model with architectural support, HASTM, that marks transactions in the cache memory. The model presented by Shriraman *et al.* [40], RTM, has a similar key idea: use a software model accelerated by hardware, in this case using an extension of the MESI coherence protocol.

Ramadan and Rossbach *et al.* [37][36] worked in a real implementation of software transactional memory in a Linux kernel, MetaTM/TxLinux. Based on a stack mechanism, this transactional memory simplifies the large kernel, that uses diverse synchronization primitives, without loss of performance.

1.2. Objectives

After this wide overview of hot techniques applied to solve the main key attributes of the terascale computing, we will be specific about our objectives.

As the number of cores increases we could expect that our applications run faster and process more data, but the higher demands to the memory hierarchy and to the interconnects limits the system scalability.

Taking more cores we will have more memory references requiring higher throughputs to access to the memory hierarchy. The interconnections must be aware of this, in addition to the needs of the coherence protocols, and of the size, thermal, and power constrains.

To achieve a good cache hierarchy performance we must take into account several factors: the number of cache levels, the policies at each level, the cache sizes, the set-associativity, the cache block size, the number of cores that share the cache space, the sharing degree, the coherence protocols... And all these factors under size, thermal, and power constrains. Determine the optimal cache hierarchy design is not easy, and no single solution provides the best performance for all the workloads.

The exploitation of the last-level cache is especially important. This is usually shared among all cores because of the better use of memory space. Unfortunately, share the cache memory can produce pollution between applications, resulting in performance degradations. As no single solution provides the best performance, adaptive private/shared caches will provide the best of both implementations.

The sharing degree of our last-level cache can change the overall performance significantly. Having high-degree shared caches in a multicore architecture does not have any advantage, increasing the hit latency with an insufficient reduction of misses. Dybdahl *et.al.* [11] propose an adaptive partitioning scheme that limits the number of ways that each cache can use. This proposal maintains a private partition per core, making it interesting due to its controlled sharing degree. However, this technique has limitations to be implemented with good scalability.

In this thesis we present a study of the Dybdahl's scheme and its scalability in 4-, 8- and 16-cores architectures. Observing its problems, we have proposed a novel distributed adaptive scheme that tries to solve the scalability limitation. For our proposal, moreover, we have studied several performance implications due to its distributed nature.

The rest of this document is organized as follows. The next chapter presents an overview of cache memories. The chapter 3 revises the related work. The baseline architectures used for this study are presented in chapter 4. Chapter 5 describes the Dybdahl's scheme and how we implement it. The next

chapter presents our proposal. In chapter 7 we describe our methodology and we discuss about the metrics used. The results of our simulations are presented in chapter 8. We conclude in the next chapter.

2. Background: the cache memory

2.1. Overview of the cache memories

The cache is the highest or first level of the memory hierarchy encountered once the address leave the processor. Because of locality and the higher speed of smaller memories, caches can substantially improve the performance reducing the increasing memory gap, the gulf in access time and capacity, between the processor and the memory.

Transparently addressed caches are the most common form of general-pourpose processor caches due their reasonable improvement of performance. These cache memories store chunks of 2^B bytes of data called cache blocks. When the processor requests an address it first checks whether that memory location is in the cache comparing the upper address bits of the memory location to all tags in the same cache set (see Figure 2.1). The cache set where a block is mapped is determined by the bits *b+1* to *b+n*. When *n* equals the logarithm of the number of blocks in the cache, we have a direct-mapped cache; when it is 0, we have a fully associative cache; in intermediate cases, we have a *n*-way set associative cache.

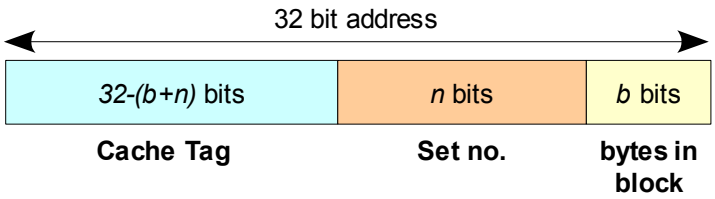


Figure 2.1: To identify a block in the cache we need to find the tag bits in the set determined by the set bits of the address requested.

When the processor finds a requested data item in the cache, it is called a *cache hit*. When it doesn't find the data, a *cache miss* occurs. Cache misses occur because data is accessed for first time, cumpulsory misses; because the cache is not large enough to hold the working set, capacity misses; because we hace more requests than the associativity degree, conflict misses; or due flushes to keep coherency in multiprocessors.

Cache Organization

The fundamental trade-off between cache latency and hit rate, larger caches have better hit rates but longer latency, motivates the use of multiple levels of cache.

Multi-level caches introduce new design trade-offs. If all data in the L1 cache must also be somewhere in the L2 cache we have a *strictly inclusive* cache. These caches are simple to implement and facilitate the coherence protocols design. Moreover they allow the use of different cache lines which reduce the size of the cache tags. If we decide that data is guaranteed to be in at most one of the L1 and L2 caches, never in both, we have an *exclusive* cache. The advantage of exclusive caches is that they can store more data. *Non-inclusive* caches attempt to achieve the best of both worlds. A non-inclusive cache does not require the sustaining of inclusive property as an absolute requirement.

Management of cache contents

When a cache miss occurs, the data retrieved from the main memory is copied into the cache, forcing a

block deletion to make room. The state-of-the-art processors employ various policies such as the LRU and pseudo-LRU, the least-recently used block is removed from cache; the random policy, the block is chosen randomly; the FIFO policy, the block that was brought in the first time is replaced; and the MRU policy, the most-recently used block is replaced. The LRU policy takes locality into account and have good performance at the expense of cost and complexity of the hardware. The random algorithm reduce the implementation cost and it has the advantage that candidate blocks are spread uniformly across the entire cache without favoring any one area, but it ignores principle of locality resulting in lower performances. Various pseudo-LRU heuristics, a tree-based approximation of the LRU, have been proposed to reduce the hardware cost maintaining a good performance.

Heuristic	Storage requirements	Action on cache hit	Action on cache miss
random	$log_2(ways)$	None	Update the LFSR register
FIFO	$nsets \cdot log_2(ways)$	None	Increment the FIFO counter
LRU	$nsets \cdot ways \cdot log_2(ways)$	Update the LRU stack	Update the LRU stack
pseudo LRU	$nsets \cdot (ways-1)$	Update the tree bits	Update the tree bits

Table 2.1: Pseudo-LRU policies reduce the hardware cost by approximating the LRU mechanism [1].

When data is written to a present cache block the information can be written to both the block in the cache and the block in the lower-level memory, it is a *write through* policy. In a *write back* policy the information can be written only to the block in the cache and mark this block as modified to write it to main memory when it is replaced.

When the write operation produces a miss, we can use two different policies: *write allocate* and *no write allocate*. In write allocate policy the block is loaded on a write miss, followed by the write-hit action. On no write allocate policy, the block is modified in the main memory and not loaded into the cache.

Management of cache coherence

In multiprocessors, including CMPs, caching shared data introduces a new problem because the view of memory held by two different processors is through their individual caches, which, without any additional precautions, could end up seeing two different values. We have to implement a protocol that grants write propagation, all writes must become visible at some time to other processes, and write serialization, all writes have to be seen in the same order by all. To begin, a cache must track the state of its contents, and states in which a block may find itself can be varied.

Using three cache states, the MSI protocol guarantees the coherence of our cache memory. When a block is requested, it goes to S (shared) state. If another processor requests the same block it remains in state S. When one of the processors writes to this block change the state to M (modified) and all the caches with shared copies have to invalidate them (state I).

To provide a lower overhead write mechanism, the E state (exclusive) is added. A processor may not write to a cache block unless it first acquires a copy of the block in E state. This protocol is known with the name MESI.

MSI and MESI protocols are implemented on write-back caches and all coherence activity passes

through a centralized point, the backing store. To make it easier to distribute the task of maintaining cache coherence, the O state (owned) is added. The MOESI protocol allows the holder of a modified block (M state) to transfer the block directly to another processor and to bypass the backing store completely, at which point the new processor places the block into the O state. Thus, the MOESI protocol is useful in systems that have a good client-to-client bandwidth like CMPs or NUCA organizations.

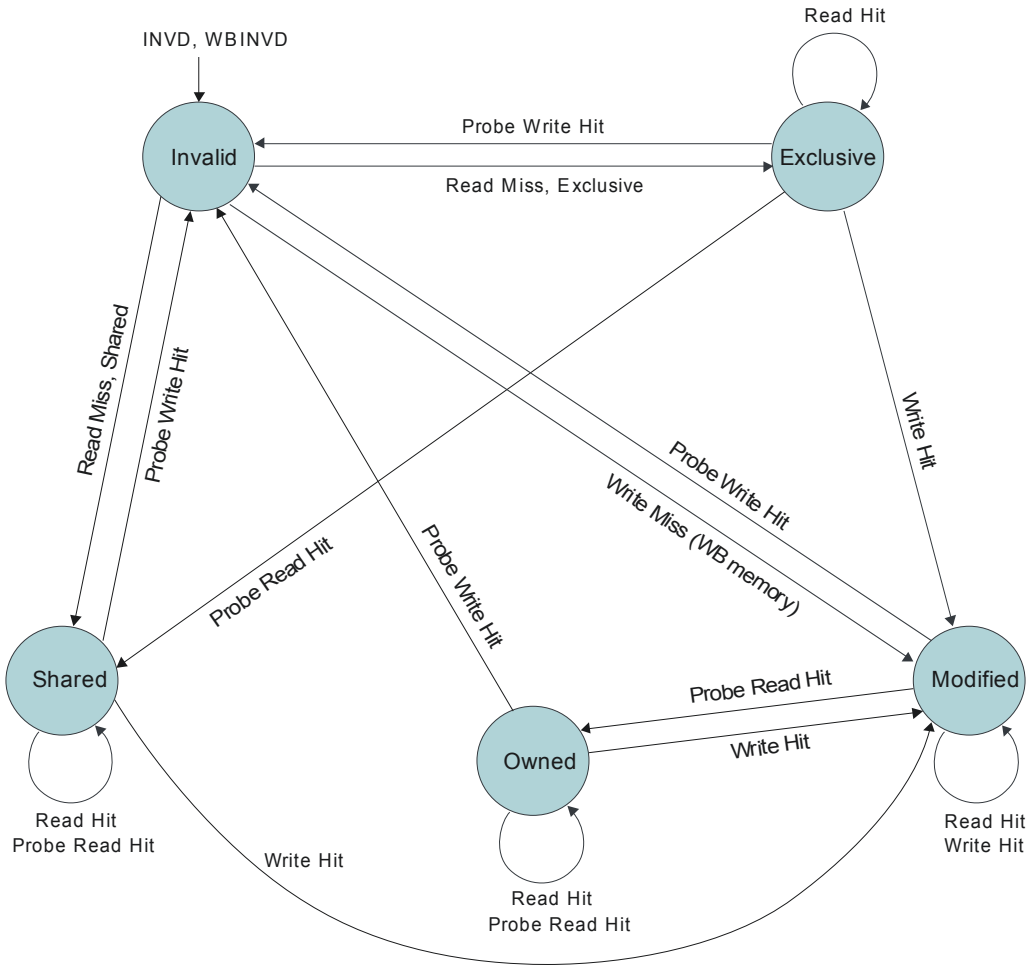


Figure 2.2: State diagram of MOESI protocol. MOESI effectively reduces memory traffic, increasing available bandwidth. Figure was extracted from AMD64 Architecture Programmer's Manual [44].

Cache performance

Considering the number of memory references and the hit latency constant values, the processor performance equation tells us that to reduce the computational time we must reduce the number of misses or the miss penalty.

$$CPU_{time} = (CPU_{cycles} + Memory_{stall\_cycles}) \times CycleTime$$

where

$$Memory_{stall\_cycles} = Instr \times Memory_{refs} \times Miss_{rate} \times Miss_{penalty}$$

Hennessy and Patterson [16] identified that to reduce the miss rate we can use a larger block size, a bigger cache, or use a higher associativity. To reduce the miss penalty we must use multilevel caches and give reads priority over writes. In this thesis, we are focused on the reduction of the number of misses using a virtual higher associativity.

3. Related work

3.1. NUCA caches

Before showing the recent work about adaptive caches, we think that is important to speak briefly about the NUCA caches because they are part of our framework.

In future technologies, large on-chip caches with a discrete hit latency will be undesirable, due to increasing global wire delays across the chip. Data residing in portions of cache closer to the processor could be accessed much faster than data that reside physically farther. The exploit of this non-uniformity was first proposed by Kim *et al.* [21].

NUCA cache architectures divide the whole cache into smaller banks using a two-dimensional switched network. Data is distributed among all the banks, so latency to obtain a cache block is determined by the request and response routing times (from the processor to the bank that contains the requested data) plus the latency of the bank.

When the mapping of data into banks is statically determined, based on the block index, and thus can reside in only one bank of the cache, we call this static NUCA or S-NUCA. Kim *et al.* determine that the channel area overhead is 5.9% of the total area of banks and results show an IPC improvement of up to 10% respect an unbanked cache.

Using the switched network, data can be gradually promoted to closer and faster banks as they are frequently used. We call this dynamic non-uniform cache D-NUCA. The main difficulties of this model turn up in the search algorithm. The data access policy applied in this schema can be incremental, with high latency, or use multicast, that consumes high bandwidth and power. Some approaches using extra partial tags can speedup the search significantly.

NuRAPID is the alternative proposed by Chishti *et al.* [9] which outperforms D-NUCA in power efficiency achieving similar performance results. In NuRAPID, cache tags are copied to the local node tag set and the data is replicated in a closer cache to reduce subsequent accesses.

3.2. Adaptive caches

The hybrid shared/private solutions that dynamically partition the cache memory according to the demand are a good solution to the problems of lack of isolation of shared memories.

To avoid the performance degradation in CMPs due poor isolation of working sets Speight *et al.* [41] proposed an adaptive mechanism. They supose a CMP architecture with a coherent L2 cache per core. Off-core cache accesses are managed by L3. The proposed mechanism manages write-backs limiting the unnecessary clean write-backs if another cache already has a valid copy. On a L2 write-back, the cache tries to move blocks to neighboring L2 caches that is faster compared to write the data to L3. Speight *et al.* also implemented a replacement policy that tries to kept L2 lines with high reuse potential. To identify these lines they use a table called Write Back History Table. On a replacement, first they choose invalid blcoks and blocks in shared state (because there are more valid copies).

Cooperative Caching is the technique proposed by Chang and Sohi [8]. It is similar to the technique proposed by Speight, and it tries to maximize the L2 usage, to reduce the off-chip accesses, creating a globally-managed logical shared L2 with private caches. In this case, they use a unified Central

Coherence Engine that keeps a copy of all the cache tags. To reduce off-chip accesses allows the cache-to-cache transfers of clean data, and implements a replication-aware replacement policy.

Recently Herrero *et al.*[18] have published a distributed version of Cooperative Caching with directory structures splited in smaller blocks that permits a better balance of network traffic avoiding bottlenecks. It outperforms the version of Chang and Sohi by 57%.

An adaptive NUCA cache partitioning scheme was designed by Dybdahl and Stenstrom [11]. This scheme wants to maximize the last-level cache performance sharing caches in a dynamically controlled way. The decision of how many cache ways are shared is done estimating the effects of increasing one block per set, counting the hits in last evicted blocks, and the effects of decreasing one block per set, counting the accesses to the LRU block in set. The sharing engine evaluates the gain and modifies the partitions every 2000 cache misses.

Patt and Qureshi [33] applies a similar idea in his Utility-based Cache Partitioning. The hits in different positions of the LRU stack tell us the performance we will achieve with a N-way cache partition. Its goal is to find the best partitioning using these information. They evaluate this idea on a dual core architecture with an average 11% speedup over LRU-based cache.

SP-NUCA [28] is a recently dynamic cache partitioning technique according to the access pattern. Using an extra bit per block (private/shared) and adjusting the replacement policy they divide the last-level cache into private and shared blocks, and place private data closer to its owner. SP-NUCA improves S-NUCA by 16%.

Beckmann, Marty and Wood [3] presented a mechanism to replicate data to limit the delays due global wires and minimize the cache access time. Their mechanism, ASR, monitors the workload behavior and replicates the blocks when the benefit of replication exceeds the cost. To estimate the benefits it counts the hits on remote L2 caches, and to estimate the costs it counts the hits on the LRU blocks. ASR outperforms shared caches by 29% and provides performance stability.

Replacement policies must guarantee that reusable data are kept in the cache hierarchy to reduce the memory latency. Problem with replacement policies is that different workloads perform better with different policies. As in any workload variability problem the solution is the adaptability. Subramanian and Loh [42] designed an adaptive processor cache manager to mitigate the effect of different memory access behaviors. Their idea uses two cache policies, LRU and LFU, and switches between them. The switch between policies has very little penalties and the proposal achieves the better performance of both policies. In a 512KB L2 they achieve a 12% performance improvement respect a LRU-based cache. Another adaptive replacement policy is the MLP-Aware replacement proposed by Qureshi and Patt *et al.* [34]. Their model uses a tournament selector to switch between a new memory level parallelism-aware policy (MLP) and a LRU policy.

3.3. Cache memory hierarchy in commercial CMP architectures[1]

Intel Core 2 Quad architecture (Yorkfield)

Yorkfield XE processor was the first Intel's consumer 64-bit multicore processor to use 45 nm technology and high-k metal gates. Yorkfield features a dual-die 3GHz quadcore design with two unified L2 caches of 6 MB each.

1 The specifications of processors presented were obtained mainly from the sites *intel.com*, *amd.com*, *ibm.com and sun.com. The date of release, numer of transistors, die area, L1 cache and memory bandwidth values were mostly obtained from wikipedia.org*, *tomshardware.com* and *hardwarezone.com*

Model	QX9650
Release date	November 11, 2007
# of Cores	4
CPU clock	3 GHz
Process Technology	45nm
# of Transistors	820 million
Die area	2x107 mm²
L1 Cache	32KB data + 32KB instruction
L2 Cache	2x6MB unified with Intel Smart Cache
Memory Bandwidth	21 GB/s

Table 3.1: Intel Core 2 Quad (Yorkfield) QX9650 technical specifications.

The cores in Yorkfield share, between two, two 6MB unified L2 caches that are managed with Intel Smart Cache technology. Smart Cache allows an adaptive dynamic cache partitioning, which means the L2 cache is constantly and dynamically adjusted to match the data load of each core and maximize L2 utilization.

AMD Opteron Quad-Core architecture (Shanghai)

The new AMD Shanghai processors are build on the foundation laid by the AMD Barcelona processors with some key technology advancements. Shanghai features a native quad-core architecture and the powerfull Barcelona's unique on-die L3 cache design, and brings additional enhancements for software developers.

Model	2384
Release date	November 13, 2008
# of Cores	4
CPU clock	2.7 GHz
Process Technology	45nm
# of Transistors	758 million
Die area	240 mm²
L1 Cache	64KB data + 64KB instruction
L2 Cache	4x512KB private
L3 Cache	6MB unified on-die, 48-way
Memory Bandwidth	17.6 GB/s

Table 3.2: AMD K10 (Shanghai) 2384 technical specifications.

Shanghai processors have local private L1 and L2 caches and a larger L3 cache, respect Barcelona processors, due to use of 45nm technology. Caches are completely non-inclusive, in exception of L3 that can behaves as an inclusive cache by keeping a copy of data when it is possible to be shared. Thus, the caches act as victim buffers for the caches higher up in hierarchy. L3 is dynamically shared between the cores, each one getting initially an equal share of the cache. The idle cores can lend their space to the other cores to increase their performance.

IBM Power6 architecture

The IBM's Power6 microprocessor moves from previous out-of-order designs to an in-order design. Presented at the IEEE International Solid-State Circuits Conference in 2006, processor is a dual core design and capable of two way SMT.

Release date	June 8, 2007
# of Cores	2, 2-way SMT
CPU clock	4.7 GHz
Process Technology	65 nm
# of Transistors	790 million
Die area	341 mm²
L1 Cache	64kB data + 64k instruction
L2 Cache	2x4MB, private with fast access to other cache
L3 Cache	32MB shared, off-die
Bus Bandwith	80 GB/s

Table 3.3: IBM Power6 Architecture technical specifications.

The Power6 has 128 KB of eight-way associative L1 cache and two 4 MB L2 cache, one for each core. The second level cache is not shared, but a core have fast access to neighbor's cache. The two cores share a 32 MB L3 cache which is off die.

Sun UltraSPARC T2 architecture (Niagara 2)

Sun Microsystems' UltraSPARC T2 microprocessor is the second generation of multicore and multithreaded SPARC processors. It supports concurrent execution of 64 threads by utilizing eight SPARC cores, each with eight hardware threads. Sun started selling servers with the T2 processor in October 2007.

Release date	October 2007
# of Cores	8, 4-way SMT
CPU clock	1.4 GHz
Process Technology	65 nm
# of Transistors	500 million
Die area	342 mm²
L1 Cache	8KB data + 16KB instruction
L2 Cache	4MB, 8 banks,16-way
Bus Bandwith	60 GB/s

Table 3.4: Sun UltraSPARC T2 (Niagara 2) technical specifications.

Each UltraSPARC T2 core has a 24KB L1 cache and all cores share a 4MB banked L2. The second level cache is 16-way associative and cores access it via a high bandwidth crossbar.

4. Baseline memory architecture

We define a baseline architecture as the starting point of this thesis. A manycore-capable configuration is needed, with good scalability and with regularity. We modeled the 4-, 8- and 16-cores CMP configurations that we can see in figures 4.1 and 4.2.

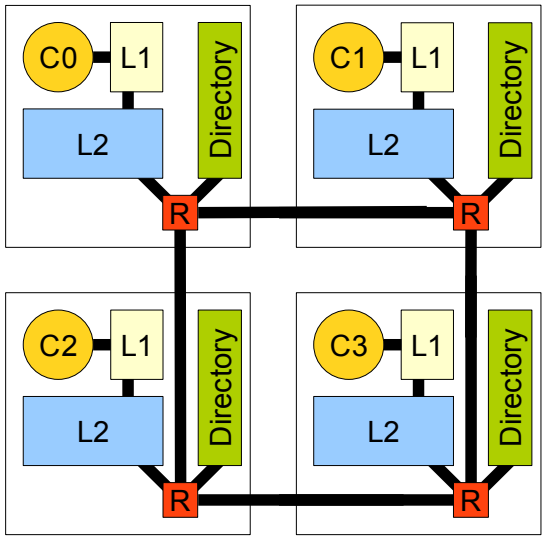


Figure 4.1: Representation of the baseline architecture with 4 cores. Our configuration is a CMP with two levels of cache and a distributed directory.

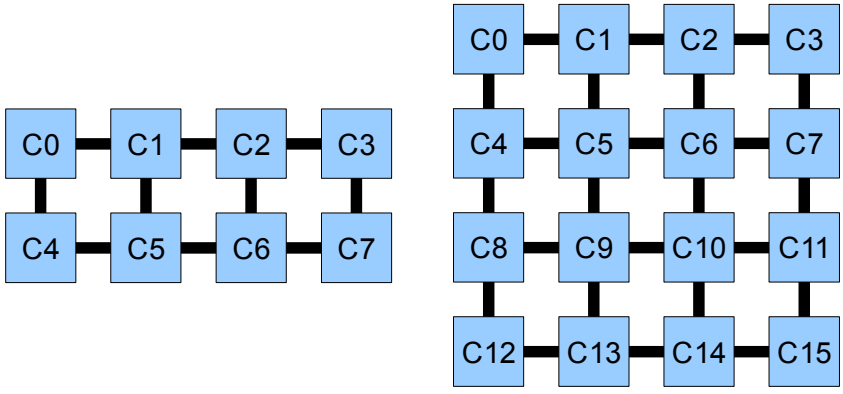


Figure 4.2: Simplified schema of the 8-cores and 16-cores configurations.

Each node of our CMP contains a processor with a private L1 cache that connects to a private or shared L2 cache bank. The L2 caches maintains its coherence using the MOESI protocol. The caches use pseudo-LRU replacement policy and are fully exclusive. The use of exclusive caches has been imposed by the GEMS simulator, we have no preference on it respect of an inclusive model.

To maintain the content of the whole cache we use a directory. It is distributed along the nodes and we address it with the lower order bits of the requested addresses. With a directory, we avoid having to search blocks using a broadcast that consumes high bandwidth and power.

To connect our nodes we chose a two-dimensional mesh network. Mesh networks have a regular structure and low area requirements making them suitable for the terascale architectures.

The cache sizes and access latencies, and other configuration details of the architectures proposed will be detailed in chapter 7.

5. Adaptive shared/private NUCA cache partitioning scheme

5.1. Adaptive partitioning scheme

How we have seen formerly, the applications that share a cache memory can suffer from pollution problems. As part of his thesis, Haakon Dybdahl, together with Per Stenström, had designed a dynamic partitioning scheme on a NUCA cache [11]. This system guarantees a minimum private area for the applications, yielding the rest as shared, but only when this sharing compensates the lose of private cache blocks. The main objective of the proposed technique is to minimize the total number of cache misses.

Based on an architecture with private last-level caches, the adaptive model sets a quota mechanism that determines dynamically the number of ways that each processor can use in a *set* (see figure 5.1). In this case, we consider a set all the sets in the last-level private caches with the same index. The *ways* that can be used by each processor are assigned dynamically estimating the benefits to assign one cache block to a processor from another processor.

The adaptive method also implements a replacement policy that allocates new data in private, and close and fast, partitions and moves the replaced data to the shared area.

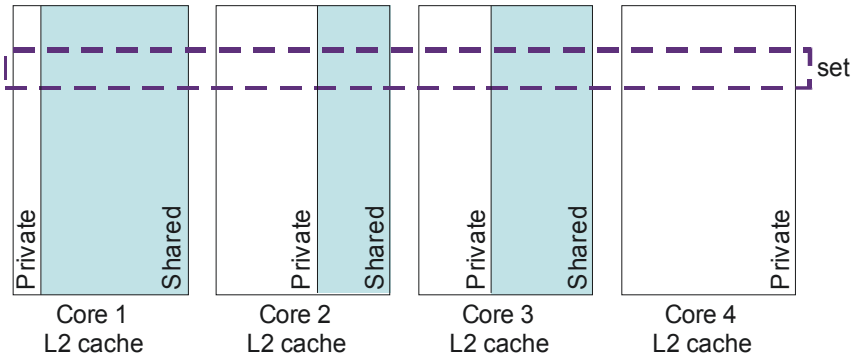


Figure 5.1: Schema of a 4-cores configuration with private and shared partitions defined. Dybdahl's technique names as *set* all the sets with the same index. The sharing of the last-level cache is done per way, the first ways in each bank are private and the rest shared.

Architectural framework

Dybdahl's work starts with an architectural framework of a four-core system. Each core has a three levels of cache, where the L3 is the last-level cache. L3 caches form a NUCA cache with fast access to the local L3 cache and a slower access time to neighboring caches. Local L3 cache has a latency of 14 cycles, and access to a neighboring cache has an extra penalty of 5 cycles. They assume that the caches are connected in a all to all topology and are self-coherent due they do not contemplate the possibility to share data.

Conceptually, all the last-level caches are connected to a centralized sharing engine that implements the proposed scheme.

To keep the processor owner of each block and to maintain the number of blocks that each processor

uses per set, the block entries in the last-level cache must be extended with the ID of the processor that owns the cache block. A cache block is represented in table 5.1.

Index	Tag	LRU data	Cache line data	Core ID
...	...	...	...	...

Table 5.1: Each core has a limited number of ways that can hold. To control this, each cache block is extended with a owner identificator.

Cache quotas

Each core has a limited number of ways that can hold. These maximums are determined by a repartition policy and they are stored on a set of registers as we represent in table 5.2. Every register indicates the maximum cache blocks that a processor can use per set.

Description	Core 0	Core 1	Core 2	Core 3
Max. # of blocks in set	3	4	2	3

Table 5.2: Registers with the maximum number of blocks in set per core for both the private partition and the shared partition.

When a processor has assigned less cache blocks than the associativity of the local last-level cache it uses only its private partition. If a block in the private partition is evicted, the processor is still allowed to place this block, but only one block, in the shared partition. This block has no warranties to keep for a long time due to it will be a candidate for the replacement policy.

If we have more blocks assigned than the associativity of the local last level cache, we can allocate our exceeding blocks in neighboring shared caches. In this case, a block will only be replaced if it is the head of the LRU stack.

As we use quotas, the update of the maximum number of ways in set per core will be carried out in a lazy form. When the repartition policy decides to modify the assignment of the ways, it only writes the maximum values per core in the registers. Who truly cares about the use of the ways is the replacement algorithm. It looks at the maximum registers and at the blocks' owner ID to limit the use of blocks in the set. If a processor losses one block the exceeding blocks that is using per set will be progressively replaced.

At the start up, the maximum number of blocks is assigned 75% of the local cache associativity. This first blocks act as a private cache whereas the rest are a contribution to the shared partition.

Estimating the effects of increase/decrease one block

The adaptive schema tries to minimize the total number of cache misses. For this, the mechanism estimates the benefits of increasing and decreasing one block per set.

To estimate the cache misses that would have been avoided if the number of ways assigned was increased by one we count the hits in the shadow tags table (see table 5.3). Shadow tags hold the last evicted block in set per processor. On a cache miss, if the address requested equals the address in the same set of the shadow tags table, it means that a miss could have been avoided with one more block in set. Then we increase by one the shadow tags counter.

The effect of decreasing one cache block per set is calculated counting the hits in the LRU stack head per core at any request. The LRU policy obeys that an access that hits in cache containing *N*-ways is guaranteed to also hit if the cache had more than *N*-ways. Monitoring the hits in the LRU head we

calculate the extra misses that will occur on a (*N-1*)-ways cache.

Set number	Core 0	Core 1	Core 2	Core 3
0	7ED1CE64	3514E5C	672A1D00	2ADD400
1	7DC95D88	3514E40	672A1FF8	2A4E708
...	...	...	...	...

Table 5.3: The shadow tags keep the last evicted blocks in set per core to estimate the effect of having one more cache block.

Counter	Core 0	Core 1	Core 2	Core 3
Hits in the LRU blocks	32	11	0	12
Hits in the shadow tags	24	0	9	54

Table 5.4: Counters maintained by the adaptive technique to estimate the effect of decreasing and increasing one cache block in set per core.

Comparing the hits in the LRU blocks and the hits in the shadow tags (see table 5.4) we can estimate the effects of removing a cache block of a processor and assign it to another processor. Thus, total number of cache misses that would occur with the a new partitioning.

Repartitioning policy

The adaptive technique needs to reevaluate the partition sizes per core on a regular basis. The reevaluation period needs to be long enough to measure the cache sensitivity and short enough to make the scheme dynamic. A period of 2000 cache misses seems to make sense.

The repartition algorithm (see algorithm 5.1) looks for the processor with a high reduction of cache misses with one more cache block, the one with more hits in the shadow tags, and compares it with the processor with less penalties with one cache block less, the one with less hits in the LRU counters. If the gain is higher than the loss, one cache block of the processor with the lower loss is provided to the core with the highest gain.

```
high_gain := MAX(hits_ShadowTags)
low_loss := MIN(hits_LRU)
if  high_gain > low_loss then
        maximum_blocks_in_set [high_gain] ++
        maximum_blocks_in_set [low_loss] --
end if
```

Algorithm 5.1: Every 2000 cache misses the partition sizes are reevaluated, if the gain is higher than the loss one cache block per set is provided to the core with the highest gain.

As the algorithm could leave without private cache blocks a processor, we need to define some constraints in the set of partitions. A processor will always have at least one private cache block and never more shared blocks that the private partition size.

Characterization of the NUCA model

A NUCA model can be characterized by defining the behavior of the data placement policy, the data migration policy, the data access policy and the data replacement policy.

The *access policy*, the data searching algorithm in the NUCA cache memory space, used in this technique involves a two phase process. First we look for a tag in the private cache partition. If there is no match, the rest of the set is checked, in other words, we check the private partition. To check the neighboring caches we assume that we use a broadcast request.

When a request produce a miss on the private partition of the local last-level cache and a hit on our shared partition or on the shared partition of a neighboring cache, the *migration policy* is allowed to change the data placement in the NUCA cache memory in order the access time in following accesses. The technique that we are studying exchanges the data in the shared partition with the LRU block of the local private partition. Thus, the most recently accessed block is moved closer the the processor and the least recently block of the private partition is evicted to the shared area.

The *placement policy* determines where a data element should be placed in the NUCA cache when it comes from the off-chip memory. In our case study, is clear that the new incoming data goes directly to our private partition. If the local private area has no free entries, a present block must be replaced.

The *replacement policy* is the algorithm in charge to determine determines how the NUCA cache behaves when we have no free cache entries to allocate a new block. On a cache miss, data loaded from memory is allocated in the private partition. If the private partition has no invalid blocks, a block must be selected to be replaced. Dybdahl's scheme replaces the block on the head of the LRU stack of the local private partition. We name this block *displaced*.

The displaced block will be send to the shared partition without go through a centralized backing store. If the shared partition has no invalid entries, the shared engine determines which block will be replaced. The replacement algorithm (see algorithm 5.2) used by the shared engine looks for the LRU block that belongs to a processor that is using more than the maximum blocks in set that has assigned. If none of the processors uses more than the assigned blocks in set, the LRU block of the set is evicted.

```
function Find_Block_to_Evict
        for LRU_stack_position := blocks_per_set, 1 do
                procID := owner_block(LRU_stack_position)
                if max_blocks_per_set[procID] < blocks_in_set(procID) then
                        return LRU_stack_position
                end if
        end for
        return LRU_block
end function
```

Algorithm 5.2: Algorithm for finding the block to evict; the function returns the position of the block.

Implementation cost

The implementation cost of the partitioning mechanism, due the extra storage, is estimated in 0.5% for the 4MB last-level cache baseline.

Let consider p the number of cores, s the number of sets, b the number of blocks, t the number of bits per tag, and w the number of bits of a register.

The owner identifier added per cache block cost $log_2\ p \times b$. The maximum blocks in set registers and the counters to maintain the hits in shadow tags and the hits in the LRU have a cost of $3 \times p \times w$.

The shadow tags require $s \times p \times t$ bits, however, shadow tags are not needed for all sets. Monitoring only the 6% of the sets is sufficient to estimate the cache sizes.

The total storage cost is then

$$log_2 p \times b \ + \ 3 \times p \times w \ + \ 0.6 \times s \times p \times t$$

5.2. Implementing Dybdahl's scheme in a 2-level exclusive cache

To implement the Dybdahl's scheme on our baseline architecture we need to do some modifications. Remember that our baseline uses two cache levels fully exclusive and caches are connected using a mesh network.

Meanwhile Dybdahl's mechanism assumes an externally monolithic NUCA cache where the sharing engine is part of the last level cache, our NUCA model will consider a sharing engine independent from the L2 cache memories. Our sharing engine will maintain a copy of cache block tags and the owner of each block. This decision is taken due to the physical impossibility to keep a global LRU stack using distributed last-level caches.

The sharing engine will be placed after the routing element as one more node as we can see in figures 5.2 and 5.3. All the caches will connect to it directly, via the routing element, but no traffic is allowed to use this wires in exception of the traffic explicitly directed to or from the sharing engine. Thus, the links to the sharing engine can not be used to route requests or responses in a short path.

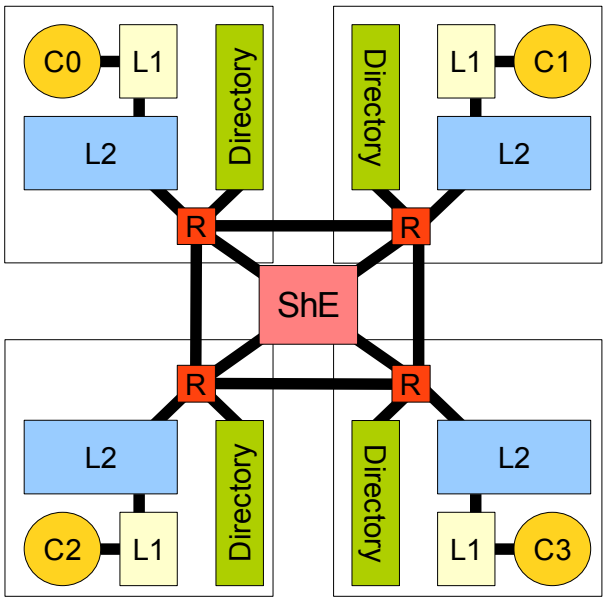


Figure 5.2: Proposed implementation for the Dybdahl's scheme. All the processors are connected to a centralized sharing engine that maintains a copy of the L2 tags and the globally LRU stack.

All the modifications in the L2 tags or the LRU stack will be send to the sharing engine. The cache block replacements and touches in the LRU stack will arrive at the sharing engine delayed due the routing latency. The lazy nature of this schema does not make this a problem.

The use of exclusive caches and a directory to maintain the coherence introduces changes in the

behavior of the NUCA cache. The exclusive cache reduces the need to exchange the data due to the absence of data movements between L2 cache blocks. The presence of the directory eliminates the need to broadcast a searching message to the private partitions.

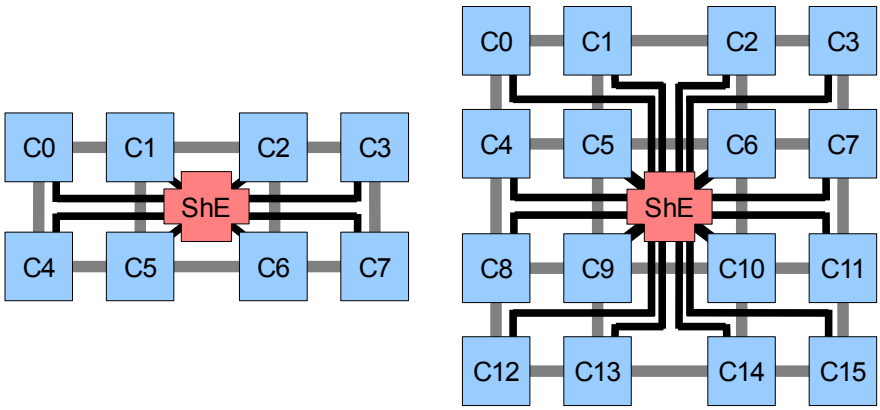


Figure 5.3: Simplified schema of the 8-cores and 16-cores configurations. Maintaining a centralized sharing engine breaks the regularity of our mesh network.

Characterization of the NUCA model

The *access policy*, the data searching algorith, used in this technique involves a two phase process. First we look for a tag in the local cache partition. Note that we unify the search doing it in both the private local partition and the shared local partition. This separation has only significance for power reduction and it is out the scope of this thesis. If there is no match, we check the directory for that block. The directory will forward our request to the owner cache or to the main memory.

When a request produces a miss on the private partition of the local last-level cache and a hit on the shared partition, the *migration policy* is allowed to change the data placement in the NUCA cache. As we are working with an exclusive cache, this migration involves a movement of data between a L1 cache and a L2 cache. The data in the shared partition goes through the local private partition and is allocated directly in the L1 cache. When this data is evicted from the L1 it will be written back to the local private L2 partition.

The *placement policy* has the same behavior described before, the new incoming data goes directly to our L1 cache. If the L1 cache has no free entries, a present block must be replaced and written back to the L2 cache. If the local private partition does not have invalid blocks we will replace the least recently used block. The replaced block will be send to the shared partition. Is the sharing engine the mechanism in charge to determine where to allocate the *displaced* block. If the shared partition has not available entries, the *replacement policy* looks for the LRU block that belongs to a processor using more than the maximum blocks in set that it has assigned. If none of the processors uses more than the assigned blocks in set, the LRU block of the set is evicted.

Repartitioning policy

To adapt the Dybdahl's scheme to 8- and 16-cores architectures we had to change the repartitioning period. We have chosen a proportional period to the 2000 misses of the 4-cores architecture, $500 \times number\_of\_cores$. For the 8-cores architecture the repartition period will be 4000 misses, and 8000 misses for the 16-cores architecture. We have tested experimentally smaller period values that were poorly significant and higher values that perform similarly but with slower adaptability.

We added explicitly to the partition algorithm the minimum and maximum number of blocks in set per cache (see algorithm 5.3). We thing that it is not fully obvious that if we do not apply this restriction during the selection of the processor with the higher gain and the processor with the lower loss our repartition algorithm will be stalled many times. This effect is produced because an application with

loss zero will be repetitively selected and then we could not take off his single private block. The same effect will occur with a processor with the highest gain that has the maximum number of blocks in set assigned.

```
high_gain := MAX_incrementable(hits_ShadowTags)
low_loss := MIN_decrementable(hits_LRU)
if  high_gain > low_loss then
        maximum_blocks_in_set [high_gain] ++
        maximum_blocks_in_set [low_loss] --
end if
```

Algorithm 5.3: Every repartition period, we compare the benefits of the core with less than the maximum blocks assigned and the higher gain with the waist of the core with more than one block and a lower loss.

Implementation using the MOESI protocol

The implementation of this adaptive technique has been done extending the MOESI protocol. Adding extra states for the L2 cache blocks and the directory entries, we can deallocate a cache block from the private L2 cache, blocking it in the directory, and send it to the sharing engine. The sharing engine will send the block to a new placement, where the block will recover its previous state retained during the deallocation. The next section will show this with illustrative examples.

5.3. Working examples

In this section we describe the behavior of the adaptive scheme implemented with two illustrative examples. The examples show the requests and responses sent between architectural elements and the state of cache blocks and the directory entries. The states appear between keys and a state transition is represented with the initial state and the final state separated with a comma. The state NP indicates that the block is not present in the cache.

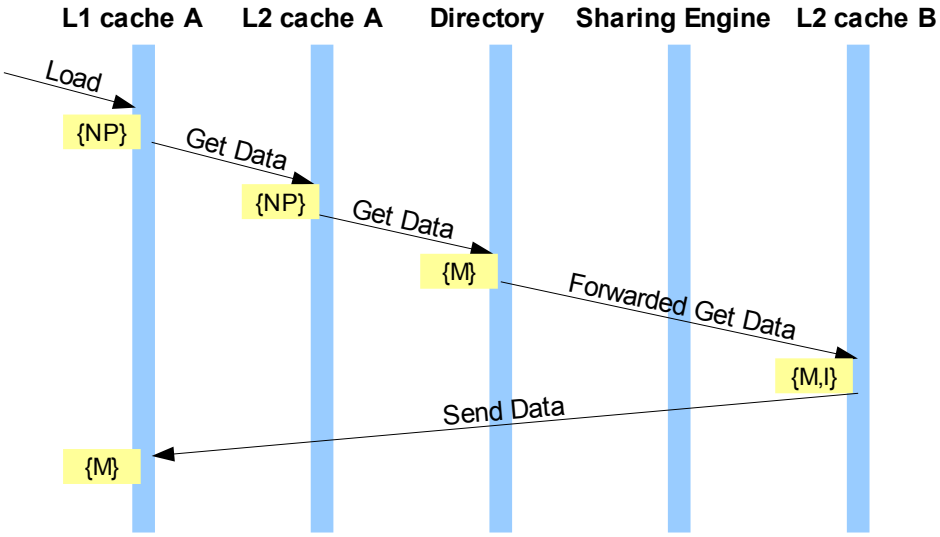


Figure 5.4: Load operation example.

Figure 5.4 presents a simple example of a Load operation that the L1 and the local L2 can not satisfy. The last-level cache send the request to the directory that looks its entries up. As the block is in the shared partition of the cache B the request is forwarded to the neighboring L2 cache B. The L2 cache responds directly to the requestor with the data and invalidates the copy that holds. The invalidation is done because we do not contemplate the possibility of share cache blocks.

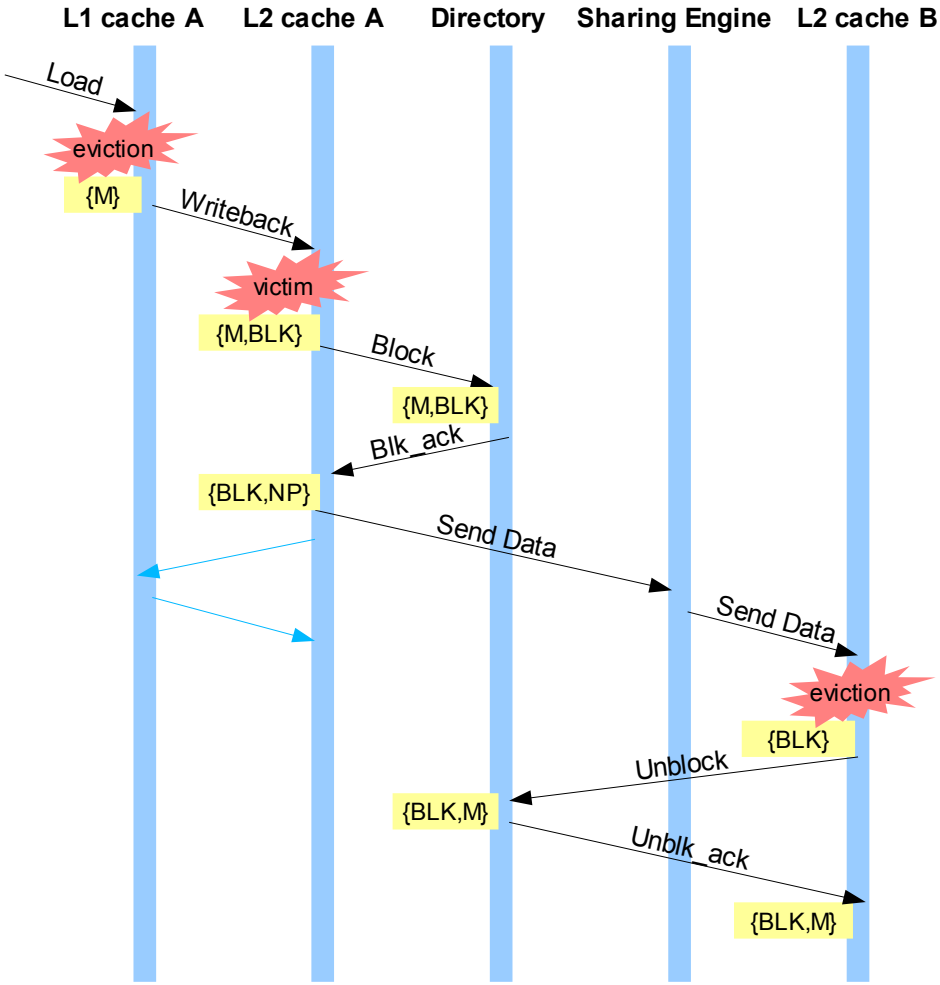


Figure 5.5: L1 writeback request example.

In figure 5.5 we can see another example. When a block in the L1 cache A is evicted, it is placed in the private partition of the L2 cache A. If the private partition of the local last-level cache has not free entries a cache block must be evicted. The victim to be evicted is selected using the LRU policy. The victim block will be sent to the shared partition. For this, the L2 cache A sends a petition to the directory that blocks that block. When the L2 cache A receives the acknowledgment form the directory, it can deallocate the block and send it to the sharing engine. The sharing engine decides where to allocate the block and sends the cache block there. If the destination cache block is in use, the present block must be evicted. When the block is available the data is allocated and we send a petition to the directory to unblock the cache entry. The directory unblocks its entry and changes the owner of the cache block. When the destination cache receives the acknowledgment from the directory, the destination L2 cache B recovers the previous cache state.

6. Scalable adaptive shared/private NUCA cache partitioning scheme

6.1. Scalability and closeness of Dybdahl's scheme

The centralized solution proposed by Dybdahl uses two main data sources to perform their replacement policy: a global LRU stack per set and the number of block in set that each core uses.

Maintaining the LRU stack per set of 16-, 32- or 64-ways associative caches is possible if we have a monolithic cache. Our baseline framework, however, has the last-level cache distributed in banks forming a NUCA cache. Implement a real LRU stack, or a pseudo-LRU approximation, is mainly not possible due to the interconnection network latency and the hardware overhead.

The LRU and pseudo-LRU policies are implemented using stacks and trees structures, see table 2.1 for its implementation costs, and have no real time-stamp that can be compared across the cores. Due to this, we can not have a distributed LRU stack in our non-uniform-latency network.

On the other hand, keeping (like our implementation of the Dybdahl's scheme) a centralized LRU stack using a copy of the cache tags will have a very high cost in area occupancy and power consumption. In addition, this block will be susceptible to suffer collision due to our limited bandwidth.

Monitoring the occupancy in blocks per set of each core is also expensive. Due to the interconnection latencies it is not possible calculate the number of blocks that each core uses at each block replacement petition. Having a centralized structure that maintains these counters will have huge area and power impacts, and may consume an important bandwidth.

Another problem that we can see in Dybdahl's implementation is the no-consideration of the network latency. One of the main problems of 2-D mesh networks is the high latencies that might return. Dybdahl's solutions looks into the LRU stack per set and selects a victim without taking into account the distance to it. This can cause that when we have bigger networks we have huge average access times.

With our novel proposal we want to address these presented problemes having in mind also the terascale requirements.

6.2. Distributed adaptive partitioning scheme

Distributed scheme

As we have seen, the model presented by Dybdahl has many difficulties to be implemented. Furthermore, by its behavior, it seems that it may have scalability problems. In this section we present our proposals to solve these problems.

To deal with the presented problems we decided to use a distributed scheme. Splitting the sharing engine into smaller parts that can run simultaneously would avoid the potential contention of the proposed centralized scheme.

Having decided how to avoid maintain a centralized management structure, we must now see how to adapt the scheme. Without a central element we can not be aware about all other caches, so our idea is to use a greedy approximation to the algorithm. The greedy approach would be applied in the

replacement policy.

In Dybdahl's scheme, the replacement policy (see algorithm 5.2) runs a search across a global LRU stack. In these searches, the premise that determines the block to be evicted is

if *max_blocks_per_set[procID] < blocks_in_set(procID)* ***then.***

Knowing that the maximum blocks per set registers keep the maximum cache blocks that a processor can use per set in both private and shared partitions, we can consider eliminating the contage of blocks in the private partition because they have no significance. Thus,

if *(max_blocks_set[procID]-cache_associativity) < blocks_shared_part(procID)* ***then.***

This simplification brings us where we wanted to arrive. We only need to be worried about the number of blocks used in the shared partition, and this counting can be done on a gradual way. While our greedy algorithm runs across the nodes searching a victim, we were counting the number of blocks occupied by each core. When we find a block that belongs to a core that uses more than its maximum number of blocks in set, we have found a victim. This greedy search can be seen in algorithm 6.1. We will be into details later.

```
function Find_Local_Block_to_Evict (in previous_blocks_in_set)
        blks_in_set := previous_blocks_in_set
        for prID := 0, number_of_processors do
                blks_in_set(prID) := blks_in_set(prID) + get_local_blks_in_set(prID)
                if (max_blks_set[prID]-cache_assoc) < blks_in_set(prID) then
                        return local_LRU(prID)
                end if
        end for
        send_to_neighbor  blks_in_set
end function
```

Algorithm 6.1: Algorithm to find the block to evict in our greedy approximation.

To implement this solution we have designed the architectures that we can see in figures 6.1 and 6.2. As noted, each node contains a small sharing engine that has access to the interconnection network and is placed very close to the L2 cache.

From this point on, we will describe the necessary hardware resources to implement our novel scheme. Subsequently, we will characterize the behavior of the proposed NUCA-based implementation.

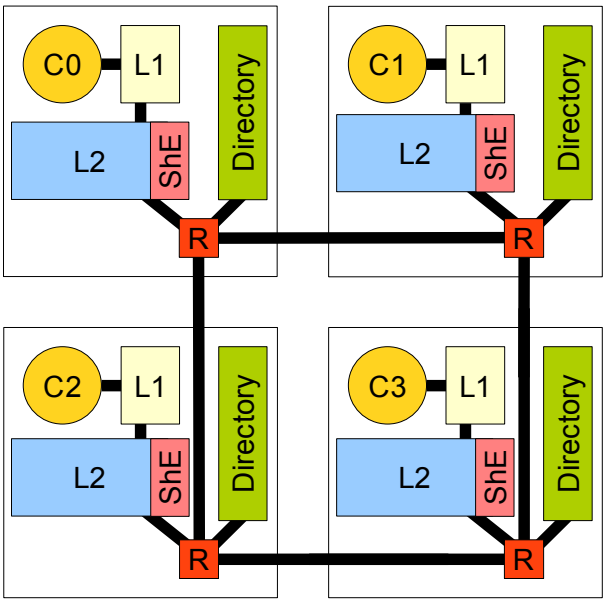


Figure 6.1: Proposed distributed scheme. All the processors have a small sharing engine that implements the adaptive partitioning policy.

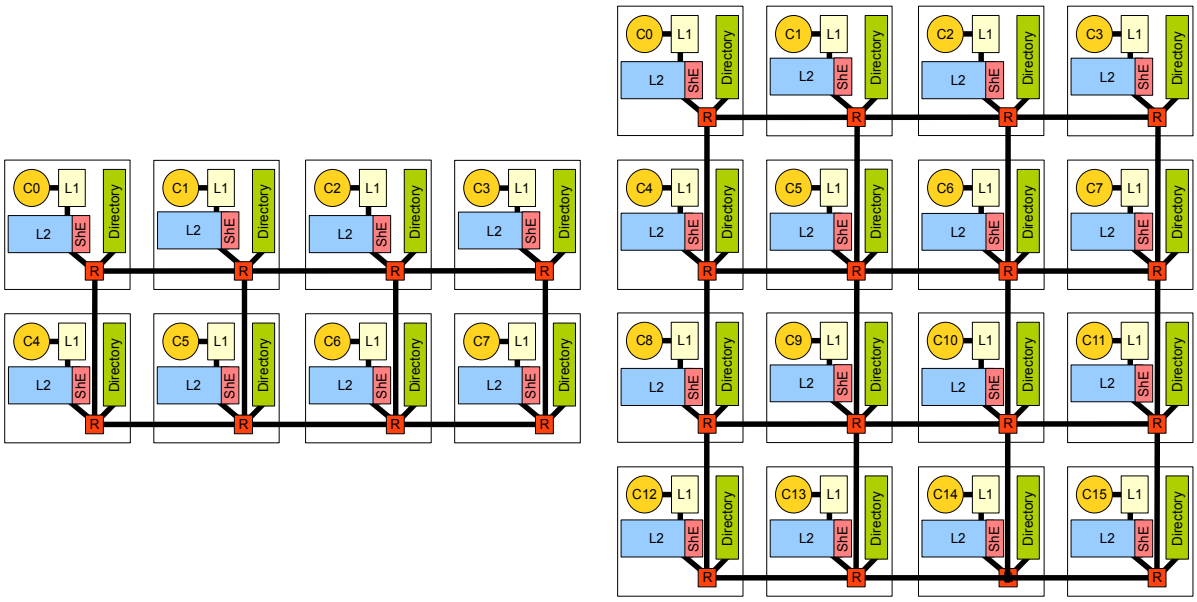


Figure 6.2: Distributed model for the 8- and 16-cores tiled architectures.

Cache quotas

Each core can hold a limited number of ways that is stored in a register. Each register indicates the maximum cache blocks per set that a processor can use. In the centralized approach, these registers are part of the central sharing engine. However, in our distributed scheme they have to be replicated to be consulted by each of the small sharing engines. This replication will not have problems because of the lazy nature of the scheme, allowing us to perform the update of these records asynchronously. Table 6.1 represents the set of register that must be replicated in each node.

Description	Core 0	Core 1	Core 2	Core 3
Max. # of blocks in set	3	4	2	3

Table 6.1: Registers with the maximum number of blocks in set replicated in each node.

As we use quotes, the update of the maximum number of ways in set per core will be carried out in a lazy form. When the repartition policy decides to modify the assignment of the ways, it only sends the maximum values per core to each sharing engine. Who truly cares about the use of the ways is the replacement algorithm.

Like the centralized scheme, at the start up, the maximum number of blocks is 3/4 of the local cache associativity.

Estimating the effects of increase/decrease one block

The adaptive schema estimates the benefits of increasing and decreasing one block per set to minimize the total number of cache misses.

To estimate the cache misses that would have been avoided if the number of ways assigned was increased by one we count the hits in the shadow tags table. In order to have a fully distributed model we want that the table with the shadow tags was also fully decentralized. To this, we assign to each processor the responsibility to maintain its own shadow tags table (see figure 6.2).

The shadow tags hold the last evicted block in set per processor. As one of our blocks in the shared partition can be evicted out of our scope, in a neighboring L2 cache, we must know when a replacement occurs. To solve this problem, each time a core replaces a block that does not belong to it, it will send a message to the owner of the cache block with the evicted address.

Set number	Core *i*
0	7ED1CE64
1	7DC95D88
...	...

Table 6.2: The shadow tags maintains the last evicted blocks in set to estimate the effect of to have one more cache block.

The effect of decreasing one cache block per set is calculated counting the hits in the LRU stack head at any request. As our distributed approach does not maintain the global LRU stack that is the base of the Dybdahl's scheme, we must find an alternative. We propose the use of the local LRU stack. Each time a cache block is displaced to the shared partition we can allocate them as the MRU block of the new core. We can consider that when this block becomes the LRU block on the neighboring cache many accesses to other blocks have already happend. Due to this, it is also possible that the cache block is the LRU block of its owner.

Comparing the hits in the LRU blocks and the hits in the shadow tags we can estimate the effects of removing a cache block of a processor and assigning it to another processor.

To keep the hits in the LRU and the hits in the shadow tags we need a set of counter. These counter must be centralized because the repartitioning algorithm must have access to them. Anyway, the repartitioning algorithm is only executed every *N* last-level cache misses, and these counters can be progressively upgraded. Table 6.3 shows this counters.

Counter	Core 0	Core 1	Core 2	Core 3
Hits in the LRU blocks	32	11	0	12
Hits in the shadow tags	24	0	9	54

Table 6.3: Counters maintained by the distributed adaptive technique to estimate the effect of decreasing and increasing one cache block.

Repartitioning policy

The adaptive technique needs to reevaluate the partition sizes per core on a regular basis. The repartition algorithm looks for the processor with a high reduction of cache misses with one more cache block, the one with more hits in the shadow tags, and compares it with the processor with less penalties with one cache block less, the one with less hits in the LRU counters. In our distributed implementation these counters must be centralized in some place but as mentioned before this is not a problem.

If the gain is higher than the loss, one cache block of the processor with the lower loss is provided to the core with the highest gain (see algorithm 6.2). As the algorithm could leave without private cache blocks a processor, we establish that the processor will always have at least one private cache block and never more shared blocks that the private partition size.

```
high_gain := MAX_incrementable(hits_ShadowTags)
low_loss := MIN_decrementable(hits_LRU)
if  high_gain > low_loss then
        maximum_blocks_in_set [high_gain] ++
        maximum_blocks_in_set [low_loss] --
end if
```

Algorithm 6.2: Every 2000 cache misses the partition sizes are reevaluated.

Characterization of the NUCA model

Like in Dybdahl's model, the *access policy* used in this technique involves a two phase process. First we look for a tag in the local cache and if there is no match, we check the directory for that bank. The directory will forward our request to the neighboring cache or to the main memory.

When a request produce a miss on the private partition of the local last-level cache and a hit on the shared partition, the *migration policy* is allowed to change the data placement in the NUCA cache. As we are working with an exclusive cache the data in the shared partition goes through the local private partition and is allocated directly in the L1 cache.

The *placement policy* has the same behavior described in the centralized technique, the new incoming data goes directly to our L1 cache. If the L1 cache hasn't free entries, a present block must be replaced and written back to the L2 cache. If the private last-level cache does not have invalid blocks we will replace the least recently used block. The replaced block will be sent to the local sharing engine.

The sharing engine is the mechanism in charge to determine if we allocate the *displaced* block in the local shared partition or we must sent it to a neighbor. If the local shared partition does not have any possible victim, the *replacement policy* will send the block to a neighboring sharing engine. To decide the neighbor to which we send the block we use a round robin policy forbidding that the block goes back on the same route.

The possible victims are determined using the algorithm 6.1. When we evict a block from the private partition we send it to the sharing engine. If there are not free blocks in the shared partition, we count the number of blocks that each cache uses in the local shared partition. If any of them uses more blocks than its maximum we have a victim. If not, we will send the block to a neighboring cache together with the number of blocks that each core uses. In this way, each message will go closer to the occupancy of each core. The process is repeated in the present sharing engine. The next section shows this with an example.

6.3. Working example

In this section we describe the behavior of the distributed adaptive scheme implemented. The examples show the requests and responses sent between architectural elements and the state of cache blocks and the directory entries. The states appear between keys and a state transition is represented with the initial state and the final state separated with a comma. The accumulative counters used in the messages between the sharing engines are represented in a box over the sharing engine line time.

Lets suppose that we are using our 4-core implementation. Table 6.4 shows us the maximum blocks in set assigned to each core, and the table 6.5 shows who is the owner of each cache block in a concrete set.

Description	Core A	Core B	Core C	Core D
Max. # of blocks in set	5	3	1	3

Table 6.4: Maximum number of blocks in set per core.

Last-level cache block	Way 0	Way 1	Way 2	Way 3
L2 cache A	A	A	A	A
L2 cache B	B	B	B	A
L2 cache C	C	C	D	B
L2 cache D	D	D	D	B

Table 6.5: Block owners of a set. Darker blocks indicate the private partition.

Observe Figure 6.3. When a block in the L1 cache A is evicted it wants to be placed in the private partition of the L2 cache A. This private partition has not free entries and a cache block must be replaced. The victim to be replaced is selected using the LRU policy.

The victim block will be sent to the shared partition. For this, the L2 cache A sends a petition to the directory that blocks the victim block. When the L2 cache receives the acknowledgment from the directory it can deallocate the block and send it to their sharing engine A.

The sharing engine counts the number of block that uses each core. In this case, as all the last-level cache is private we return 0 for all cores. We do not have a possible victim and we must send the displacement petition to a neighboring sharing engine. The round robin algorithm returns the sharing engine B as our destination.

Sharing engine B (the L2 cache B) shares one block that belongs to core A. Core A can use 1 block in the shared partition, so this is not a victim. We update the counters to [1,0,0,0] and send the petition to the next sharing engine.

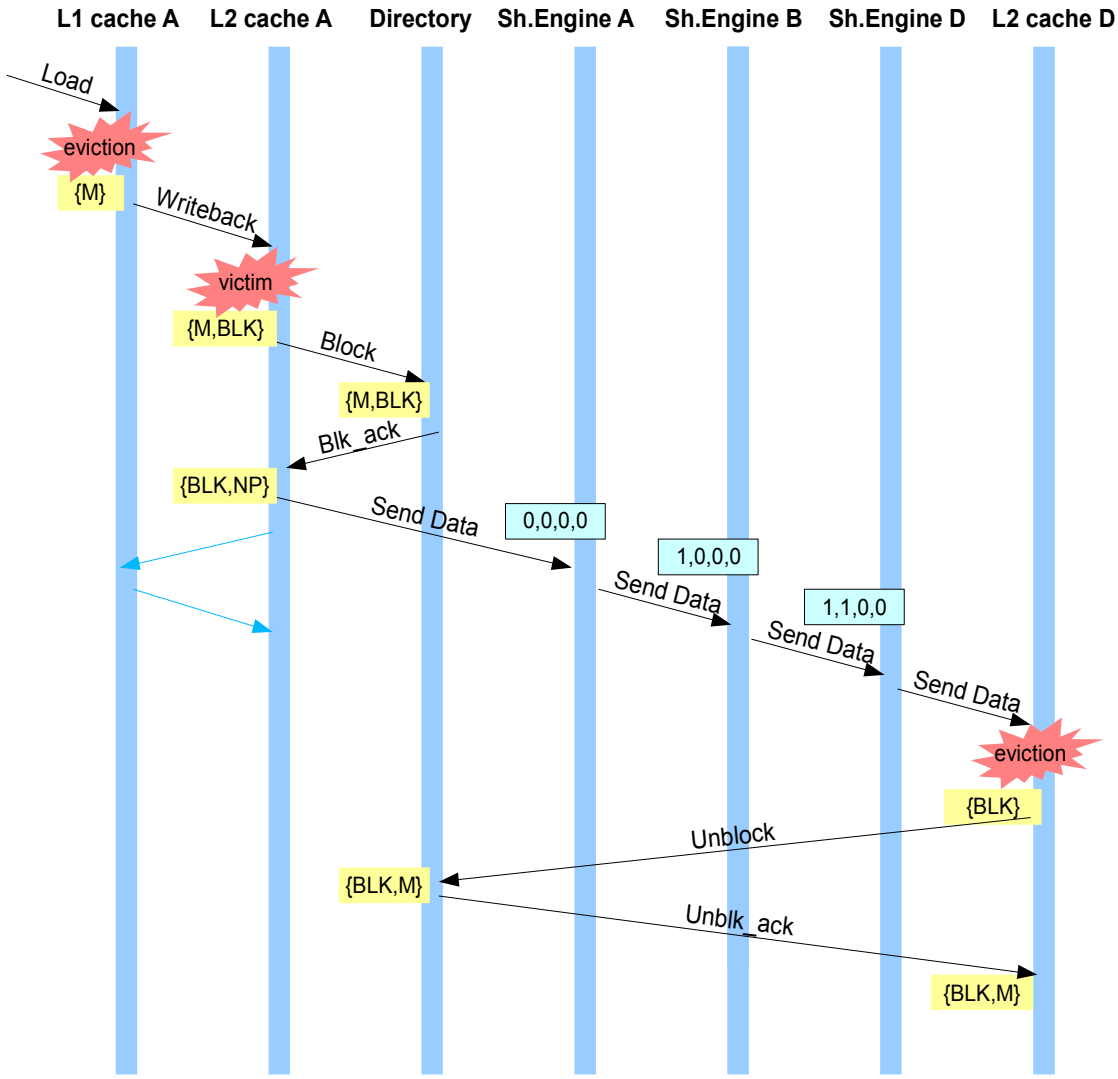


Figure 6.3: L1 writeback request example.

The sharing engine D (the L2 cache D) has one block shared that belongs to core B. The core B is not allowed to allocate blocks in the shared partition, so we have a victim. The sharing engine sends the block to the L2 cache D.

In the L2 cache D, if the destination cache block is in use the present block must be evicted. When the position is available the data is allocated and we send a petition to the directory to unblock its cache entry. The directory unblocks its entry and changes the owner of the cache block. When the L2 cache D receives the acknowledgment from the directory the block can recover its previous cache state. The state NP indicates that the block is not present in the cache.

6.4. Performance variability due to the applications placement

Our proposal uses a greedy approximation of the Dybdahl's scheme to find a new placement for the evicted blocks. The greedy algorithm performs the search of a victim block across a route determined by the round robin counter of each sharing engine. Since this route is not aware of the closer cache block where we can place our displaced data, this can not be optimal.

Moreover, the applications that run in neighboring nodes will affect our performance. Due to our

greedy approximation, having neighbors with high maximum blocks in set values may produce that the first possible victim appears at least at two network hops. On the other hand, if we have few blocks per set assigned, it is possible that our shared partition suffers a high number of collision misses.

These two problems motivate the study of heuristic routings, to find routings closer to the optimal, and the application placement effects.

In the chapter 8 we present a brief preliminary study of the effects that the applications' placement can produce. The exploration of heuristic routings performance will be part of the future work.

7. Methodology

7.1. Simulation environment

The proposed techniques have been evaluated with Simics [12], a full system execution-driven simulation platform capable of simulating high-end target systems with sufficient fidelity and speed to boot and run operating systems and commercial workloads. On Simics we run Solaris 9 operating system with Sun Studio 11 compilers.

Over Simics we run Ruby, a module of GEMS [26], that provides a detailed memory hierarchy model that allows us to determine the timing of the memory system. Thus, the cache latencies, the interconnection network and the coherence protocol. The interconnection links have a latency of 1 cycle and the routing elements have a latency of 3 cycles. The rest of the environment configuration can be seen in table 7.1.

Private		**Shared**	
# of Processors	4 or 8 or 16	# of Processors	4 or 8 or 16
Processor cores	inorder, 1-issue	Processor cores	inorder, 1-issue
Cache block size	64 bytes	Cache block size	64 bytes
L1 Data	32KB, 4-way	L1 Data	32KB, 4-way
L1 Instruction	32KB, 4-way	L1 Instruction	32KB, 4-way
L1 Latency	4 cycles	L1 Latency	4 cycles
# of L2 Banks	4 or 8 or 16	# of L2 Banks	4 or 8 or 16
L2 Cache Bank	512 KB, 4-way	L2 Cache Bank	512 KB, 16- or 32- or 64-way
L2 Latency	12 cycles	L2 Latency	12 cycles
L2 Tag Latency	4 cycles	L2 Tag Latency	4 cycles
# of Directory banks	4 or 8 or 16	# of Directory banks	4 or 8 or 16
Directory Latency	4 cycles	Directory Latency	4 cycles
Memory Latency	280 cycles	Memory Latency	280 cycles

Table 7.1: Memory system parameters for private and shared configurations.

The GEMS simulator uses a specific language named SLICC, Specification Language for Implementing Cache, to implement the memory components of Ruby. The language models the memory components behavior as a finite state machine. We have modified the *slicc* files to add the centralized and our distributed schemes, and we have adapted the last-level caches and the directory behaviors.

We compare our schema with a private cache with the same configuration, a two times bigger private cache (in number of sets), and a shared cache based on a S-NUCA model. As we commented previously, Dybdahl defined a maximum use of the shared partition limited to the size of our private

partition. Therefore, the *big* private configuration will be our upperbound.

7.2. Workloads

To create different workloads we have used the SPEC2000 benchmarks. We use SPEC2000 instead SPEC2006 because SPEC2006 have bigger footprints that slowdown very much our simulations.

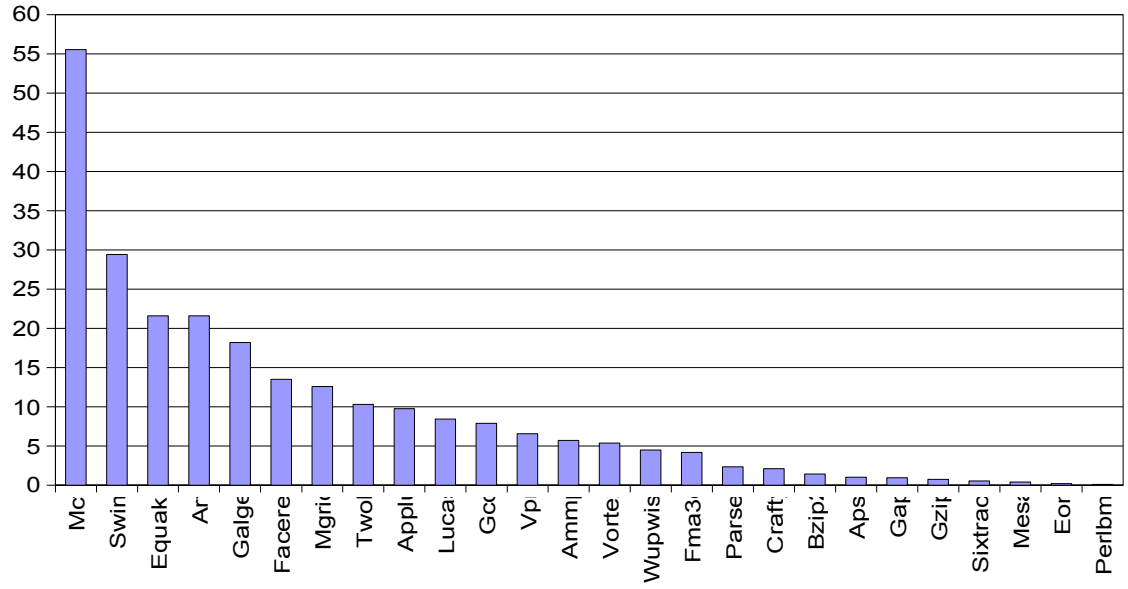


Figure 7.1: SPEC2000 applications ordered by its misses per thousand instructions.

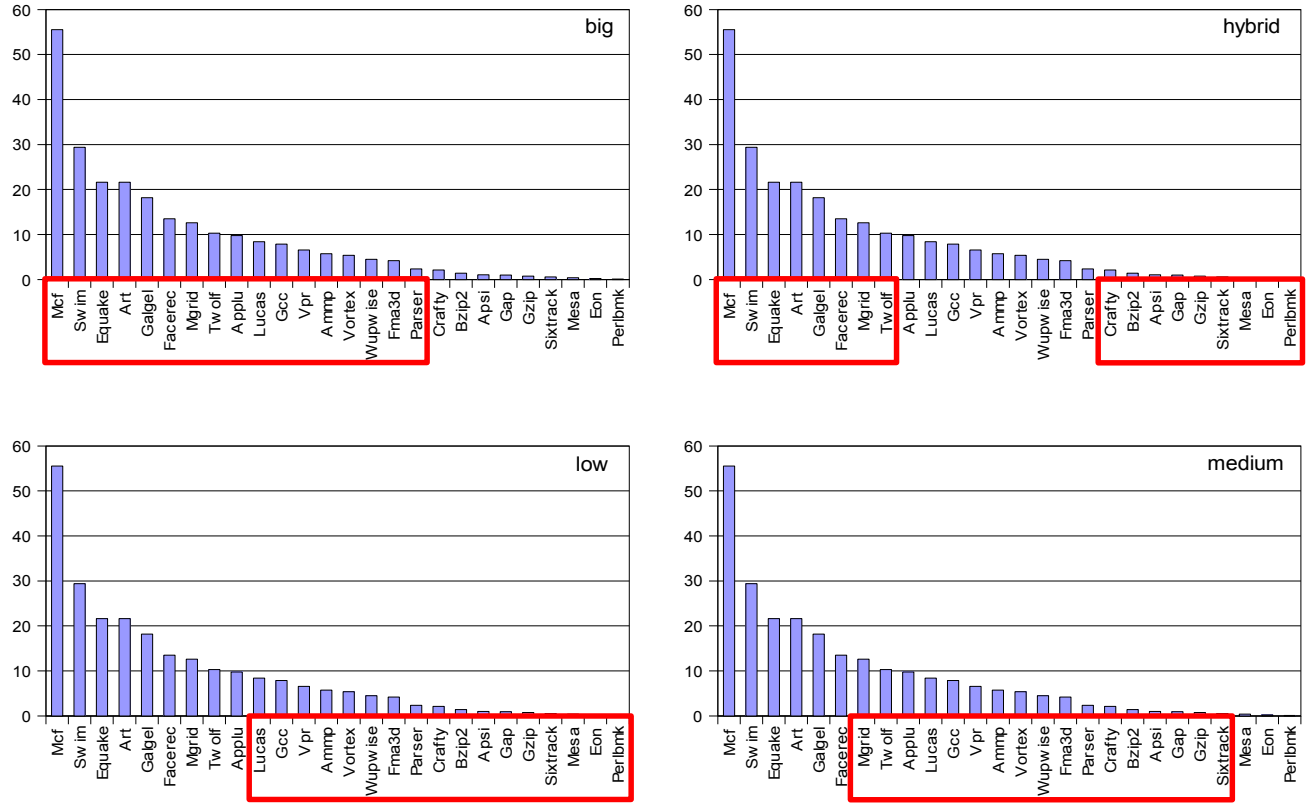


Figure 7.2: Different workloads for the 16-cores architecture.

We profiled the SPEC2000 benchmarks classifying the applications depending on its cache miss rate, see figure 7.1. For our 4-, 8- and 16-cores architectures, we have created four different workloads composed by cache-intensive applications, non-depending on cache applications, a workload with fifty-fifty, and a workload with applications with an intermediate miss rate. We named this configurations big, low, hybrid and medium respectively. Figure 7.2 shows an example of how we select the benchmarks in each workload.

The misses per thousand instructions showed had been measured in the same simulation point were we run the simulations. To this, we firstly profiled the full benchmarks and we determine the simulation points. These start points were selected after the initialization interval and within the function with a higher weight in the execution time of each application. Thus, we have tried to select a representative point of each benchmark. Table 7.2 presents the skipped instructions in each benchmark and its command line.

We can see in Figure 7.2 how the big workload for the 16-cores architecture it is not fully composed by memory intensive applications. Due to this, we create a special workload for this configuration called huge with the eight first benchmarks, the ones with higher miss rates, running twice each. For this architecture we do not present the medium workload because it does not finish the entire simulation due to problems in GEMS simulator.

Each workload have been executed for a warm-up of 500 million cycles and have been simulated during 500 million cycles.

Benchmark	Instructions skipped	Command line
Ammp	6510 millions	ammp.rr < ammp.in
Applu	10400 millions	applu.rr < applu.in
Apsi	3690 millions	apsi.rr
Art	2383 millions	art.rr -scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10
Bzip2	1442 millions	bzip2.rr input.source
Crafty	19230 millions	crafty.rr < crafty.in
Eon	1372 millions	eon.rr chair.control.cook chair.camera chair.surfaces chair.cook.ppm ppm pixels_out.cook
Equake	2650 millions	equake.rr < inp.in
Facerec	23230 millions	facerec.rr < ref.in
Fma3d	16400 millions	fma3d.rr
Galgel	5260 millions	galgel.rr < galgel.in
Gap	320 millions	gap.rr -l ./ -q -m 192M < ref.in
Gcc	4880 millions	cc1.rr 166.i -o 166.s
Gzip	7160 millions	gzip.rr input.source 60
Lucas	2800 millions	lucas.rr < lucas2.in
Mcf	1170 millions	mcf.rr inp.in
Mesa	8935 millions	mesa.rr -frames 1000 -meshfile mesa.in -ppmfile mesa.ppm
Mgrid	2371 millions	mgrid.rr < mgrid.in
Parser	4903 millions	parser.rr 2.1.dict -batch < ref.in
Perlbmk	1605 millions	perlbmk.rr -I./lib diffmail.pl 2 550 15 24 23 100
Sixtrack	11970 millions	sixtrack.rr < inp.in
Swim	16240 millions	swim.rr < swim.in
Twolf	3590 millions	twolf.rr ref
Vortex	12200 millions	vortex.rr bendian1.raw
Vpr	8990 millions	vpr.rr net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2
Wupwise	7510 millions	wupwise.rr

Table 7.2: Skipped instructions in each benchmark and their command lines.

7.3. About metrics

In CMPs, multiple processor cores share several hardware resources such as the cache memory, increasing the contention and degrading the performance of each core and losing of fairness between applications.

Measuring the performance of a system is highly discussed and several solutions have been proposed [43] [25]. If we use the throughput as a performance metric we will not care about the fairness between applications and we do not want that to increase the performance of a few cores we have to degrade the performance of others.

Unlike the approach of Dybdahl, we not only want to reduce the total number of off-chip misses, but we want to reduce the misses the cores with higher miss-rates without harming the rest. Due to this, we decided to present our performance results using the harmonic mean of the speedups of each core, *i.e.*

$$Overall\ speedup = \frac{n}{\sum_{i=0}^{n} \frac{1}{x_i}}$$

where x_i is the relative speedup respect the private model of each core, this is $\frac{IPC_i}{IPCprivate_i}$, and n the number of cores.

The harmonic mean is appropriate for situations when the average of ratios is desired. It tends to mitigate the impact of large outliers, compared to the arithmetic mean, aggravating the impact of small ones. In our case, the harmonic mean will reduce the weight of the individual speedup giving more importance to non-interference the performance of cores with lower miss-rates.

When we show the number of misses per thousand instructions of each configuration, however, we use the sum of misses per thousand instructions of each core. In this case, we are only interested in view the total off-chip misses that the adaptive models can avoid.

Before showing the results, it is important to remember Amdahl's law. In an architecture with 16 cores, if we achieve a performance improvement of 40% in two of them, the Amdahl's law shows that the speedup that we will get is

$$Speedup = \frac{1}{(1-\frac{2}{16})+\frac{\frac{1}{16}}{2.5}} < 1.09$$

This result advances that the performance gain values that we should expect for the adaptive schemes will be small.

8. Simulation results

8.1. Exploring the behavior of the adaptive scheme

Before we present the simulation results, its interesting to show the behavior of Dybdahl's adaptive model in action. Two are the interesting points: what decisions take the adaptive model and how the system reacts to these decisions. Figure 8.1 shows the maximum number of blocks in set assigned to each processor during a simulation interval of 10 million cycles. We appreciate the temporal evolution of the blocks assigned to each core and how the implemented technique adapts dynamically to the space requirements of each application.

Figure 8.2 represents the evolution in time of the cache occupancy. This chart shows us the percentage of the whole cache that each application uses during the simulation interval. We can appreciate the correlation between the maximum number of block in set that we can use and the cache occupancy of each core.

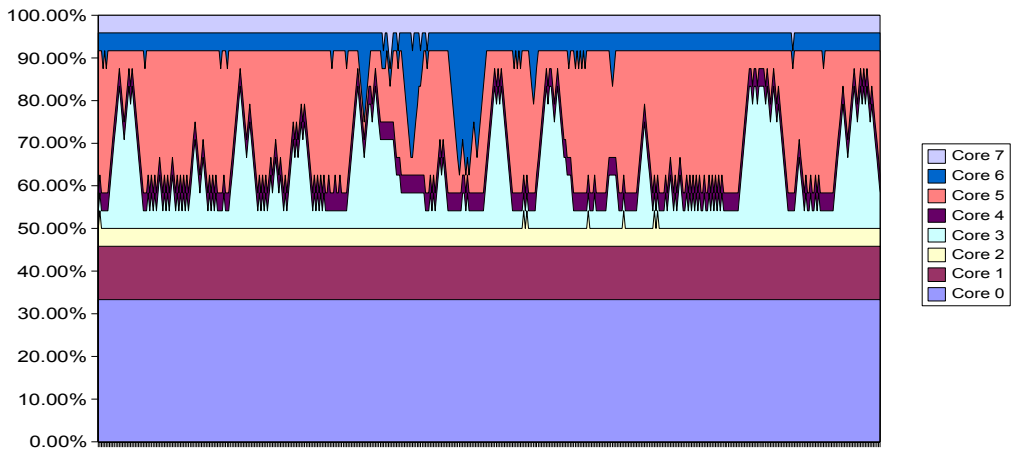


Figure 8.1: Temporal evolution of the maximum number of cache block in set per core. Figure corresponding to a 10M cycles interval of the big workload for the 8-cores architecture.

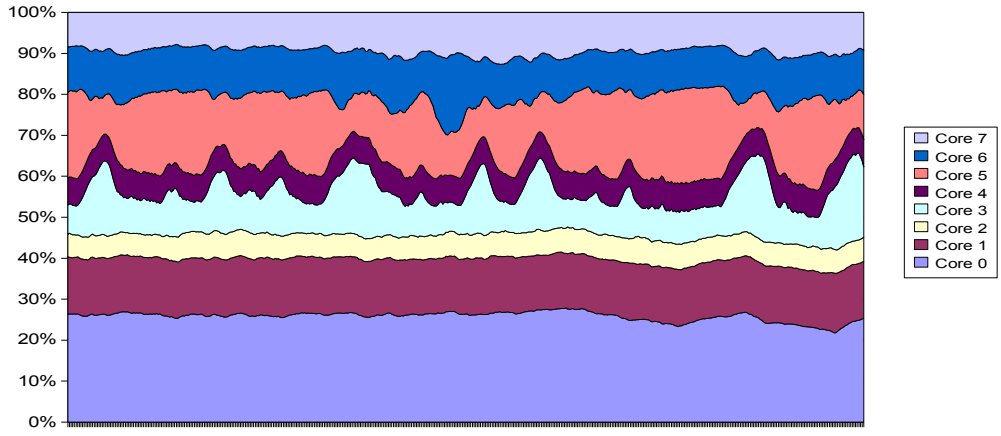


Figure 8.2: Cache occupancy per core during a simulation interval of 10 million cycles. Figure corresponding to a 10M cycles interval of big workload for the 8-core architecture.

The previous figures show us the adaptive behavior of the Dybdahl's technique validating our implementation. With one *quota* register per core the processors share their cache space adaptively. Some of the proposed techniques cited in the related work, however, use a fine grain quota mechanism per block [28] or share the sets without a maximum number of ways [8][18]. Due to this different sharing degrees per set, we consider the possibility of using multiple maximum registers for each core, one for every *N*-sets. Figure 8.3 analyzes the cache occupancy of each core per set of *N*-sets. We consider 16 *subsets* of *s/16* sets, where *s* is the number of sets. Figure shows that the variability among subsets occupancy is low, less than 1% in average. This brief study allows us to dismiss the need to implement more than one maximum blocks in set register per processor.

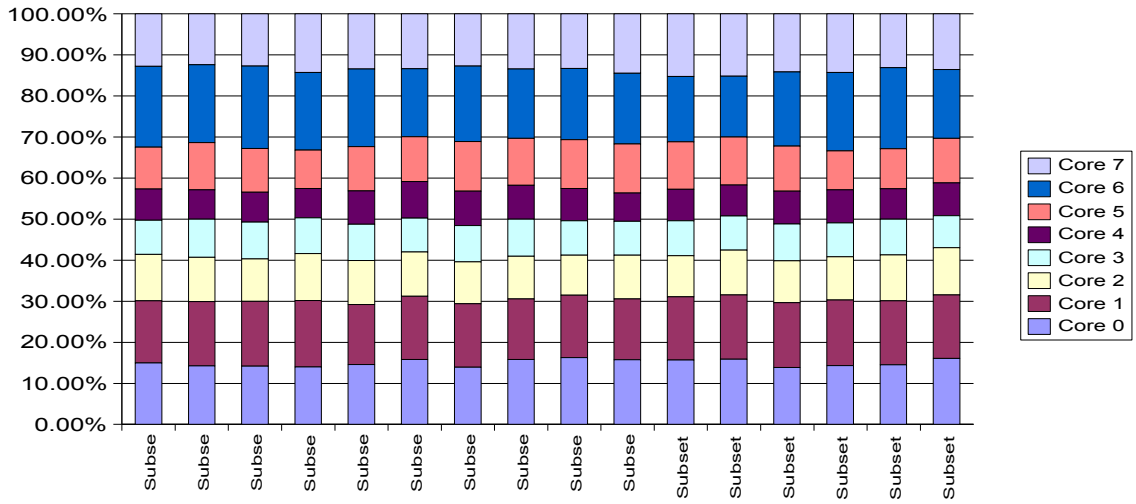


Figure 8.3: Occupancy of the last-level cache in subsets of *s/16* sets at an instant during the execution of the Dybdahl's model.

After these comments, the rest of the section will show the simulation results for the 4-, 8- and 16-cores architectures. We will see also, a detailed analysis of the application's placement influence for the 16-cores configuration and we will discuss the implementation cost of our model.

8.2. The 4-cores configuration

The implementation presented by Dybdahl was based in a 4-cores architecture. He assumed a tightly coupled NUCA architecture that it is slightly different from ours. The access time to a neighboring cache in our implementation is relatively bigger than in Dybdahl's configuration. This reason means that our model will be more sensitive to the use of the shared partition.

The results of our implementation of the Dybdahl's scheme show a 4% improvement of the system's performance in average compared to that of the private model. In figure 8.4 we can see how on big and medium workloads the scheme do a good work reducing the number of off-chip misses as we can appreciate in figure 8.5. Still, the shared model outperforms the adaptive implementation on the medium workload. The medium workload for the 4 cache configuration runs *ammp, vortex, wupwise* and *fma3d*. Three of these benchmarks have small cache requirements and have very low sensitivity to the cache size. The fourth SPEC, can achieve important speedups with high associativity caches. In the shared model this benchmark can exploit the 16-way cache memory with low interferences. The adaptive models can not outperform this model due to the limited number of shared cores (a maximum of 4 in the 4-core architecture, thus, the sum of the 25% of ways of each cache). To clarify the results of this model you can see IPC per core of the medium model in appendix A.

In the hybrid workload we achieve good results too, but lower due to the applications we are executing. Two applications are not sensitive to cache sizes, and the other two are highly memory

intensive and need a high associativity cache. Considering the big model as our upperbound, we must bear in mind that the adaptive scheme is very close to the maximum speedup.

The low model runs only not-memory-bound applications. The performance of these applications can not be improved, but we can use this model to check the robustness of the scheme. As we can appreciate the performance does not degrade.

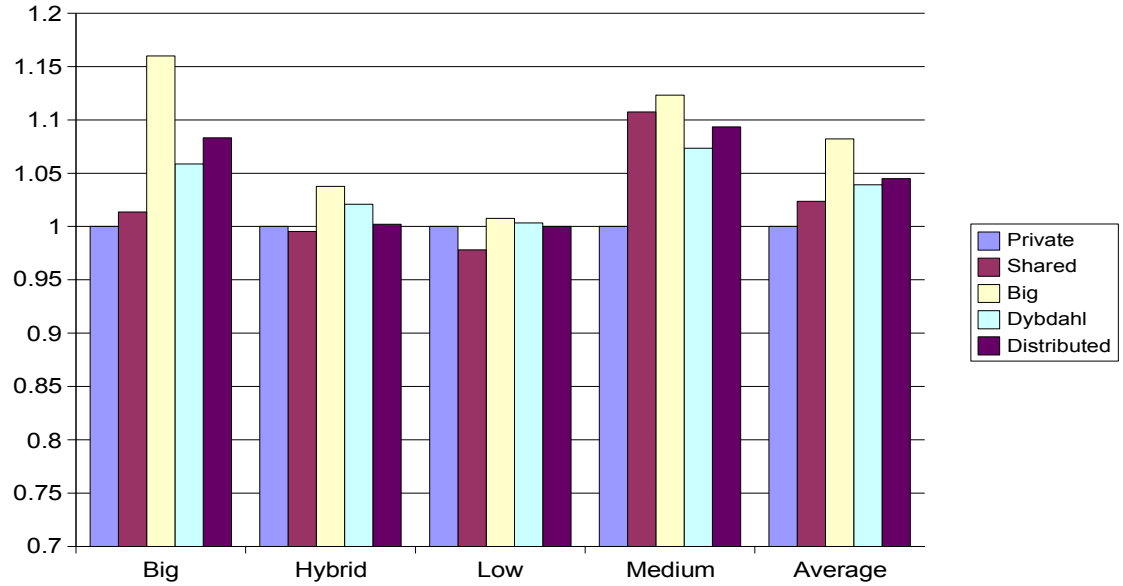


Figure 8.4: Harmonic mean of the speedups relative to the private model for the 4-cores configuration for the different workloads.

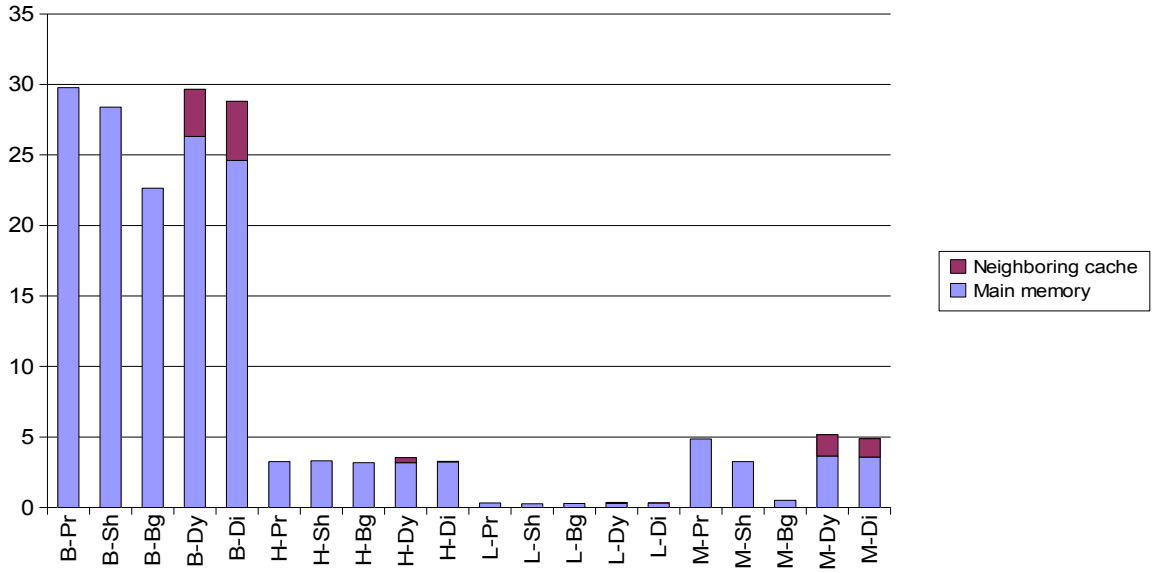


Figure 8.5: Misses per thousand instructions in the last-level cache for each workload and model configuration (expressed W-M). For the adaptive models we separate the off-chip misses and the misses to neighboring caches.

If we observe Figures 8.4 and 8.5 we can see that our proposed model outperforms the Dybdahl's scheme reducing the off-chip misses. This is not true for the hybrid model.

There's an evident reduction on the misses per thousand instructions in our distributed model respect the original adaptive scheme, see figure 8.5. The reason for the reduction is the least interference between the applications with less cache blocks assigned. The distributed model first looks at its shared partition. The applications with few blocks assigned will replace their own shared blocks before to send the evicted block to a neighboring cache, and the applications with more blocks assigned will send the evicted blocks to the closer neighboring caches. As Dybdahl's implementation looks for a victim in the LRU stack does not take into account the distance factor. This can be observed in figures 8.6 and 8.7 where we can see the distance between the evicted block and the new placement. It is clear that our model reduces the access time to the shared partitions.

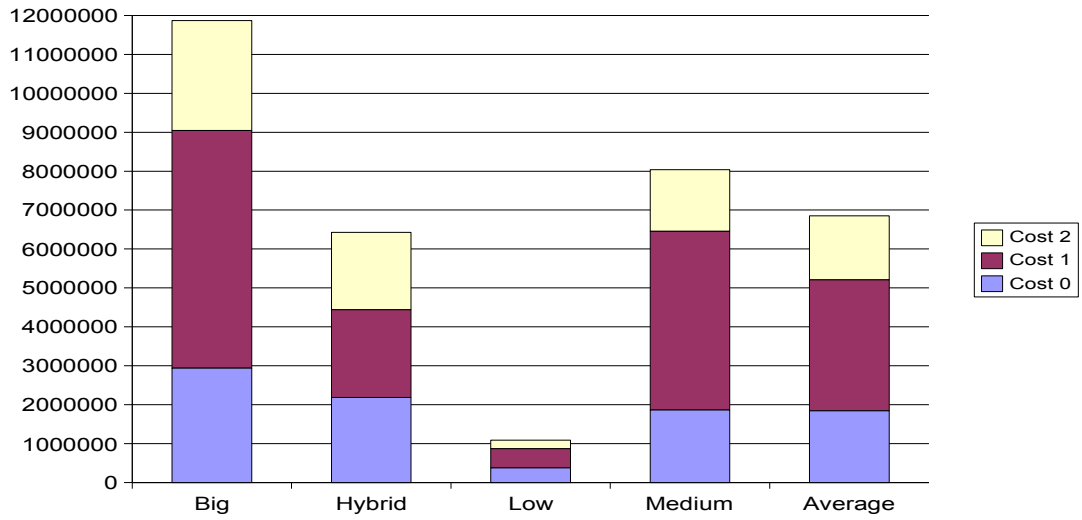


Figure 8.6: Distance of the evictions in the 4-cores Dybdahl's implementation. The cost expresses the number of hops in the internconnection network.

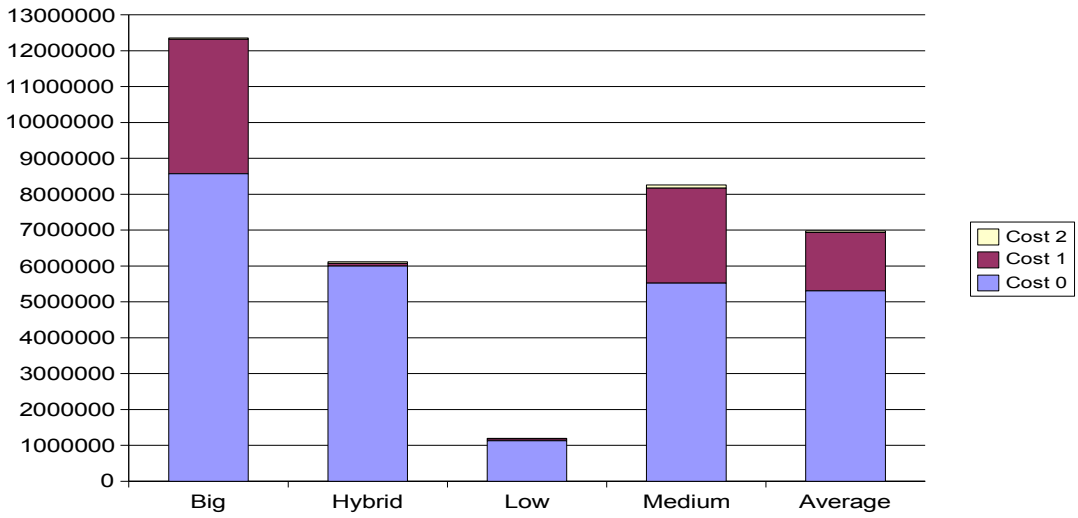


Figure 8.7: Distance of the evictions in out 4-cores model. The cost expresses the number of hops in the internconnection network.

On the hybrid workload our proposal can not outperform the Dybdahl's scheme. In this case, the IPC per core of Dybdahl's scheme shows that it improves the performance of the two memory-intensive applications at expenses of decrease the performance of the rest. Our proposed scheme is more conservative and achieves a high IPC rate for only one of the memory intensive applications. The other

benchmarks are executed with near no-degradation. If we observe Figure 8.7 we can see that the evictions in our scheme are performed mainly locally. This is due to a conservative partitioning, near the initial. Its complex to explain this behavior, the applications with lower miss-rates do not demand more blocks; the memory-intensive applications, due to its space requirements, never would benefit of having one more block, so they neither demand it. With this, the repartitioning algorithm remains stalled.

Private	0,746	0,042	0,090	0,869
Dybdahl	0,744 *(0,997)*	0,043 *(1,014)*	0,098 *(1,081)*	0,865 *(0,994)*
Distributed	0,745 *(0,998)*	0,042 *(1,000)*	0,091 *(1,010)*	0,870 *(0,999)*

Table 8.1: IPC of the private model and the implemented techniques on the hybrid workload. The values in italics are the relative speedup respect the private model.

The performance of the memory hierarchy is the product of the number of memory operations and the average access time. Figure 8.8 shows us the average miss latency of the L1 cache and its standard deviation. Is interesting to see that our distributed adaptive model have lower access time than the centralized model. This is due to it displaces the blocks closer.

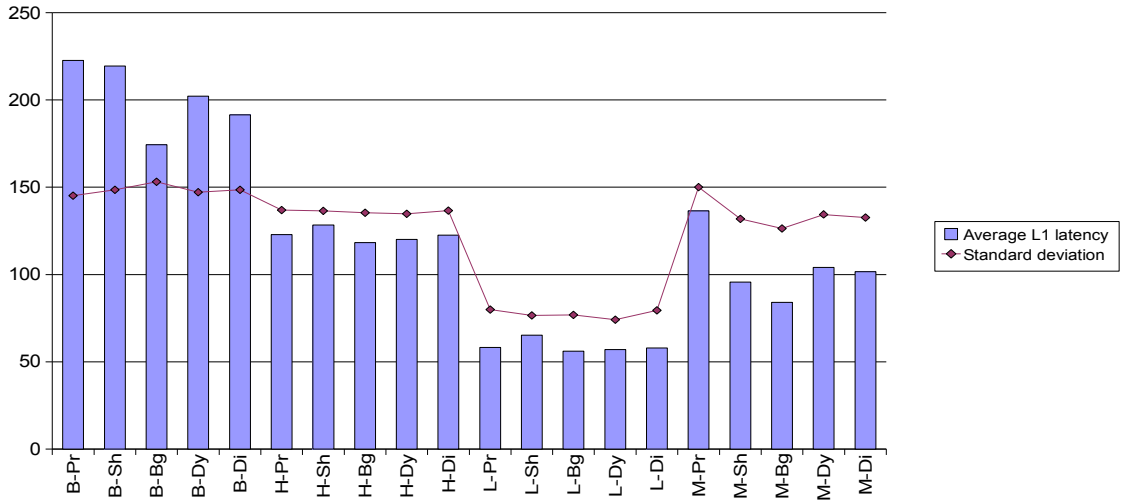


Figure 8.8: L1 cache average miss latency in cycles.

8.3. The 8-cores configuration

In the 8-core architecture we start seeing the scalability problems of the centralized adaptive model. Figure 8.9 shows the harmonic mean of the relative speedups of each cache model. The Dybdahl's scheme do not have, in average, gain in performance. It shows performance loss in the big and the medium workloads, it maintains the performance for the low workload respect the private configuration, and it achieves near a 4% performance improvement for the hybrid workload.

The problems with the big and the medium workloads are explained by a poor isolation. In figure 8.10 we can see the number of misses per thousand instructions. The adaptive model reduces the number of off-chip misses at expenses of extra misses satisfied by the neighboring caches. To reduce the off-chip misses we assign bigger quotas to cores that run highly-memory-intensive applications. Due to this, the applications with less blocks per set assigned have degradations in their performance. As the harmonic

mean tends to mitigate the impact of large outliers, the results for these workloads are not as good as we would expect. In appendix A we can see the IPC of each core for these workloads. In the big workload we can see high speedups but also important degradations. In the medium workload the application have similar miss-rates and the degradation is more extended due to the repartition balancing.

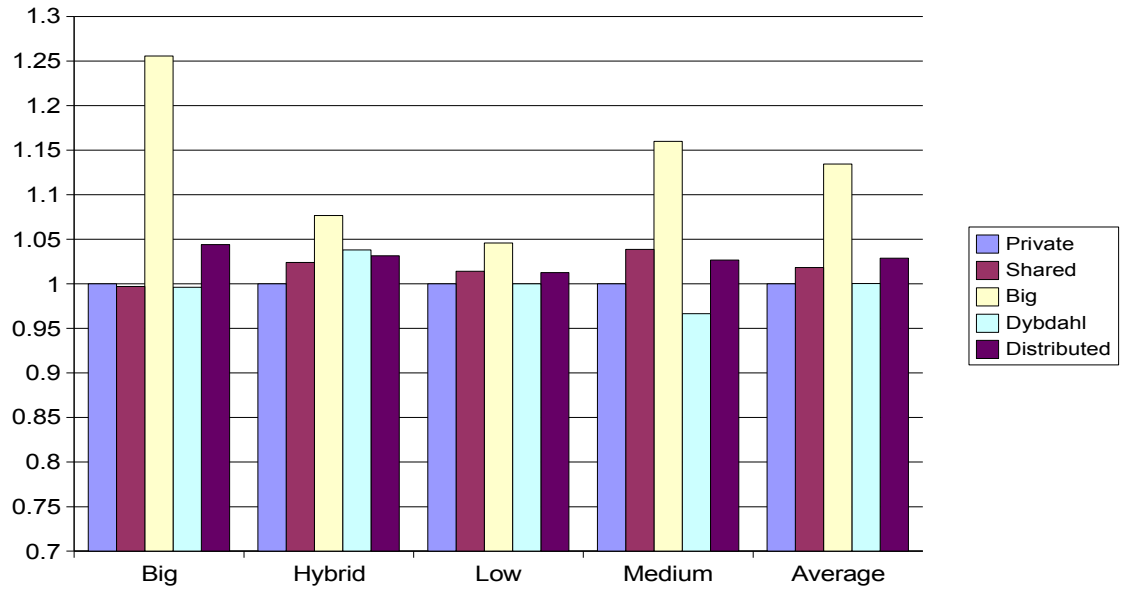


Figure 8.9: Harmonic mean of the speedups relative to the private model for the 8-cores configuration for the different workloads.

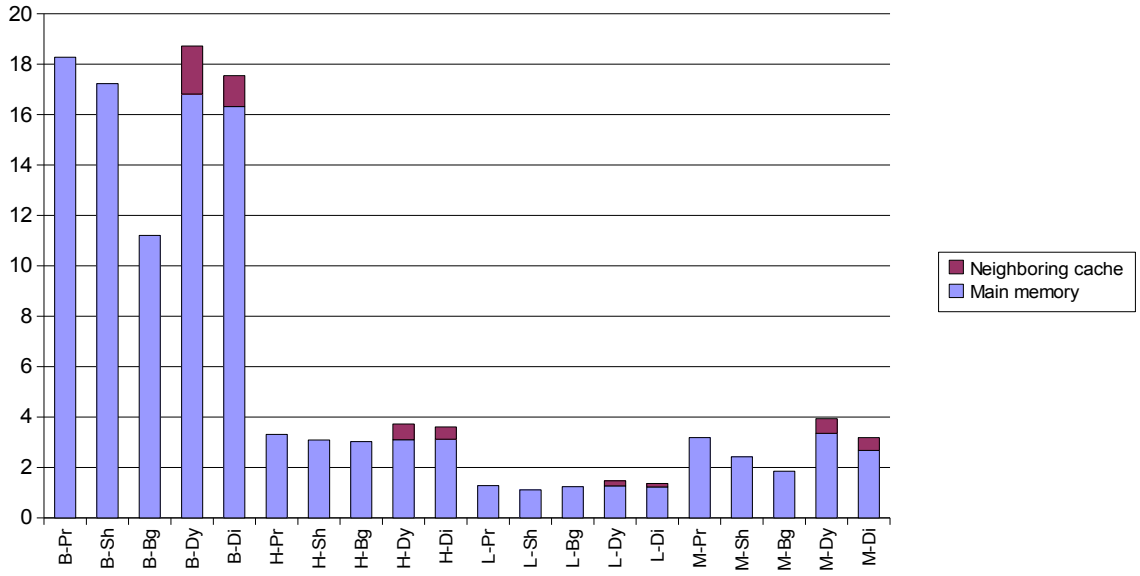


Figure 8.10: Misses per thousand instruction in the last-level cache for each workload and model configuration (expressed W-M). For the adaptive models we separate the off-chip misses and the misses to neighboring caches.

As we commented previously, the Dybdahl's model does not take into account the distance between the consumer, the processor, and the *displaced* block. In this model, the blocks are evicted far and the latency to access them is big taking into account the number extra misses. The distance of evictions can be seen in figure 8.11.

Even without taking into account the distance penalties, the centralized scheme outperforms private

and shared models in the hybrid workload. This is possible because of the nature of the applications, four highly memory-intensive and four with a very low miss rate. In this case is clear which applications must transfer its cache blocks to the shared partition, and the model can reduce the off-chip misses without penalize the applications with few miss-rates.

Our distributed model outperforms the centralized model in all the workloads except the hybrid. The smallest gain on the hybrid workload is because of the same reasons explained in the previous section. The distributed scheme behaves conservatively and few partitions were modified. Due to this, an important amount of displacements are local (see figure 8.12).

Our proposal on the lower workload achieves a similar speedup than the shared model, and on the medium workload the benefits of the shared model are bigger. In both cases the shared model can fit the working sets in the 32-way associative cache, and our model is not capable to reduce many off-chip misses as we can see in figure 8.10.

The big workload has the best result achieving a 5% speedup respect the shared model. In this case our model reduce significantly the off-chip misses without increasing very much the misses satisfied by the neighboring caches. Figure 8.12 shows us the distance evictions that is closer than the distance of evictions in the centralized model. This leads the miss latency of the distributed model, observable in figure 8.13, to be much lower than the average miss latency of the original adaptive scheme.

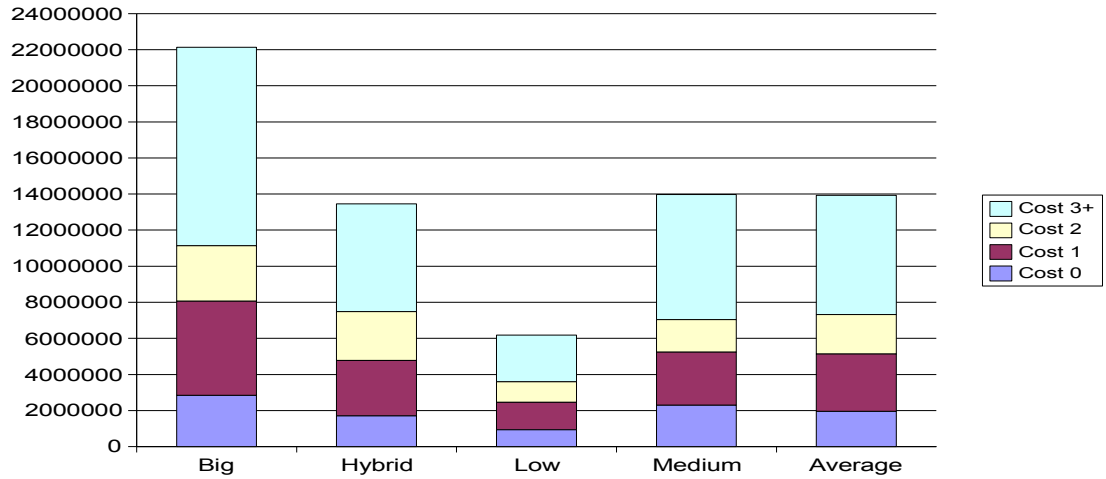


Figure 8.11: Distance of the evictions in the 8-cores Dybdahl's implementation.The cost expresses the number of hops in the internconnection network.

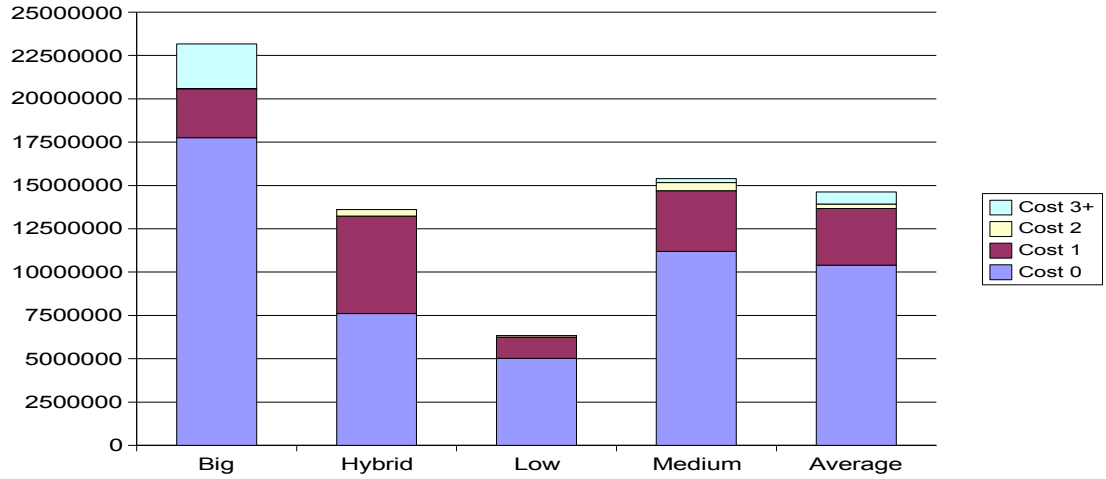


Figure 8.12: Distance of the evictions in our 8-cores model. The cost expresses the number of hops in the internconnection network.

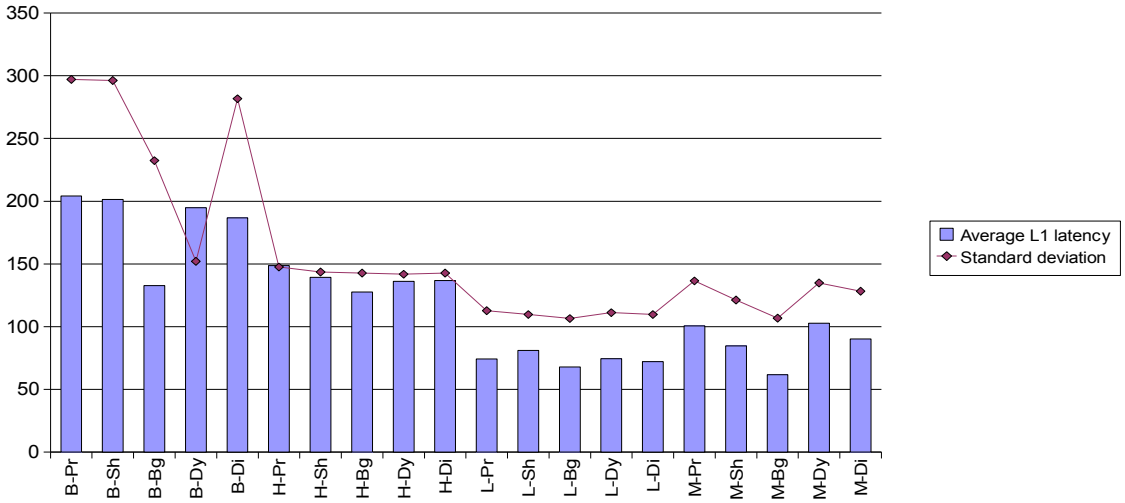


Figure 8.13: L1 cache average miss lantency in cycles.

8.4. The 16-cores configuration

As was explained in section 7, for the 16-cores architecture we introduce a new workload called huge with 8 memory-intensive applications running twice each one.

The Dybdahl's scheme implemented have a bad behavior in this architecture, observe Figure 8.14. For the hybrid and the low workloads it maintains the same performance than the private model, but for the big and the huge workloads its performance is degraded. These results are influenced by the use of the harmonic mean that aggravate the impact of negative speedups. As we commented in the previous section, this centralized model outperforms the IPC of certain applications at expenses of important performance degradation in other applications. Thus, it have a poor isolation.

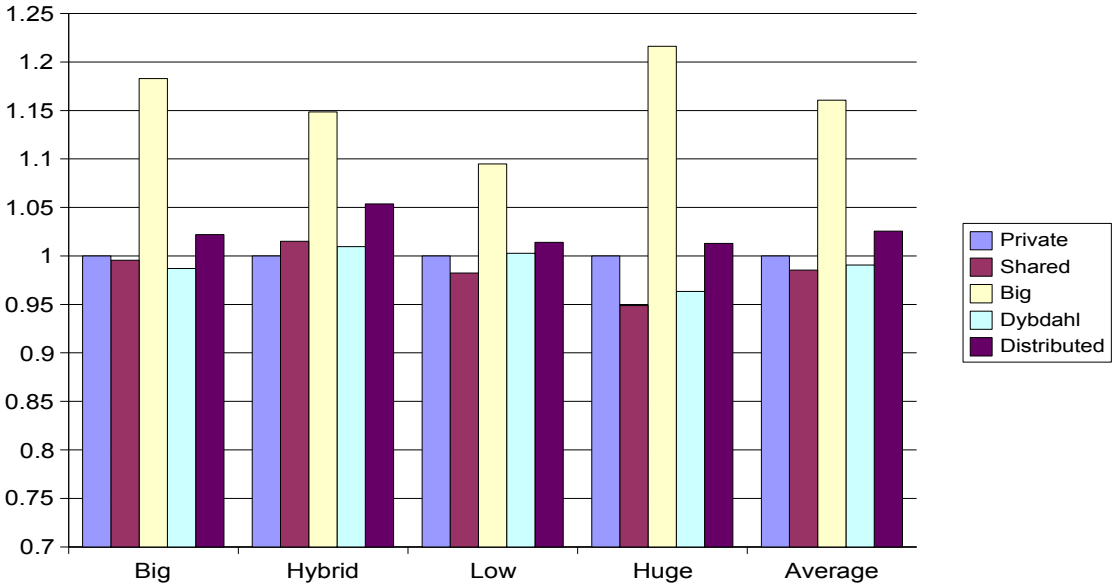


Figure 8.14: Harmonic mean of the speedups relative to the private model for the 16-cores configuration for the different workloads.

Two are the main problems of Dybdahl's architecture: the pollution in the shared partition and the distance of evictions. We have already discussed the distance of eviction problem, see figure 8.17 for the 16-cores architecture. Pollution in the shared partition is due to the use of the LRU stack. Even with the quota limitation per core, the shared partition behaves likes a fully shared partition, as more cores share the partition higher will be the number of interferences among applications. As the blocks in the shared partition that belongs to cores with few assigned blocks will be replaced firstly, the performance of the cores with lower miss rates will be degraded firstly due to the pollution.

In figure 8.15 we can see how in the big workload, for example, the off-chip misses per thousand instructions could not be reduced, besides the introduced extra misses.

On the other hand, the proposed adaptive scheme do not suffer from these problems. The evicted blocks are sent to closer caches, figure 8.17, and the possible pollution is also limited to the closer neighboring caches. In the 16-core architecture, see Figure 8.14, we achieve a 3% speedup in average compared with the private model. Moreover, our proposal shows a strongly robustness with no performance degradation in any case.

If we look at the big workload we could see a case to be clarified. We can see how our model outperforms the shared model by near a 3%. Figure 8.18 shows that the number off-chip misses in our model, however, is slightly bigger. Figure 8.15 shows that our model have a lower avareage miss latency. To this controversies, we can clarify that number of misses per thousand instructions presented is a bit misleading in this particular case. The amounts of work that each configuration does are different, due to this, the misses per thousand instructions values reflect the misses in different sets of instructions and we can not directly compare them. In the appendix A we can see how for big workload the shared model speedups certain non-memory-bound applications while our distributed scheme achieve high improvements in applications with lower IPC. This produces that the shared workload executed mode instruction some applications and our model executes more instructions of a different model (*i.e.*, the simulation window has changed somewhat between the two executions).

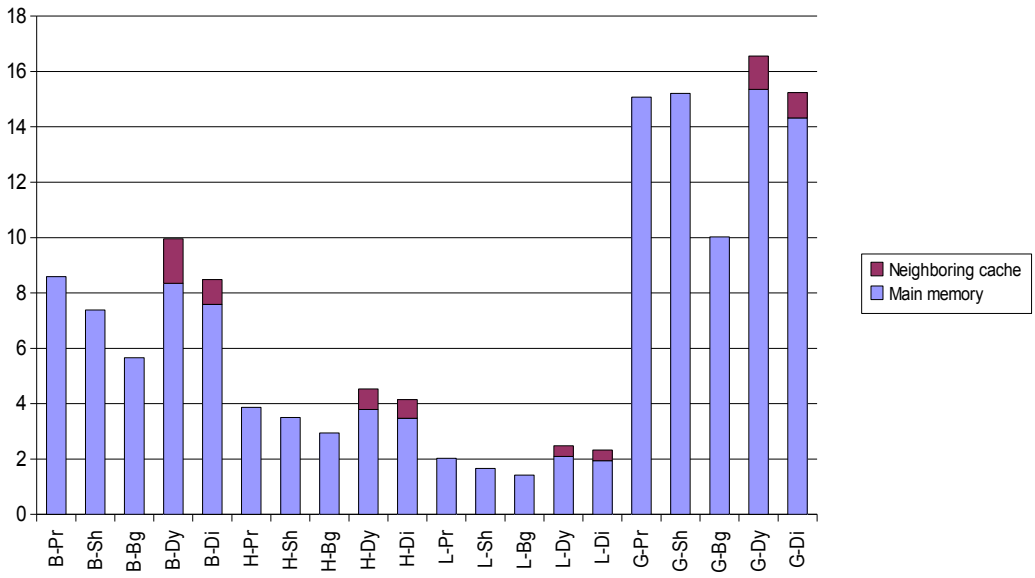


Figure 8.15: Misses per thousand instruction in the last-level cache for each workload and model configuration (expressed W-M). For the adaptive models we separate the off-chip misses and the misses to neighboring caches.

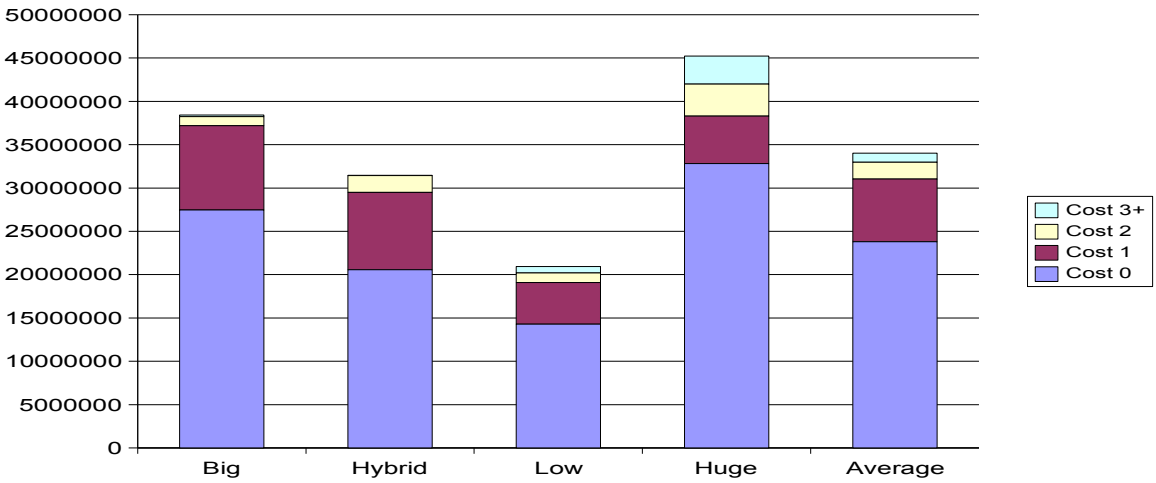


Figure 8.16: Distance of the evictions in our 16-cores model. The cost expresses the number of hops in the interconnection network.

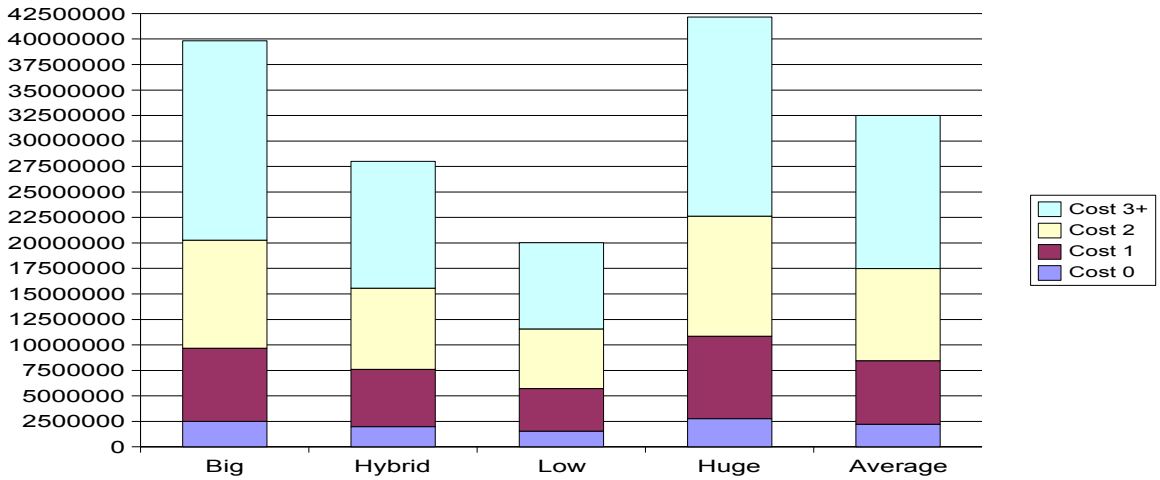


Figure 8.17: Distance of the evictions in the 16-cores Dybdahl's implementation. The cost expresses the number of hops in the interconnection network.z

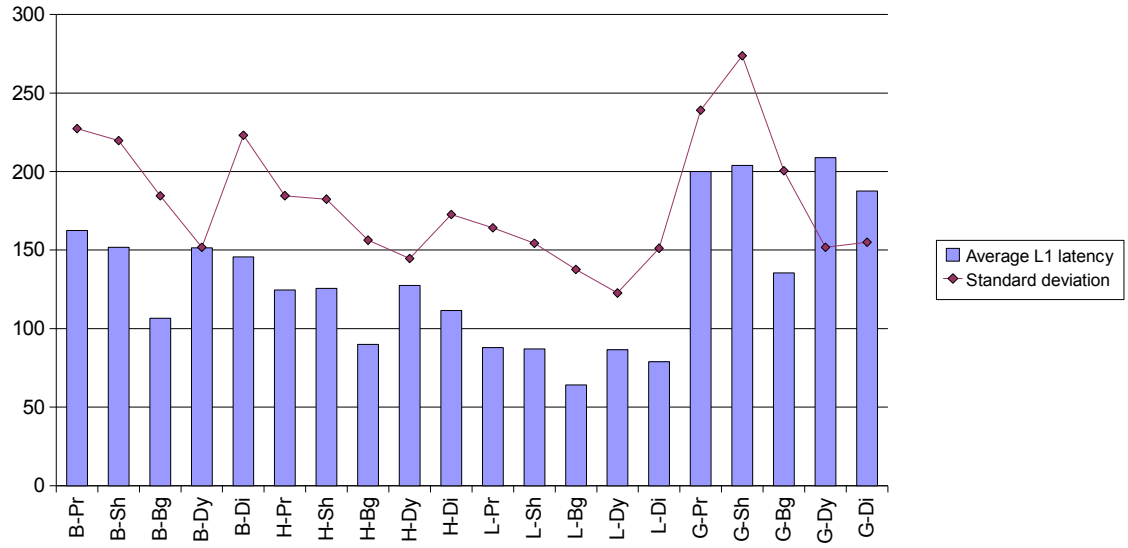


Figure 8.18: L1 cache average miss latency in cycles.

8.5. Placement of applications

We have discussed in chapter 6 the variations on the performance that can introduce the placement of the applications on our mesh-CMP architecture. In this section we want to study the effect of placing applications with different miss rates in different cores in the 16-cores architecture. To this, we have created 3 additional workloads for each set of applications, that is to say, three different placements for the big, the hybrid and the low workloads (unfortunately, we have had problems with the huge checkpoints).

We created artificially this workloads using the Solaris command `procbind` that allows us to assign a specific application to a specific processor. We build a good and a bad situation that we can see in figure 8.19, and a new random placement. To do so, we classify the applications by its miss rate. In the good situation, the four applications with more misses run in the diagonal and the next four near the corner, leaving room for the shared partition in the middle. In the bas situation the four applications with high miss rates are allocated vertically on a side, followed by the next four applications and then the rest. In this case, the shared partition will be on the opposite side.

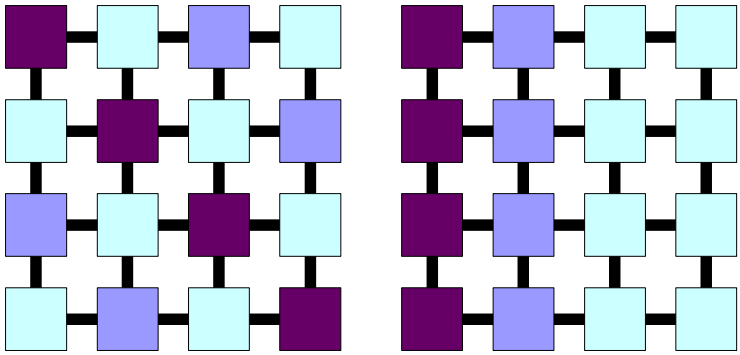


Figure 8.19: Placement for applications in the good and bad workloads. Darker colors represent higher miss rates.

Figure 8.20 show the performance results for the workloads using the set of applications *big*. We can see how different workloads present notably different results. To be short, we will present this results in stock charts that show in columns the average values (left Y-axis) and the ranges of values with a

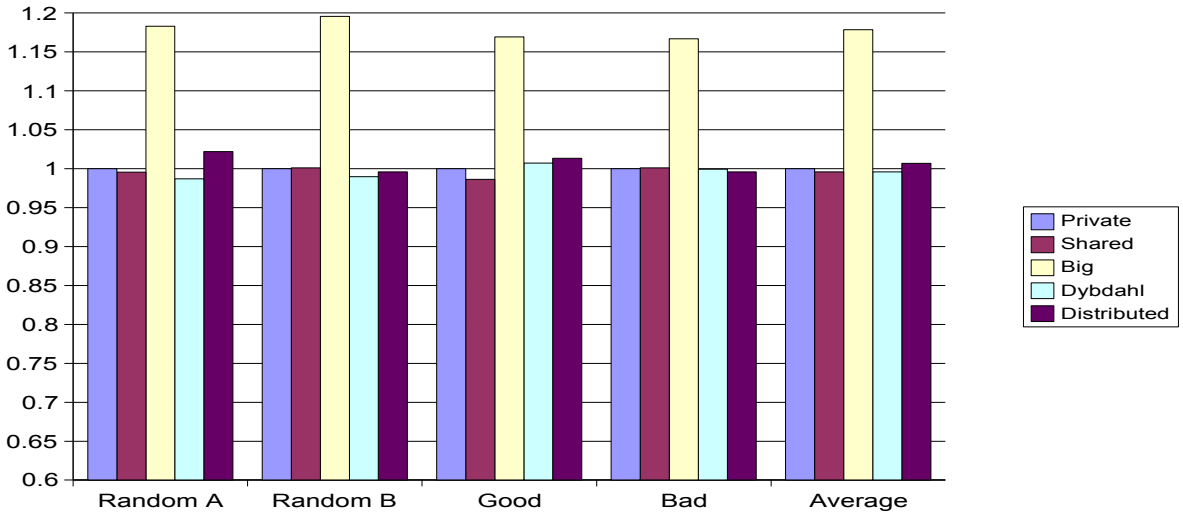


Figure 8.20: Performande of different workloads with the set of applications big.

vertical line (right Y-axis). Figures 8.21, 8.22 and 8.23 presents the variability in performance of the applications sets big, hybrid and low.

Our model present a 2% variation in performance in big and low sets, and a 5% in the hybrid set. These variations raise the need to consider the placement of cores in future work.

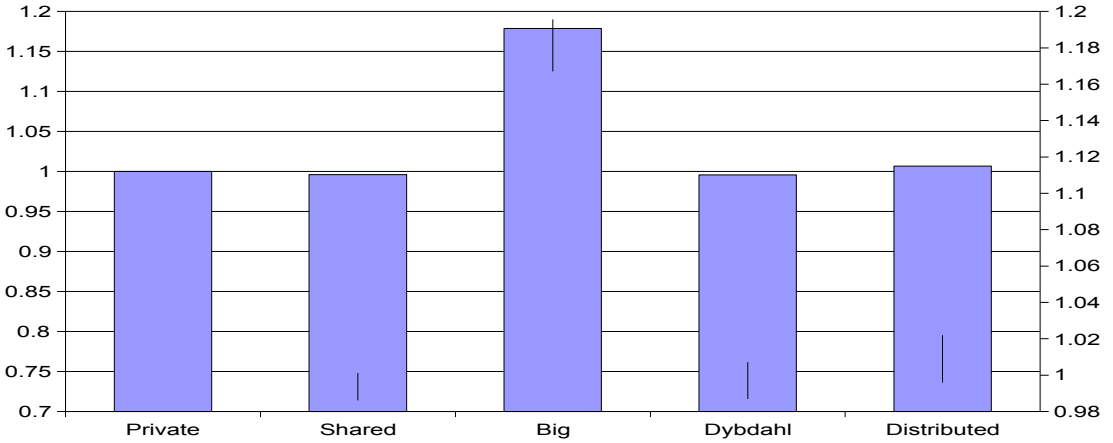


Figure 8.21: Performance variability in the set of applications big.

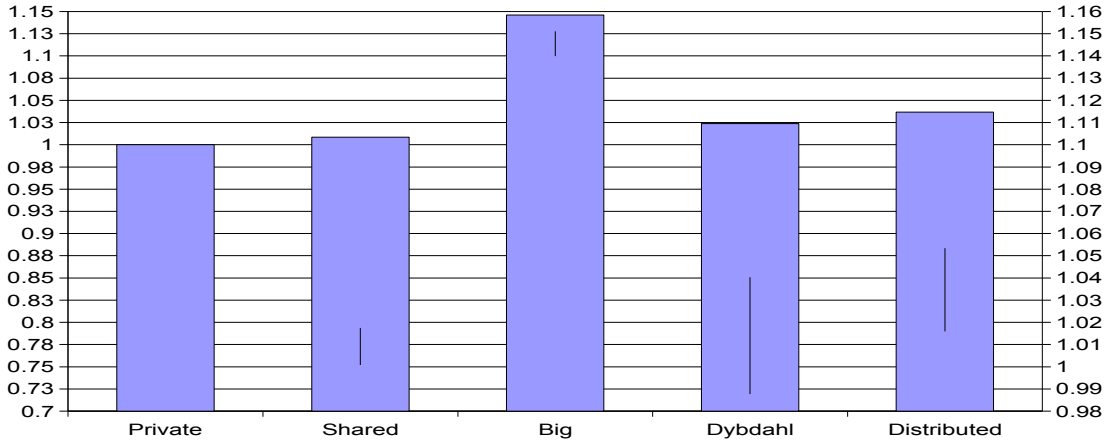


Figure 8.22: Performance variability in the set of applications hybrid.

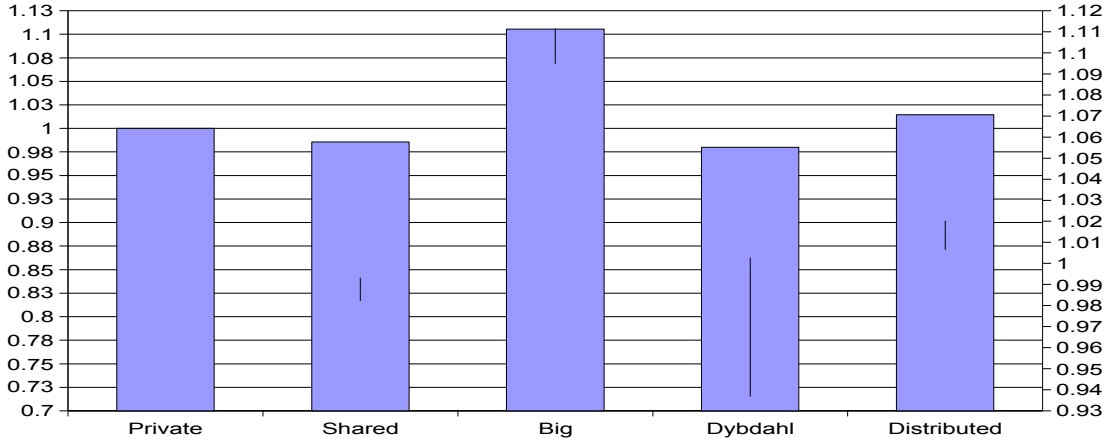


Figure 8.23: Performance variability in the set of applications low.

8.6. Implementation cost

The implementation cost of Dybdahl's partitioning mechanism, due the extra storage, is estimated in

$$log_2 p \times b \; + \; 3 \times p \times w \; + \; 0.6 \times s \times p \times t \text{ bits}$$

where p the number of cores, s the number of sets, b the number of blocks, t the number of bits per tag, and w the number of bits of a register. Dybdahl supposed the reduction of the shadow tags table ($s \times p \times t$ bits) monitoring only the 6% of the sets, that he estimates sufficient for calculate the cache partitioning.

We have implemented this reduction in our 16-core architecture to estimate the effects on the performance of our system. We only monitor one of every 16 sets. Figure 8.24 shows the results. The centralized model has in average a slightly variation due to the reduction of the shadow tags. Our proposed model presents a bigger performance degradation in average. The highest difference is in low workload. We can attribute this difference to a worst behavior in the estimation of a workload with very low miss rates. At all, the reduction of shadow tags offers an important hardware saving with a close performance and must be considered. The monitoring of only a 6% of the shadow tags, in our model, would also reduce the number of messages between last-level caches.

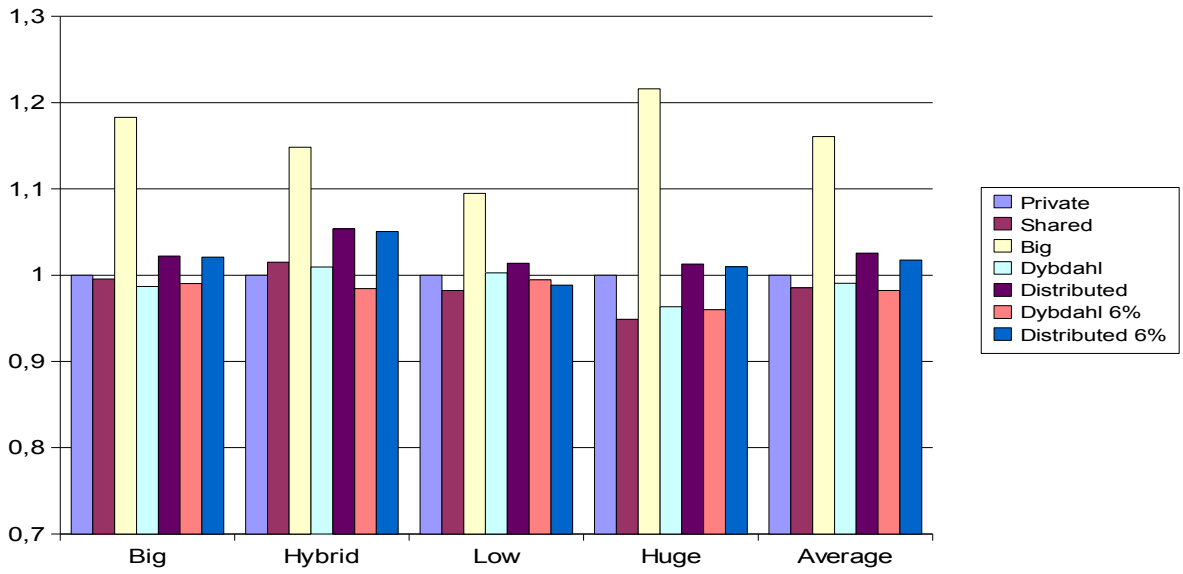


Figure 8.24: Performance variability using a reduced set of shadow tags per workload.

9. Conclusions and future work

The number of cores per chip will continue growing for the next years. The bandwidth requirements of these highly parallel processors scales linearly putting high pressure on the memory hierarchy that limits the scalability of the new manycore processors. Several solutions have been proposed to solve this problem, some of them using adaption as the way to ensure a good performance and the scalability of the system on different workloads.

Dybdahl and Stenstrom [11] proposed an adaptive shared/private NUCA cache partitioning scheme with an interesting quota mechanism to ensure a controlled cache sharing degree. The main goal of this mechanism is to reduce the total number of off-chip misses. When we scale this model to architectures with more than four cores three are the main problems: the implementation cost, the pollution in the shared partition and the distance of evictions. This centralized model uses a global LRU stack that is unmaintainable due to its time and area needs. Moreover, using only the LRU stack we do no take into account the distance of the evicted blocks and their new placements, and the pollution in the shared partition is increased degrading the performance of the cores with lower miss rates.

To solve all these problems we proposed a novel distributed adaptive scheme based on a *greedy* approximation of Dybdahl's technique. Our proposal presents a regular structure easy to implement and highly scalable. Sending the evicted blocks to the neighboring caches in a round robin order it reduces the distance of the replacements and maintains a lower and less-focused pollution. We have also seen that our greedy approach affects the repartitionig policy becoming more conservative.

Our proposal have the best performance in all three architectures for all four configurations. In the implemented 4- and 8-cores architectures our model outperforms by a 3% the private model. In the 16-cores architecture it achieves a 2% speedup respect the private model and a 3% speedup respect the shared model. We must consider that these small overall values really represent big improvements of the performance of one or more cores. Our implementation also present a high robustness.

The exploration of the effects of the placement of the applications has showed a considerable variability in the systems performance. A detailed study of these effects must be considered as part of our future work. Also interesting could be the study of a possible interaction or management of the placement of the applications by the operating system.

This thesis is focused in the study of the memory hierarchy in presence of co-runner applications. Part of our future work will be focused on the study of this scheme running parallel applications with sharing of data. This case would be more complex due to the need of data migratory policies.

The use of a distributed scheme makes also interesting the study of heuristic routings, to find the best paths, and different interconnection networks.

To conclude, we can say that our proposal improves the performance of the private and shared models and solves the problems of the Dybdahl's scheme. Our solution performs particularly well when it works with applications with high miss rates and shows a strong robustness in the other workloads. These factors and its terascale capability make our novel distributed scheme an interesting technique to keep working on it.

10. References

[1] H. Al-Zoubi, A. Milenkovic and M. Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. *Proceedings of the 42nd annual Southeast regional conference*, 2004.

[2] M. Azimi, N. Cherukuri, D. N. Jayasimha, A. Kumar, P. Kundu, S. Park, I. Schoinas, and A. S. Vaidya. Integration Challenges and Tradeoffs for Tera-scale Architectures. Intel Technology Journal, Volume 11, Issue 3, 2007.

[3] B.M. Beckmann, M.R. Marty and D.A. Wood. ASR: Adaptive Selective Replication for CMP Caches. International Symposium on Microarchitecture, 2006

[4] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCauley, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb. Die Stacking (3D) Microarchitecture. International Symposium on Microarchitecture, 2006.

[5] B. Black, D. W. Nelson, C. Webb, and N. Samra. 3D Processing Technology and Its Impact on IA32 Microprocessors. International Conference of Computer Design, 2004.

[6] S. Borkar, N. P. Jouppi, and P. Stenstrom. Microprocessors in the Era of Terascale Integration. Proceedings of the conference on Design, Automation and Test in Europe, 2007.

[7] D. Chandra, F. Guo, S. Kim and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. International Symposium on High-Performance Computer Architecture, 2005.

[8] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. International Symposium on Computer Architecture, 2006.

[9] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures. MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, 2003.

[10] S. Cho and L. Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. International Symposium on Microarchitecture, 2006.

[11] H. Dybdahl and P. Stenstrom. An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. International Symposium on High-Performance Computer Architecture, 2007.

[12] P. S. Magnusson *et al.* Simics: A Full System Simulation Platform. IEEE Computer 35(2), 2002.

[13] A. Fedorova, M. Seltzer, and M. Smith. Improving Performance Isolation on Chip Multiprocessors Via an Operating System Scheduler. International Conference on Parallel Architectures and Compilation Techniques, 2007.

[14] L. Hammond, V. Wong, M. Chen, B.D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. International Symposium on Computer Architecture, 2005.

[15] J. Held, J. Bautista, and S. Koehl. From a Few Cores to Many: A Tera-scale Computing Research Review. Intel White Paper, 2007.

[16] J. Hennessy and D. Patterson. Computer Architecture: a quantitative approach. Morgan Kaufmann, 4th edition, 2006

[17] Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support For Lock-Free Data Structures. International Symposium on Computer Architecture, 1993.

[18] E. Herrero, J. Gonzalez, and R. Canal. Distributed Cooperative Caching. Parallel Architecture and Compilation Techniques, 2008.

[19] J. Huh, D. Burger, and S. W. Keckler. Exploring the Design Space of Future CMPs. International Conference on Parallel Architectures and Compilation Techniques, 2001.

[20] B. Jacob, S. Ng and D. Wang. Memory Systems: Cache, DRAM, Disk. Morgan Kaufmann, 2007.

[21] C. Kim, D. Burger, S. W. Keckler. NUCA: A Non-Uniform Cache Access Architecture for Wire-Delay Dominated On-Chip Caches. International Symposium on Microarchitecture, 2004.

[22] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling. International Symposium on Computer Architecture, 2005.

[23] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. Symposium on Principles and Practices of Parallel Programming, 2006.

[24] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Transactions on Computers, 1979.

[25] K. Luo, J Gummaraju and M. Franklin. Balancing thoughput and fairness in SMT processors. International Symposium on Performance Analysis of Systems and Software 2001.

[26] 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. Multifacetâ€™s General Execution-driven Multiprocessor Simulator (GEMS) Toolset. SIGARCH Computer Architecture News, 2005.

[27] A. McDonald, J. Chung, B. Carlstrom, C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. International Symposium on Computer Architecture, 2006.

[28] J. Merino, V. Puente, P. Prieto, and J.A. Gregorio. SP-NUCA: A Cost Effective Dynamic Non-Uniform Cache Architectures. SIGARCH Computer Architecture News, 2008.

[29] K. E. Moore, J. Bobba, Michelle J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Logbased Transactional Memory. International Symposium on High-Performance Computer Architecture, 2006.

[30] S. Mysore, B. Agrawal, S. Lin, N. Srivastava, K. Banerjee, and T. Sherwood. Introspective 3D Chips. International Conference on Architectural Support for Programming Languages and Operating Systems, 2006.

[31] L. A. Polka, H. Kalyanam, G. Hu, and S. Krishnamoorthy. Package Technology to Address the Memory Bandwidth Challange for Tera-Scale Computing. Intel Technology Journal, Volume 11, Issue 3, 2007.

[32] K. Puttaswamy and G. H. Loh. Implementing Caches in a 3D Technology for High Performance Processors. International Conference of Computer Design, 2005.

[33] M. Quereshi and Y. Patt. Utility-Based Cache Partitioning: A low-overhead, highperformance, runtime mechanism to partition shared caches. International Symposium on Microarchitecture, 2006.

[34] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. SIGARCH Computer Architure News, 2006.

[35] N. Rafique, W. Lim, and M. Thottethodi. Architectural Support for Operating System Driven CMP Cache Management. International Conference on Parallel Architectures and Compilation Techniques, 2006.

[36] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional Memory for and Operating System. International Symposium on Computer Architecture, 2007.

[37] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and Managing Transactional Memory in an Operating System. Symposium on Operating Systems Principles, 2007.

[38] B. Saha, A. Adl-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. International Symposium on Microarchitecture, 2006.

[39] N. Shavit and D. Touitou. Software transactional memory. Symposium on Principles of Distributed Computing, 1995.

[40] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. SchererIII, and M. F. Spear. Hardware Acceleration of Software Transactional Memory. Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, 2006.

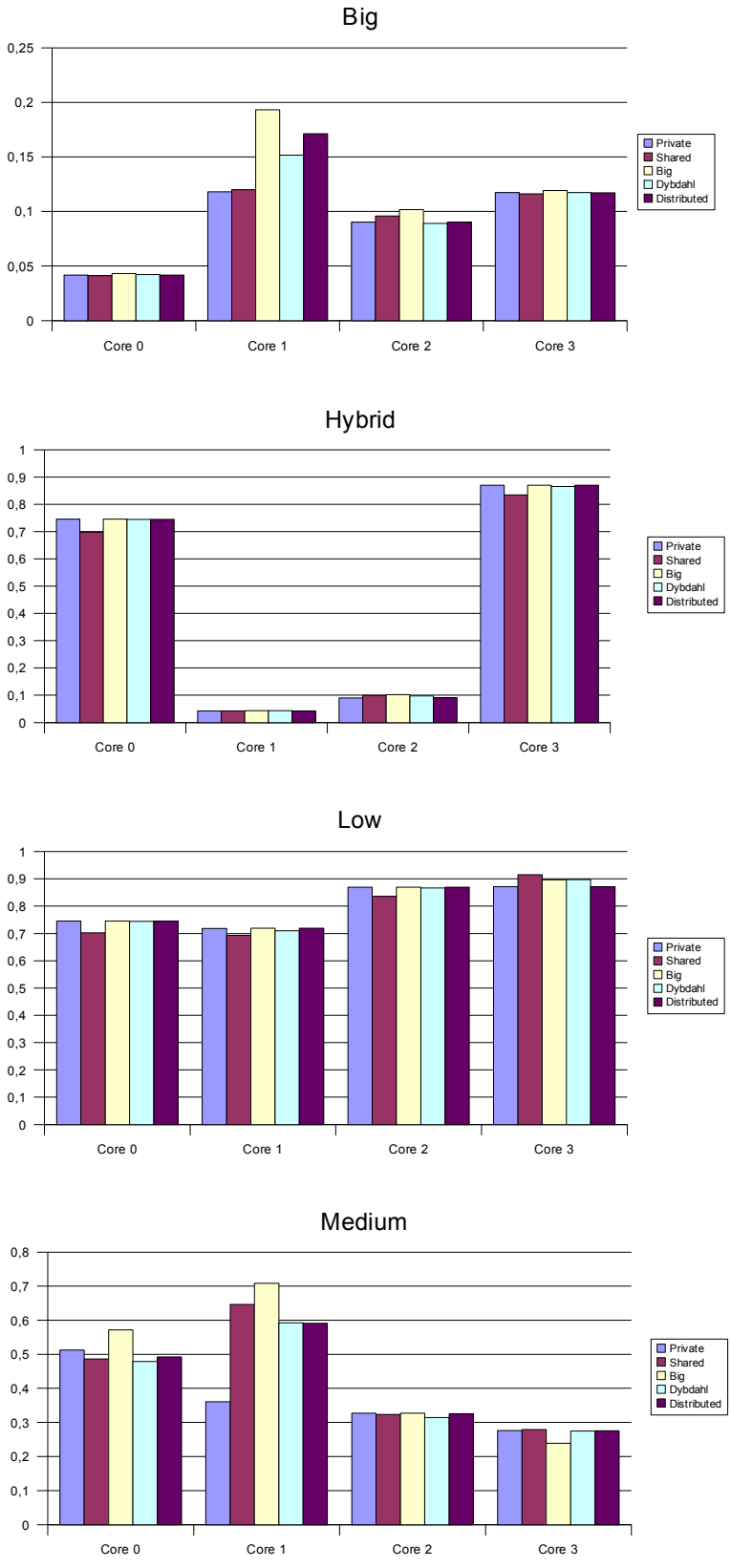
[41] E. Speight, H. Shafi, L. Zhang, and R. Rajamony. Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors. International Symposium on Computer Architecture, 2005.

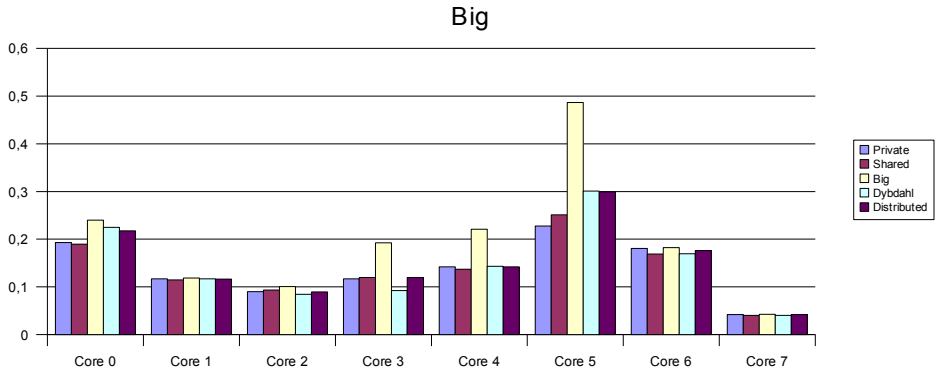
[42] R. Subramanian, Y. Smaragdakis, and G. H. Loh. Adaptive Caches: Effective Shaping of Cache Behavior to Workloads. International Symposium on Microarchitecture, 2006.

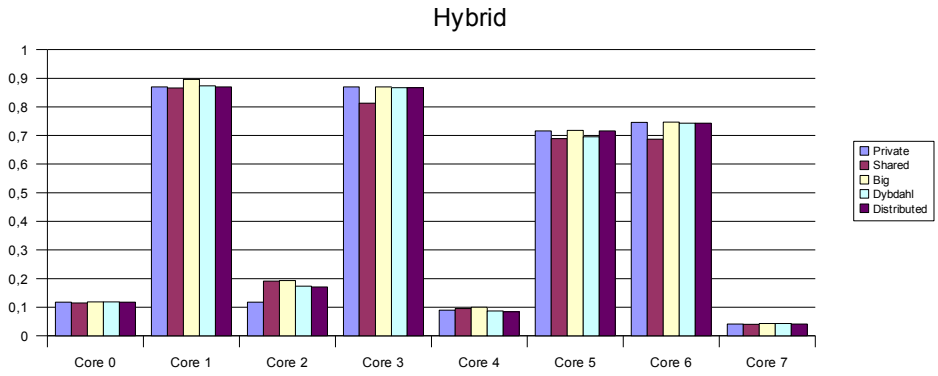
[43] J. Vera, F.J. Cazorla, A. Pajuelo, O.J. Santana, E. Fernandez and M. Valero. Measuring the Performance of Multithreaded Processors. SPEC Benchmark Workshop, 2007.

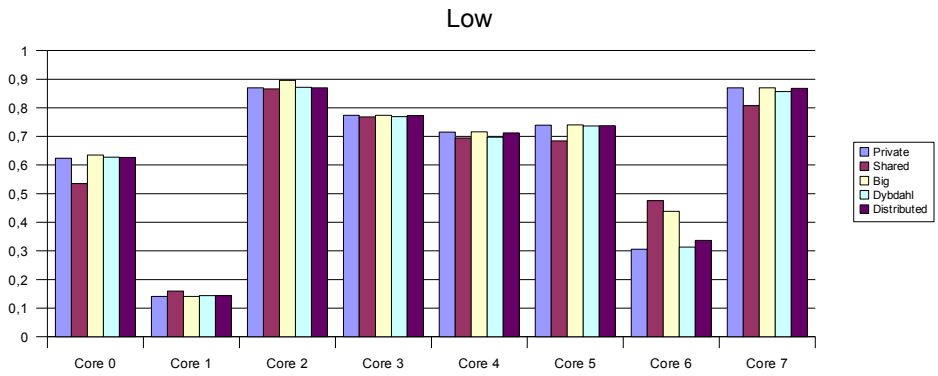
[44] AMD64 Architecture Programmer's Manual, Volume 2: System Programming. Advanced Micro Devices, September 2007.

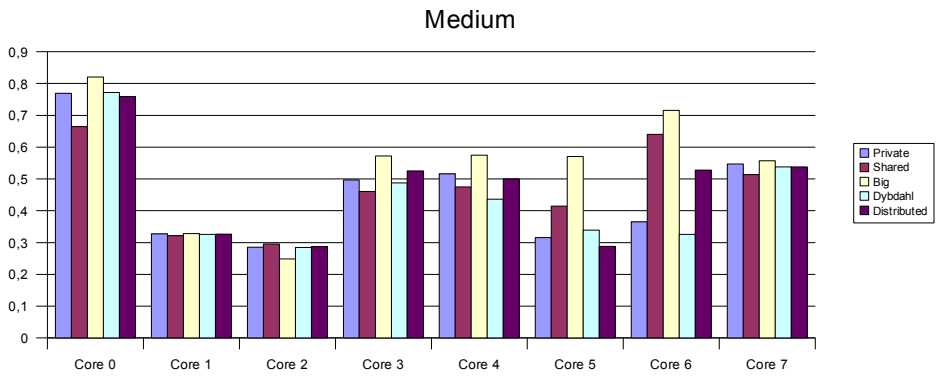
Appendix A. Instructions per cycle

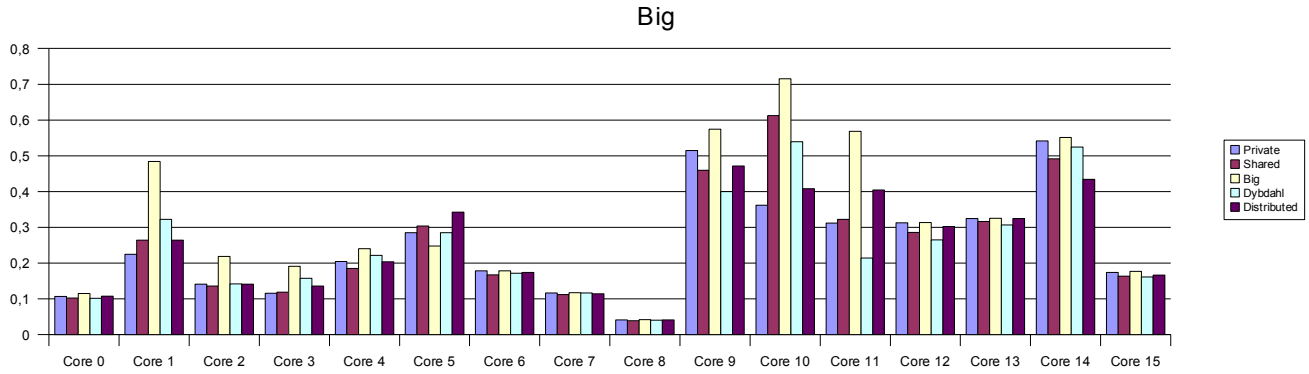


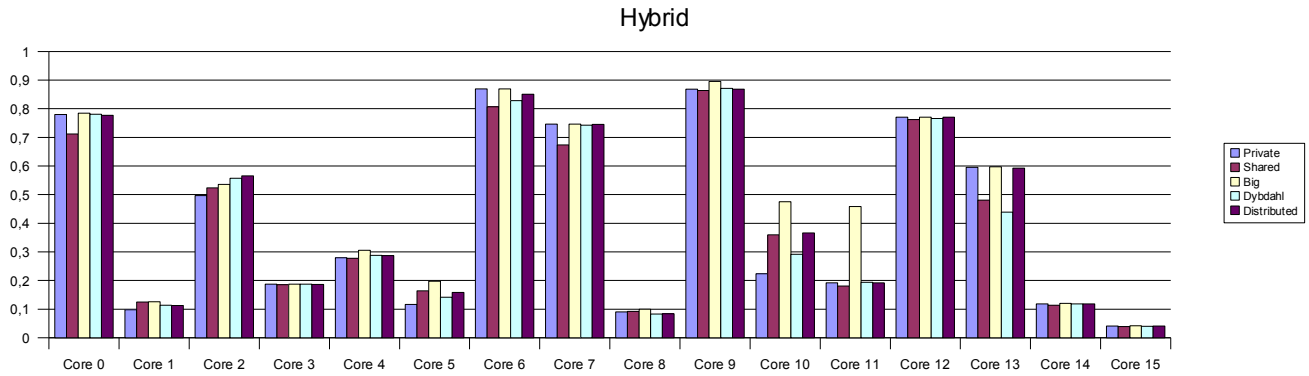

Big
0,6
0,5
0,4
0,3
0,2
0,1
0
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
Private
Shared
Big
Dybdahl
Distributed

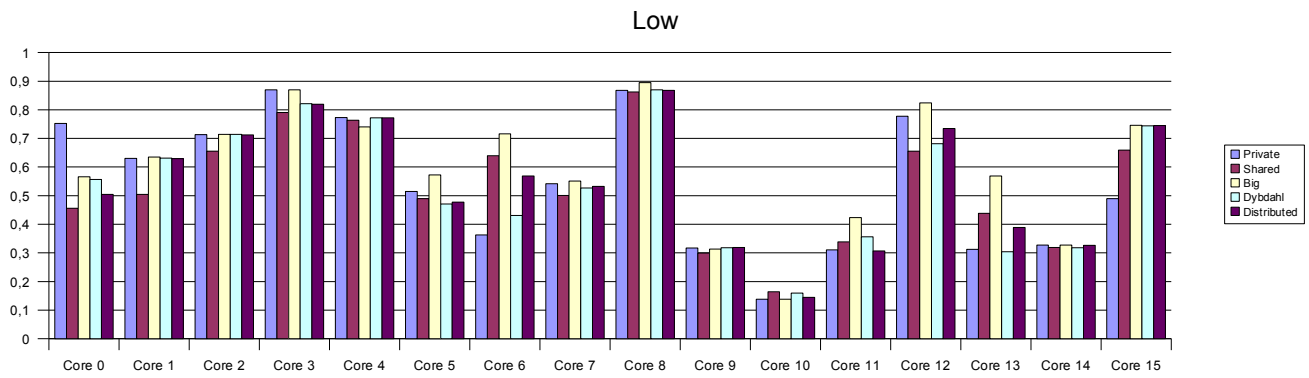

Hybrid
1
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
Private
Shared
Big
Dybdahl
Distributed

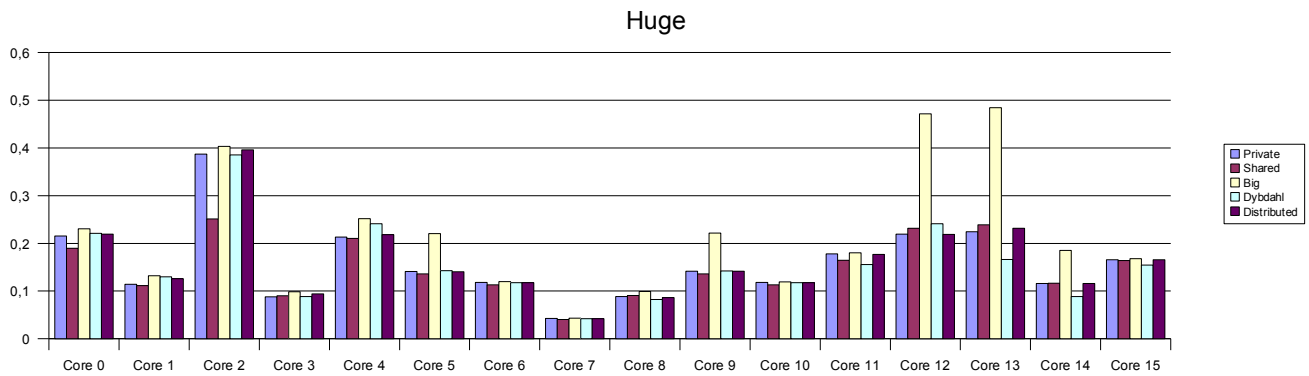

Low
1
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
Private
Shared
Big
Dybdahl
Distributed


Medium
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
Private
Shared
Big
Dybdahl
Distributed


Big
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
Core 8
Core 9
Core 10
Core 11
Core 12
Core 13
Core 14
Core 15
Private
Shared
Big
Dybdahl
Distributed


Hybrid
1
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
Core 8
Core 9
Core 10
Core 11
Core 12
Core 13
Core 14
Core 15
Private
Shared
Big
Dybdahl
Distributed


Low
1
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
Core 8
Core 9
Core 10
Core 11
Core 12
Core 13
Core 14
Core 15
Private
Shared
Big
Dybdahl
Distributed


Huge
0,6
0,5
0,4
0,3
0,2
0,1
0
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
Core 8
Core 9
Core 10
Core 11
Core 12
Core 13
Core 14
Core 15
Private
Shared
Big
Dybdahl
Distributed