CASTELL: A HETEROGENEOUS CMP ARCHITECTURE SCALABLE TO HUNDREDS OF PROCESSORS

Felipe Cabarcas Jaramillo

ADVISORS:

Alex Ramirez Bellido
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
Mateo Valero Cortés
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center

Submitted to the Departament d'Arquitectura de Computadors
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy on Computer Architecture
at the
Universitat Politècnica de Catalunya

July 2011

A Vero, Pablo, Dani, Diva, Germán

Acknowledgments

To my advisors Alex Ramirez and Mateo Valero without their support this project had not been possible.

The help of other students was very important for me. To Pieter Bellens, for his collaboration on the CellSs. To all who have been involved on the simulator especially Carlos, Augusto and Milan.

I deliberately left Alex Rico out of the list of the simulator, because he deserves a very special mention; we have been together since the beginning, since the first week, and he has gave me hours of technical discussions and his friendship.

Thanks to Xevi, Oriol, Miquel and Alex for making lunch time enjoyable and for their friendship.

To all the projects that have gave support: the European Network of Excellence HIPEAC-2 (ICT-FP7-217068), the ENCORE project (ICT-FP7-248647) and the IBM-BSC MareIncognito project. Program AlBan, the European Union Program of High Level Scholarships for Latin America scholarship No. E05D058240CO.

Finally, to the two institutions that gave me most of the financial support: the Barcelona Supercomputing Center and the Universidad de Antioquia that also gave me the time to come to Barcelona to pursue this degree.

Abstract

Technology improvements and power constrains have taken multicore architectures to dominate microprocessor designs over uniprocessors. At the same time, accelerator based architectures have shown that heterogeneous multicores are very efficient and can provide high throughput for parallel applications, but with a high-programming effort. We propose Castell a scalable chip multiprocessor architecture that can be programmed as uniprocessors, and provides the high throughput of accelerator-based architectures.

Castell relies on task-based programming models that simplify software development. These models use a runtime system that dynamically finds, schedules, and adds hardware-specific features to parallel tasks. One of these features is DMA transfers to overlap computation and data movement, which is known as double buffering. This feature allows applications on Castell to tolerate large memory latencies and lets us design the memory system focusing on memory bandwidth.

In addition to provide programmability and the design of the memory system, we have used a hierarchical NoC and added a synchronization module. The NoC design distributes memory traffic efficiently to allow the architecture to scale. The synchronization module is a consequence of the large performance degradation of application for large synchronization latencies.

Castell is mainly an architecture framework that enables the definition of domain-specific implementations, fine-tuned to a particular problem or application. So far, Castell has been successfully used to propose heterogeneous multicore architectures for scientific kernels, video decoding (using H.264), and protein sequence alignment (using Smith-Waterman and clustalW). It has also been used to explore a number of architecture optimizations such as enhanced DMA controllers, and architecture support for task-based programming models.

Contents

Acknowledgments i

Abstract iii

1 Introduction 1
- 1.1 Motivation 2
- 1.2 Objective 4
- 1.3 Thesis Overview 5
- 1.4 Document Structure 6
- 1.5 Historical Context 6

2 Programming Model 9
- 2.1 State of the Art 9
- 2.2 OmpSs 10
  - 2.2.1 Irregular task dependence graph applications 10
  - 2.2.2 CellSs 11
- 2.3 Conclusions 12

3 Castell 13
- 3.1 State of the Art 13
- 3.2 The Castell Architecture 15
  - 3.2.1 Master processors 16
  - 3.2.2 Worker processors 16
  - 3.2.3 Shared Last Level Cache 17
  - 3.2.4 Memory Controllers 17
  - 3.2.5 Synchronization Module 18
  - 3.2.6 Network On Chip 18
- 3.3 Memory Spaces 18
- 3.4 Conclusions 21

4 Experimental Methodology 29
- 4.1 Tasksim 29
- 4.2 Benchmarks 34

4.3 Additional applications . . . 35
4.3.1 H.264/AVC Decoder . . . 35
4.3.2 FASTA Smith-Waterman . . . 36
4.4 Conclusions . . . 36

5 Off-Chip Memory 37
5.1 SDRAM State of the Art . . . 38
5.2 Related Work . . . 39
5.3 Architecture . . . 41
5.3.1 Memory System . . . 42
5.3.2 Storage schemes . . . 43
5.4 Impact of Memory Latency . . . 44
5.5 Impact of Memory Bandwidth . . . 46
5.6 Impact of Memory Interleaving . . . 47
5.7 Conclusions . . . 49

6 Last Level Cache 51
6.1 Related Work . . . 51
6.2 Architecture . . . 52
6.3 Impact of Memory Latency . . . 53
6.4 Impact of Memory Bandwidth . . . 55
6.5 Cache Banks . . . 57
6.6 Cache Interleaving . . . 59
6.7 The Cache as a bandwidth filtering . . . 60
6.8 Conclusions . . . 60

7 On-Chip Synchronization 63
7.1 Related Work . . . 63
7.2 Hardware Semaphores . . . 65
7.2.1 Virtualization . . . 66
7.2.2 Active and Inactive Semaphores . . . 66
7.2.3 Semaphore waiting queue . . . 66
7.2.4 Required Hardware . . . 68
7.2.5 API . . . 68
7.2.6 ISA . . . 68
7.3 Use of Semaphores . . . 69
7.4 Conclusions . . . 69

8 Network On Chip 71
8.1 Related Work . . . 71
8.2 Design Space Exploration . . . 72
8.2.1 Workers per Cluster . . . 72
8.2.2 Number of Rings . . . 73
8.3 Conclusions . . . 74

9 Research Impact **77**
9.1 DMA++ 77
9.2 TaskSs 78
9.3 Bioinformatics 78
9.3.1 Smith Waterman 79
9.3.2 ClustalW 80
9.4 Conclusions 80

10 Conclusions **83**

Bibliography **87**

List of Figures **93**

List of Tables **97**

Chapter 1

Introduction

Hardware's natural parallelism and humans natural sequential thinking have influenced computer architecture in opposite directions. Hardware designers know that they can easily increase throughput, adding processing units to work in parallel. They also know that such systems have to be programmed, and since humans are better writing sequential than parallel programs, they have designed more sequential machines. Nonetheless, there has been some very successful parallel systems: supercomputers for large scientific applications and graphic processing units (GPUs) for computer graphics. Moreover, designers have used hardware parallelism at different levels to improve systems performance: VLIW, superscalar processors, SIMD functional units, etc. Most of the time, though, computer architects have struggled to make processors as fast as possible for sequential programs, making naturally parallel hardware behave like a sequential machine.

However, there is a paradigm shift on computer architecture from single- to multi-core[1] chips, motivated by the exhaustion of traditional design techniques and the increasing number of transistors available to designers. Whereas frequency scaling and instruction level parallelism (ILP) have reached their limit; Moore's law keeps motivating the industry, which keeps reducing transistor's size.

Parallel architectures are as old as computer systems; for example, early supercomputers, of the 70's, such as the CDC Star100, the Cray-I and Texas instrument's ASC were vector processors. Actually, until the introduction of microprocessors, most computing systems were parallel designs for scientific applications. Microprocessors and personal computers changed the focus of computer system designs, and for the last 40 years, computer architecture has mainly focused on sequential programs.

Chip multiprocessors (CMPs) were first proposed in 1996 by Olukotun et al.[34] in their work: "The case for single-chip multiprocessor". They observed that technology issues were going to limit performance of superscalar processors and that multi-processors were better than large uni-processors for parallel programs. They considered that simple processors could use faster clocks, and could work together to execute sequential programs as good as superscalars ones if they were faster, and could work together to execute sequential programs. Even though multi-processors have not been successful for sequential programs, most modern microproces-

[1]Core and processor will be use interchangeably throughout this thesis to describe an independent processing element, while chip or microprocessor will be used to describe an integrated circuit that contain one or more cores.

sors contain more than one core.

From the software side, parallel applications include both sequential and parallel parts with distinct characteristics each. In general, while parallel sections are usually the raw computation with simple control flow, sequential sections contain complex control flow. Sequential sections are, usually, related with scheduling, data preparation, result management, etc. This difference between sequential and parallel sections, favors heterogeneous designs, which have processors with different characteristics.

Castell is a computer architecture framework that can integrate hundreds of heterogeneous processors in a single chip, with scalable performance for parallel scientific applications, without increasing programming effort.

1.1 Motivation

Technology improvements have been reducing the size of transistors since the 70s, providing computer architects with increasing resources to improve their designs. Figure 1.1 shows the evolution of the number of transistors per chip of Intel's microprocessors; it can be seen that the number has increased following Moore's Law: the number of transistors duplicates every 2 years. Architects have used the increasing number of transistors, for example, to increase pipeline stages for higher instruction throughput; this has made conditional branches a problem, requiring additional transistors to build better branch predictors and hardware support to recover from miss predictions. Architects have also increased the size of caches; this way applications have larger storage closer to the processors. These are two examples that shows how computer architects have traditionally used the additional transistors to increase performance.

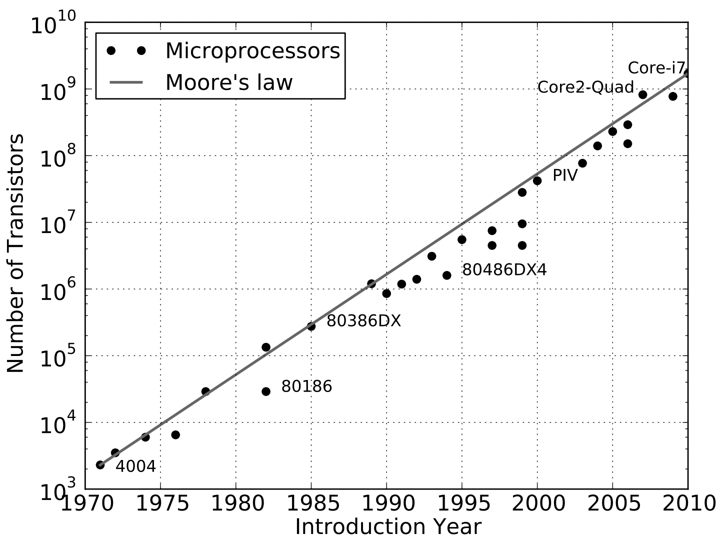


Figure 1.1: Number of transistors evolution of Intel's microprocessors and Moore's law.

However, the key factor for the throughput increase, of the last 40 years, has been transistors speed. Technology development not only has reduced the size of transistors; it has also made them faster. Figure 1.2 shows the frequency evolution of Intel's processors from 1990 to 2010. It can be seen that the switching frequency increased constantly until 2005, but it has stopped.

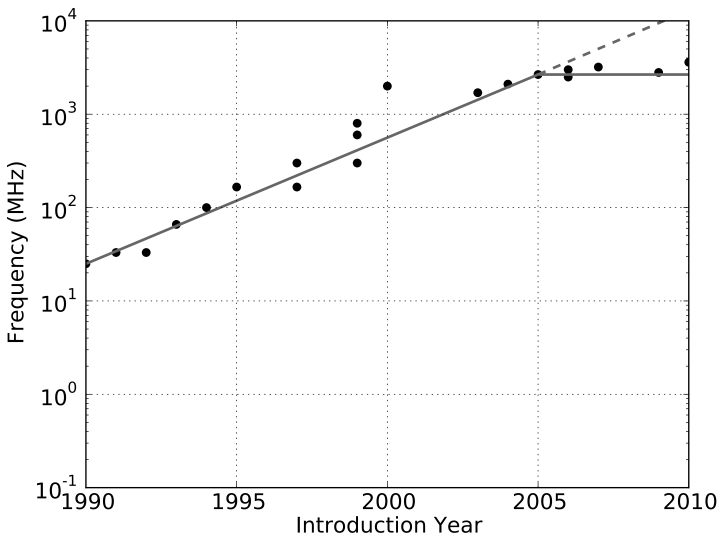


Figure 1.2: Intel's microprocessors frequency evolution.

Chip manufacturers can increase the speed of transistors further, but as it can be seen in Figure 1.3, the power density was also increasing ($P = CfV_{dd}^2$) with this and the size reduction[2]. The consequence is that chips are getting hotter and wasting more energy; furthermore, it is clear that a chip should not reach the power density of a nuclear reactor.

The question, then, is how to use the increasing number of transistors, which are not getting faster, to keep increasing throughput. The direct solution is to exploit natural parallelism of hardware. Most people in the community agree that chip multiprocessors is the solution to keep increasing performance of microprocessors. However, there is no agreement on the architecture that will be successful. Moreover, it is not likely to be a solution that fits all needs.

On one side of the spectrum we have ASICs; which are the fastest and more efficient designs for a single purpose. On the opposite side we have the general-purpose processors; which are highly programmable but not very efficient for one purpose. Midway between these two types of integrated circuits, we find special purpose processors; which are not as programmable as the general-purpose processors, but are more efficient for one purpose, even though they are not as efficient as ASICs.

Multi-core architectures proposals are as diverse as: general-purpose processors replicated several times sharing the last level cache; different types of processors with local memories and

[2]Graph adapted from: "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies", Fred Pollack, Intel Corporation, MICRO 32 Conference keynote, 1999.

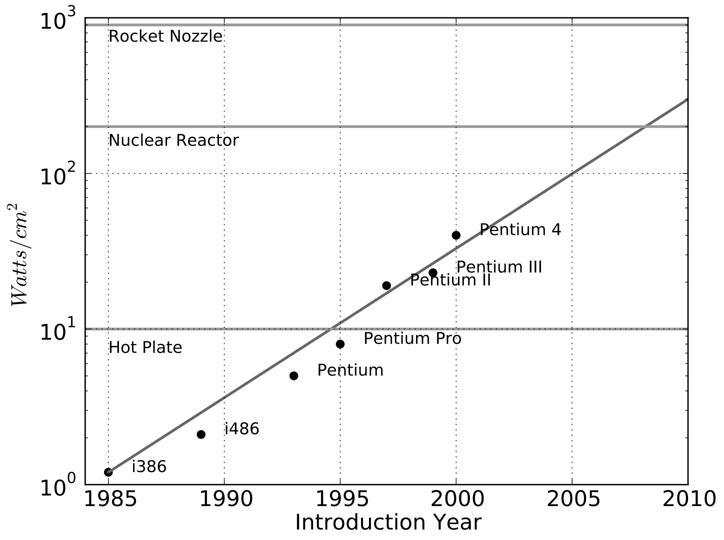


Figure 1.3: Intel's microprocessors power density evolution.

caches; or academic ones with many processors connected with a reconfigurable connecting fabric that, at compile time, routing is created depending on the data-flow of the program [26]. As could be expected, each architecture has some advantages and some disadvantages, but in most of the cases, the performance is directly proportional to the programmability effort.

Figure 1.4, which uses Pollack's rule to compare the performance of different architectures given several application characteristics. The rule says: Microprocessor performance increase due to microarchitecture advances is roughly proportional to the square root of the increase in complexity; which basically says that doubling the number of logical units only achieves 40% improvement. It can be seen, from the Figures, that heterogeneous architectures can have similar performance to the best homogeneous architecture in different scenarios.

The purpose of this thesis is to find a design that can be as efficient as special purpose processors, but with the programmability of general-purpose ones. Besides programmability, there are several key features that limit the performance of a parallel computer system: coherency of the memory modules, memory latency, memory bandwidth, and synchronization latency. This thesis addresses all of these features.

1.2 Objective

The main objective of this thesis is to propose a chip multiprocessor architecture framework with the performance of special-purpose processors and the programmability of general-purpose architectures.

The architecture should be general purpose and scalable, so it can adapt to different types of requirements and the increasing number of transistors.

Given that one on the main problems of any architecture is the gap between memory and processing performance, we study and propose ways to reduce or deal with this gap.

1.3 Thesis Overview

Castell is a heterogeneous multi-core architecture framework that combines general-purpose processors and accelerators with a bandwidth oriented memory system.

Software development is a problem for heterogeneous multiprocessors such as Castell. Therefore, we have chosen OmpSs, a task-based programming model, which is briefly described in Chapter 2. OmpsSs has shown promising results for parallel algorithms, providing performance and simplifying programmability—even in cases of irregular parallelism. It relies on a runtime system that, based on tasks inputs and outputs dependencies, dynamically schedules tasks depending on the available resources and manages data transfer across address spaces; this way programmers are not forced to divide the algorithms statically and decide how to schedule depending on expected resources.

With the programming model defined, we present Castell in Chapter 3. It is a combination of general-purpose processors for the runtime and accelerators for throughput computation.

In Chapter 4 we present the simulation infrastructure and the applications used to run the experiments. TaskSim provides a cycle accurate simulation infrastructure for heterogeneous multiprocessor experiments. We contributed to the design of TaskSim; which is a trace-based simulator, of multi-processor systems, that models: processors, caches, interconnection network, memory controllers, memory modules, etc.

While most architectures memory systems are designed to reduce memory latency, since this is the main cause of processor stalls; Castell's memory system is designed for high bandwidth. Chapters 5 and 6 show that memory latencies, several times larger than that of current SDRAMs, do not reduce performance of applications if double buffering is used for programming (OmpSs provides the double buffering automatically); but memory bandwidth of current systems can reduce significantly the performance. This is a crucial result for Castell which determines the bandwidth oriented design.

In Chapter 5 we study the storage schemes designed for bandwidth and the performance degradation of different memory interleaving. Then, in Chapter 6, we observe that the current state of the art SDRAM and number of chip pins are not enough to provide the bandwidth in the case of a 256-processor Castell, so we added a last-level cache that gives the required memory bandwidth. The bandwidth-oriented cache is distributed, shared and fine-grain interleaved, such that consecutive line-size accesses are distributed in different cache banks.

While memory latency gets hidden by the programming model and the use of the DMA engines, synchronization latency can degrade applications performance significantly on Castell. In Chapter 7, we introduce the high level synchronization mechanism of Castell: hardware semaphores. The architecture only needs to support some basic ISA instructions, in order to use this module. This hardware implementation provides the synchronization latency required.

The interconnection network on any multiprocessor is very important, since it is the way data and instructions move through the architecture. In Chapter 8 we show that the network-on-chip design is hierarchical to provide scalability and fair access to shared resources.

Finally, in Chapter 9 we present the impact of this thesis. Castell has been used to implement several parallel architectures: H.264 video decoder, Smith Waterman and ClustalW sequence aligners. It has also been used to propose hardware innovations: DMA++ a data transfer engine for unaligned transfers, and TaskSs a hardware accelerator for task management. In these works, Castell is used as the base architecture and I also provided support for simulations.

1.4 Document Structure

Figure 1.5 shows the structure of the thesis. The main objective is the design of a chip multiprocessor framework, which takes us to the first issue, how to program parallel applications for heterogeneous multicores. We chose OmpSs because it is a promising parallel programming model for heterogeneous architectures. The architecture framework is presented in Chapter 3 which is a consequence of the evaluation of the following chapters.

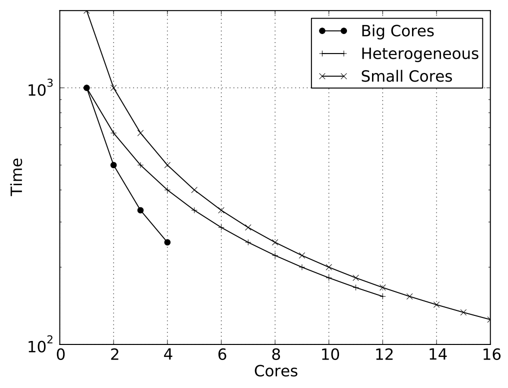
The memory system was designed to provide high bandwidth, Chapter 5 presents the off-chip memory design, and Chapter 6 the last level cache design. While the memory latency is not a problem, the synchronization latency is, so Chapter 7 presents the hardware solution. The glue to all the structures of the architecture is the interconnection which is presented in Chapter 8.

Finally, the impact of Castell is presented in Chapter 9, where we show the different implementations that have been made in our group with Castell as a framework.

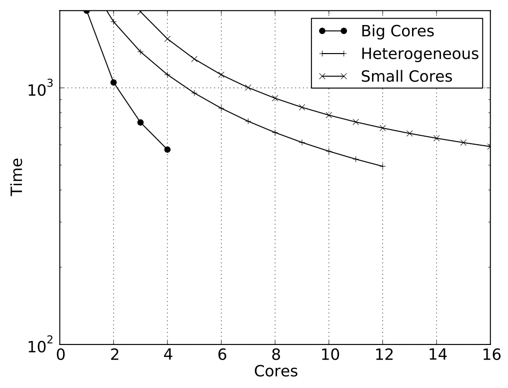
1.5 Historical Context

Figure 1.6 shows a schematic view of the environment and main influences of Castell since the thesis started. The first proposal of a single-chip multiprocessor was in 1996. IBM presented the first multi-core microprocessor, the POWER 4, on 2001, when it was clear the end of frequency scaling. In 2005, Intel and AMD presented their first multi-cores, and the consortium IBM-Toshiba-Sony developed a revolutionary microprocessor architecture for the PlayStation 3 console, the Cell/B.E. It showed good performance for gaming, but what surprised more was its high performance per Watt on scientific applications.

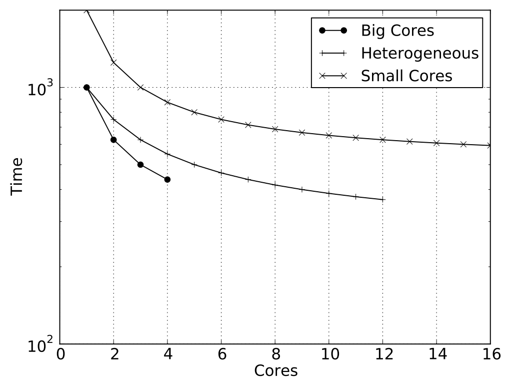
The Cell/B.E. inspired us to design Castell with the combination of OmpSs for programmability. OmpSs is a programming model developed at the Barcelona Supercomputing Center for parallel applications based on OpenMP with the addition of tasks features of the CellSs. Castell has contributed to important European research projects such as SARC, ENCORE; as well as to the BSC and IBM Mareincognito project.



(a) Fully parallel code.



(b) 10 percent serial code.



(c) 25 percent serial code.

Figure 1.4: Heterogeneous versus Homogeneous cores using Pollack's rule. Assuming a fix chip area, the graphs show a performance potential for different serial and parallel code mix of applications. we compare three types of architectures: big cores, small cores, and heterogeneous.

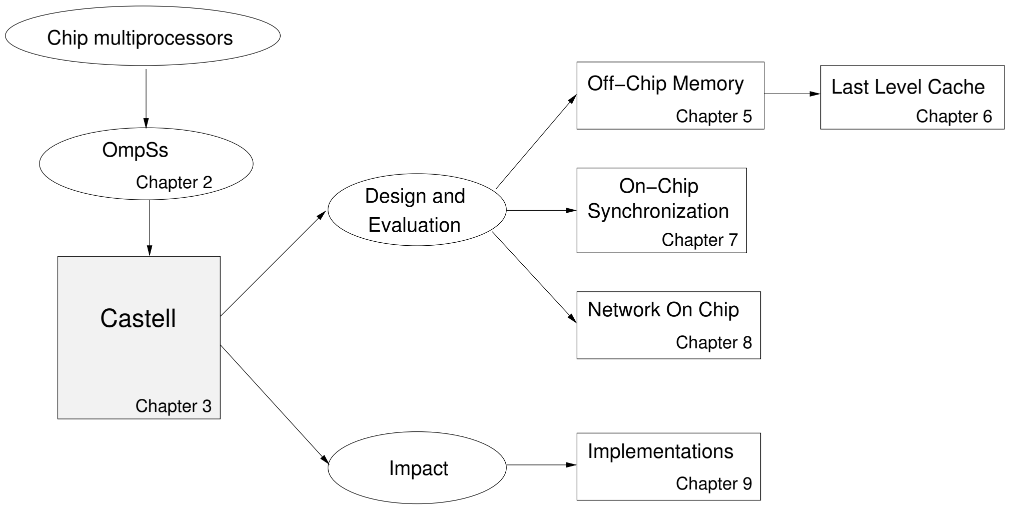


Figure 1.5: Thesis Overview.

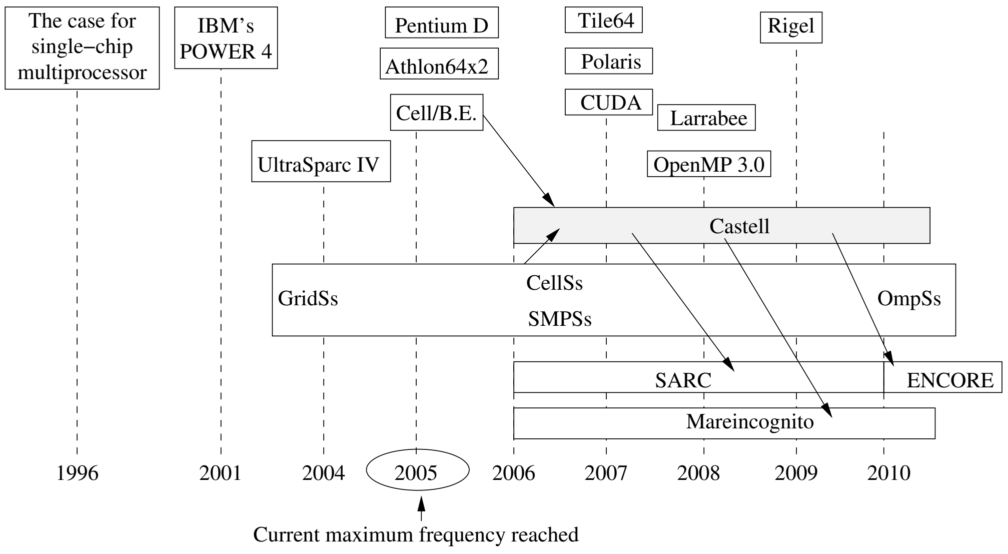


Figure 1.6: Castell environment and impact.

Chapter 2
Programming Model

Chip multiprocessors are becoming complex and heterogeneous, for example, Intel, AMD, NVIDIA, and IBM have microprocessors with general-purpose processors and programmable accelerators. Most of these designs target graphics with GPUs as accelerators. However, they are not easy to program.

A parallel program execution on a heterogeneous multiprocessor system not only requires the application to be divided into tasks; it also requires a correct mapping of tasks to processors, data distribution to optimize locality, task synchronization, etc. These features complicate programmability of heterogeneous systems. In this Chapter, we review OmpSs, the programming model used in this thesis and developed at the Barcelona Supercomputing Center.

2.1 State of the Art

The programmability of a multiprocessor system depends on many factors, such as the number of logical address spaces, the differences between processors, but mainly on how automatic is the process of mapping the application to the hardware. In other words, how much help the compiler requires, from the programmer, to generate code that efficiently uses the underlying hardware.

Large multiprocessor systems, such as supercomputers, are mostly programmed using the message passing interface (MPI) API. In these systems, with distributed memory, programmers not only have to divide applications in tasks, but they also have to distribute data to program messages for synchronization.

Shared memory systems are usually programmed using either OpenMP or Pthreads. OpenMP has been a successful API for shared memory model multiprocessor architectures in scientific applications. Based on compiler directives, it generates parallel loops, performs the required thread creation, and synchronization. While OpenMP is very good for regular applications, Pthreads is used for more irregular parallel applications. Pthread is a low-level API for thread creation and management. The programmer has to decide how to divide the application, launch tasks and synchronize them.

OpenMP 3.0 increases the parallelism that can be expressed with older versions. In addition to parallel loops, it allows parallel tasks. Therefore, independent tasks can be executed in parallel. However, OpenMP 3.0 still requires that tasks are independent. Prior to OpenMP 3.0, Cilk [5] extended C and gave the programmer the responsibility of expressing all parallelism and how tasks interact, the runtime was responsible for scheduling only.

Another important example is the unified parallel C (UPC), a low-level C programming extension. The programmer sees a single shared address space which is partitioned, and variables are associated with the physical threads. UPC is an implementation of a Single Program Multiple Data (SPMD) computation model, but the parallelism is statically distributed at the beginning of the execution.

The above examples, are some of the most common parallel programming tools, but the availability of parallel systems has increased the number of programming libraries, APIs, languages and models. For example, Sequoia is a parallel programming language that exposes the memory hierarchy to the programmer. This can also help creating very efficient programs, but it requires the programmer to divide the application and to map it to the underlying hardware.

The use of the Cell/B.E. [16] and GPUs for scientific applications has motivated the creation of programming models for heterogeneous architectures. For example, NVIDIA created the CUDA platform to use their GPUs as GPGPUs. CUDA extends C for NVIDIA's GPUs; it takes care of many of the low-level details of the architecture. However, CUDA still requires programmers to divide the application to map the underlying hardware which sometimes is not trivial.

In the case of the Cell/B.E., its Cell SDK contains a C library that requires programmers to manage almost every aspect of the underlying hardware: thread creation and management, data transfer between processors, synchronizations, management of private memories, etc. However, as in the case of GPUs, there are several higher level libraries and programming models that use the low-level libraries and present users a simplified interface. CellSs [4] is one of these proposals. In CellSs, programmers divide applications into tasks and the runtime takes care of task dependencies, data transfers, and task scheduling to available processors.

OmpSs combines ideas from CellSs, SMPSs, GPUSs, ClusterSs (which we refer as StarSs) and OpenMP; it allows users to exploit both loop parallelism and task parallelism using the compiler and runtime; the result is that it takes care of the hardware details and managing parallelism.

2.2 OmpSs

OmpSs is a task-based programming model for heterogeneous multiprocessors systems. It extends OpenMP 3.0 with all the features of the StarSs data-flow programming model. StarSs is the generic name for the different architecture-dependent data-flow programming models developed at the Barcelona Supercomputing Center (BSC). They include SMPSs (for homogeneous shared-memory architectures), CellSs (for the Cell/B.E.), GPUSs (for Nvidia GPUs), and ClusterSs (for distributed memory clusters).

2.2.1 Irregular task dependence graph applications

An important feature of task-based programming models is their ability to extract parallelism from applications with a complex (or irregular) dependence graph between tasks.

Cholesky decomposition is an operation over a symmetric, positive-definite matrix. It converts a matrix into the product of a lower triangular matrix and its conjugate transpose.

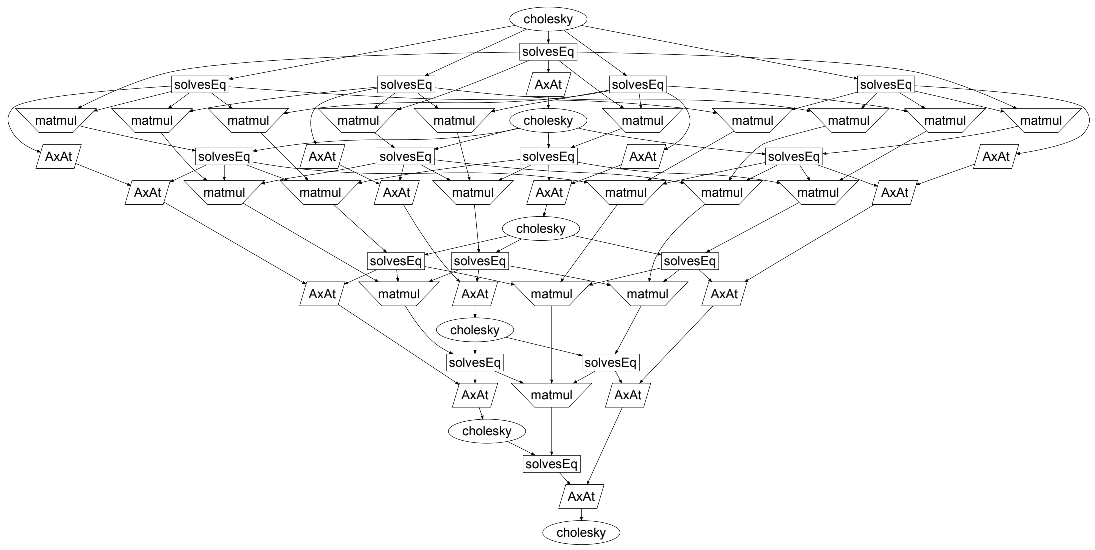


Figure 2.1: Inter-task dependence graph of a small 6x6 Cholesky decomposition.

Figure 2.1 shows the very irregular inter-task dependence pattern of a block-partitioned form of a 6×6 block Cholesky decomposition. However, coding such dependence pattern in the OmpSs programming model is fairly simple, since the dependence graph is dynamically built by the runtime library, and the parallelism is dynamically detected and exploited.

The available parallelism in Cholesky depends on the maximum width of the graph, and diminishes when the algorithm progresses.

2.2.2 CellSs

CellSs is a programming model for the Cell/B.E. which allows users to write sequential programs with a single address space. Programmers have to divide the application in tasks that can be executed on the synergistic processing elements (SPEs), but most hardware features are hidden to the users. CellSs consists of a source-to-source compiler and a supporting runtime library. The compiler translates C code, with annotations of tasks' inputs and outputs, into a standard C code with calls to the supporting runtime library.

The runtime system manages data and task scheduling without any explicit programmer intervention. This is similar, in spirit, to out-of-order (OOO) processors that automatically detect data dependencies among multiple instructions, build the dynamic data-flow graph, and dispatch instructions to multiple functional units. In CellSs, the data flow graph is not bounded by the instruction window, the granularity of instructions is much larger, and it does not require in-order commit to support precise exceptions as in OOO.

Moreover, the use of a software runtime manager provides flexibility to the programming model, as it is shown with the OmpSs extensions, since the architecture can change without having to change the program. However, an evaluation of the required speed of the runtime system for the architecture is beyond the scope of this work, but, as shown in Etsion et al. [13] and Kumar et al. [23], hardware acceleration of the runtime manager's critical sections is desirable for certain applications and specially as the number of workers increase.

This class of programming models is very flexible because the runtime knows about data dependencies. The use of the task graph at run-time allows the scheduler to reduce the system's memory bandwidth by making locality-aware dispatch decisions, as well as hide memory latencies by planning data transfers ahead of time, automatic double buffering.

As will be shown in Chapters 5 and 6, the ability of the runtime system to overlap data transfers with computation (double-buffering) using the DMA controllers, allows applications in Castell to tolerate very high memory latencies (up to thousands of CPU cycles), which permits designing a memory hierarchy that prioritizes bandwidth over latency.

2.3 Conclusions

Heterogeneous multiprocessor systems require new parallel programming models. OmpSs can handle the increasing heterogeneity of these systems, as it presents programmers with a sequential view of the programs and let the runtime handle the parallelism and hardware features that are critical for performance.

Software should opportunely use hardware features like DMA engines, synchronization mechanisms, or cache size for efficient execution. By making the runtime take care of these features, OmpSs has to be tuned for each architecture, but relieves programmers and compilers of these performance-critical operations. Moreover, it also can manage optimization decisions like double buffering—which are critical for performance in some systems. This way the programmer can concentrate on dividing the code in tasks that can be parallel and keeping the code correct.

Chapter 3
Castell

On-chip parallel computation is currently the preferred solution to increase microprocessors raw processing performance, within a given power budget [33]. However, chip multiprocessors (CMPs) struggle with programmability and scalability issues such as cache coherency, off-chip memory bandwidth and latency.

Heterogeneous multiprocessor systems as GPGPUs and the Cell/B.E. can obtain close to peak performance in parallel applications, and very high performance per Watt compared to SMP systems, due to the efficient use of their special purpose accelerators. Castell takes the best from these systems and the best from the symmetric multiprocessor (SMP) systems—their programmability—to create an architecture that it is scalable and easy to program.

3.1 State of the Art

We can classify multiprocessor systems from the point of view of the similitude of their components in two types: homogeneous and heterogeneous. All processors architecture is the same in homogeneous systems. While processors, in heterogeneous systems, can have different ISA, different memory hierarchy, different processing units, etc.

In general, as Figure 3.1 shows, in homogeneous architectures or symmetric multiprocessors (SMP) there are several processors (all the same) connected together to the last-level cache through an interconnection network.

Most chip manufacturers offer, at least, one SMP system, below we show their main characteristics:

- Intel's Sandy Bridge (Figure 3.1(a)) contains up to 8 superscalar cores. Each core is dual threaded and has a private L1 and L2 Cache, while the L3 is shared. The cores are connected with a 4 ring interconnect: request, snoop, acknowledge and a 32B wide data ring. The caches are snooped coherent. The L3 is inclusive (valid bits for the L2's), distributed and partitioned with one slice of the L3 cache for each core. The address mapping uses a single hash function, but all cores can access the entire L3.

- AMD's Bulldozer (Figure 3.1(b)) contains up to 8 superscalar cores, organizes in pairs, called modules. Each module contains 2 cores that share the L2, a large floating point unit and the SIMD, but each contains its own private L1. The L3 is shared by all modules. The module as an entity is not visible by the OS. The L1's are write-through.

- IBM's POWER 7 (Figure 3.1(c)) has 8 cores. The processors are 4-threaded simultaneous multithreaded (SMT). Private L1 and L2 caches per core while the the L3 is shared. The L3 management moves data to the closest region to a core automatically to reduce latency through different mechanisms: data can be cloned in several regions for shared data, and data can also be protected, mark as private to a core.
- Oracles's SPARC T3 (Figure 3.1(d)) contain 16 cores. The cores support 8 physical threads. The L1 cache is private per core, while the L2 cache is distributed (fine-grain interleaved) and banked.
- Tilera's Tile64 (Figure 3.2) contains 64 (simple 32bit RISC VLIW) cores. Figure 3.2 shows the 2D mesh of tiles architecture. Private L1 cache and L2 cache, but the collections of L2s create a logical distributed L3 cache. Each Tile also contain a 2D-DMA engine that can be used for transfers between memory-cache, cache-memory or cache-cache.

All of these processors contain one or more memory controllers, and they have a single address space.

On the other hand, there exist many heterogeneous multiprocessors systems; for example, the systems-on-a-chip (SoC) or embedded systems used for applications like hand-held devices. Given that they are battery-powered, they have to be very efficient. Most of them include a simple general purpose processor and a set of accelerators for image or audio processing, data compression and communication accelerators. However, they are not used, at least not now, for high-performance application. The exceptions are NVIDIA's GPUs and the Cell/B.E. designed with the same philosophy as the others, but their high performance encouraged scientists to used them for parallel applications.

Figure 3.3 depicts an architecture view of the Sony-Toshiba-IBM Cell/B.E. [16]. It was the first commercial product to include a general purpose superscalar processor and highly programmable accelerators on a single chip. It contains a general-purpose processor (PPE), and 8 vector processors (SPEs). While the PPE has a regular memory hierarchy with coherent L1 and L2 caches, the SPEs' only see their non-coherent private memories (or LS)—storing both instructions and data. The SPEs use a DMA controller (MFC) to access the memory system. All processors, SPEs and PPE, are connected through a single 4-ring interconnect (EIB) to the memory controller (MIC).

GPUs are designed for graphic acceleration with different vendors designs being very similar. They contain tens of simple processors that work with the same program counter, with limited data sharing between different processing threads, and some special functional units (shared by several processing units) for texture computation. GPU systems require a general-purpose processor to run the OS and schedule work into the GPUs. The GPUs and the general-purpose processor have different address spaces, with a DMA engine on the GPU that can asynchronously transfer data between the two address spaces.

The most interesting GPU from the point of view of general-purpose processing is NVIDIA's Tesla (Figure 3.4), which can be programmed using CUDA. Programs are divided into a very large number of threads, which are split in blocks that run the same kernel. Within a block, threads can communicate using shared memory, but threads between blocks can only communicate through global memory and atomic operations. Furthermore, the L2 is very small in relation to the number of processors that can run simultaneously, so its objective is to reduce latency for shared data when used close in time.

Beside the SMP version of Intel's Sandy Bridge, there is a version with a GPU integrated. In this version, the L3 ways are divided in coherent and no coherent, for the CPUs and GPUs respectively. The coherency of the GPU data must be enforced by software, and data can be shared between different domains with synchronization instructions. This is similar to NVIDIA's Tegra 2 and AMD Fusion processors. Tegra 2 combines a dual core ARM Cortex-A9 with a GPU, and a set of accelerators for video, audio, imagine and display. AMD Fusion processors combines AMD general-purpose processors with ATI's GPUs.

We summarize the characteristics of homogeneous and heterogeneous systems in Table 3.1.

Homogeneous	**Heterogeneous**
Processors all equal	Different types of processors
Single address space	Multiple address spaces
Coherent cache hierarchy	Private or no cache hierarchy
Low performance efficiency	High performance efficiency
Easy to program	Hard to program

Table 3.1: Homogeneous vs. heterogeneous systems for parallel applications.

Because computer architecture is a very dynamic research and commercial area, many things have changed since the beginning of this thesis 5 years ago:

- Only a few general-purpose processors contained the memory controller inside the chip. IBM's POWER 4 and Alpha's 21364 did have it on-chip.
- Intel and AMD processors were mainly uni-core processors (even though they had already presented their first dual core processors), they still relied on the north bridge for memory controller, and there was not any proposal to include a GPU inside their general purpose processors.
- GPUs were not used for general-purpose processing and CUDA had not been released.

From academia, Castell is very similar to Rigel [22] in that both architectures have a hierarchical organization and are designed to be programmed in a task-based programming model. The main differences are: the use of local memories by Castell and only regular cache hierarchy on Rigel, and the idea of single program multiple data (SPMD) model of Rigel while Castell does not make this type of restriction on the programming model.

3.2 The Castell Architecture

Castell is a heterogeneous architecture framework designed for a master-worker execution model. The design relies on performance features that task-based programming models can provide automatically. Then, Castell scalability is a consequence of the architecture design and the software using the hardware features. The programmability is left to OmpSs.

Castell consists on a combination of different processors, caches, memory controllers and a synchronization unit, connected through a hierarchical interconnection network, on a single chip. We designed Castell to be managed at runtime in a master-worker mode. Figure 3.5 shows a logical view of Castell. The (M)aster processors run the OS, the application `main()` routine and the runtime system. They generate tasks to be off-loaded to the specialized (W)orker processors, as indicated by a software runtime scheduler. The number of Masters and the configuration of clusters is implementation dependent.

Processors have direct load/store access to all memory locations: all scratchpad memories and the off-chip memory. In addition, processors access off-chip memory through a distributed LLC. The Workers contain a DMA controller for asynchronous data transfer between scratchpad memories and main memory that allows overlapping of data transfer and computation.

Key to Castell is the runtime management software, which detects and exploits task-level parallelism across multiple Workers much in the same way as an out-of-order superscalar processor dynamically detects instruction-level parallelism to exploit multiple functional units. This hides most of the architecture complexity, including processor diversity, DMA engines, and manual management of the scratchpad memories. The runtime ability to schedule data transfers ahead of time allows applications to tolerate long memory latencies, which led us to focus the architecture design on providing sufficient bandwidth to feed data to all Workers.

3.2.1 Master processors

One of the Master processors starts applications at the `main()` subroutine of the program. From there, the application can spawn multiple parallel threads that the runtime allocates to other Master processors. As these processors main functionality is to spawn tasks for Workers sequentially, their single thread performance is critical to the system as a whole. Therefore, they consist of a high-performance out-of-order design [40].

Because Masters execute the software whose data access pattern is unknown at run-time—the runtime and `main()` are not annotated—, they only access memory through the cache hierarchy. Masters contain coherent instruction and data L1 caches with a coherency protocol to exploit locality. This allows runtime data to be close to the Masters, which combined with their OOO execution, gives them the required speed for task generation and management.

3.2.2 Worker processors

In addition to the regular cache hierarchy provided to the Masters, Workers also feature a local scratchpad memory (LM). The local memories are mapped at run-time into the application's logical address space and are accessed through regular load/store instructions. This means that a memory access from a Worker has to go through the TLB in the memory management unit (MMU) in order to be steered towards the Worker local memory, another local memory, or through the cache hierarchy.

In order to avoid the latency penalty involved in sequential access to both the TLB and the local memory or the L1 cache, Workers first check a logically indexed and tagged write-through

L0 cache that behaves like a vector cache, allowing unaligned load and store operations [38]. Unaligned L0 accesses can cause two cache misses that two properly aligned L1 or local memory accesses resolve. Since both L1s and local memories only service L0 misses, they do not have to support unaligned accesses, improving their efficiency.

To avoid coherency problems, between local memories and the cache hierarchy, local memories are non-coherent, and therefore, L1 caches only capture addresses in the DRAM physical range. That is, memory address accesses to any scratchpad memory, are not cached. Further details are described below in Section 3.3.

In addition, each worker features a DMA controller to overlap data transfer and computation. The DMA controller is capable of copying data between the local memory and the off-chip memory or the other local memories.

3.2.3 Shared Last Level Cache

All off-chip memory traffic goes through a distributed (or banked) shared last level cache (LLC) that captures both misses from the L1 caches, as well as DMAs transfers from/to off-chip memory.

The distributed structure of the LLC eliminates the need to maintain coherency across its blocks, since each datum is mapped to one block based on its physical address. In addition, it enables the use of fine-grain interleaving to increase cache bandwidth on acceses to consecutive addresses. Since programs use DMAs to transfer data between local memories and main memory, the LLC typically encounters coordinated accesses to multiple cache lines. Fine grain interleaving enables the cache to serve multiple parts of a single DMA request in parallel increasing the effective bandwidth observed by requester.

The distributed nature of the cache leads to a nonuniform cache access time. However, the architecture handles long (and variable) latencies without any impact on performance. Thanks to the runtime management of data transfers, applications can exploit the size and bandwidth benefits of the distributed cache without suffering any of the latency penalties.

Since the local memory addresses are not cached on the L1, the LLC only needs to maintain coherency with the Masters' L1 caches. Such coherency engine is simplified because:

- The shared LLC is inclusive.
- The LLC contains the directory state which only needs an entry line.
- The directory only keeps per-cluster presence bits, not per-L1—invalidations are broadcasted inside each involved cluster.

Chapter 6 explores the design details and performance obtained with the LLC.

3.2.4 Memory Controllers

The memory controllers (MCs) connect the chip to the DRAM modules. Each MC supports several DRAM channels with request queue and FIFO scheduler per-channel. Therefore, requests to a given channel are handled in order, but they can execute out of order with respect to requests sent to other channels.

Similar to the shared cache design, the global address space is fine-grained interleaved across the different MCs. Given the bulk nature of memory accesses caused by the common use of DMA transfers, such a fine-grained interleaving provides better memory bandwidth than coarse-grained, as it parallelizes a typical DMA transfer both across MCs and channels. Further details will be presented on Chapter 5.

Furthermore, because the memory controllers interleave requests from many Workers, it is unlikely that a page buffer from a DRAM-bank will be reused for two consecutive requests. For this reason, we employ a closed-page DRAM policy [21].

3.2.5 Synchronization Module

Parallel applications are usually very sensitive to synchronization latency and, therefore, hardware mechanisms are critical for CMPs; Castell is not an exception, as it is shown in Chapter 7. For an architecture with hundreds of cores, the accesses to shared resources can become a bottleneck if the synchronization mechanism is slow. Castell uses a module that implements semaphore synchronization in hardware, which are very slow in software since they require OS intervention. The proposed module allows applications and the runtime system to rely on a well-known synchronization mechanism.

3.2.6 Network On Chip

In order to connect hundreds of on-chip components, Castell uses a hierarchical K-bus organization. A K-bus is a collection of buses in which a node that wants to transmit something through the network requests permission to the K-bus arbitrator. If there is no previous request for communication with the same destination port, the node is dynamically assigned to one of the buses. A 2-ring bus there can have up to 2 simultaneous data transfers in a given cycle, as long as there are no transfers to the same destination port.

As shown in Figure 3.5, we have organized Castell's Workers in clusters of 8 processors plus a Master (when needed). Each one of the clusters uses a 2-ring bus for its intra-cluster network. Each cluster has a single (full-duplex) port connecting it to the global interconnect. Further details will be presented on Chapter 8.

3.3 Memory Spaces

Architectures with local memories are commonly considered hard to program given that they require managing the different address spaces manually. Castell only has a single logical address space but multiple physical spaces. This simplifies programmability, as in Villavieja et al. [55], we have created a new `malloc` instruction called `memalloc` to let applications map a physical memory range of a Workers' local memory to a logical range and `memcpy` translates to a DMA transfer, as shown in Figure 3.6. The new `memalloc(LM, SIZE)` allows the programmer to specify the target physical range; in Figure 3.6, `A` is a vector of size 128 that is allocated into the local memory 0 (LM0). Similarly, `B` is a vector created with a `memalloc` without target parameter so the physical address range of the DRAM is used, then the memcpy becomes a DMA transfer from DRAM to LM0.

When a processor modifies a variable located on a local memory, the cache hierarchy is not affected because these regions are not cached. This implies that, the cache coherency traffic is low if most of the work is performed on the local memories. Even though the execution should prioritize the use of local memories to reduce cache coherency—as software banking on SMPs—, last level caches would still contain DMA traffic. Chapter 6 shows that the last level cache filters and reduces off-chip traffic. The following example illustrates the use of Castell single logical address space.

Chapter 2 introduced OmpSs, but it was mentioned that the experiments on this thesis were performed using CellSs. However, Castell's Workers contain both local memory and L1 cache and a single logical address space, but there is no architecture with these characteristics. The Cell/B.E., with programs written with CellSs, was chosen to collect the traces for the simulations and, therefore, the experiments are limited to programs that only use the local memory of the Workers. These results are valid for Castell, with both L1 and local memories, because the L1s are there to host shared variables, and their main objective is to extend the programmability of the architecture and provide backward compatibility. Note that applications can be written to use only local memories, but it is a hard limitation for the programmer or programming model.

Let's suppose we want to compute the equation

$$E = f(A, B0) + g(A, B1) + h(A, B2) \tag{3.1}$$

where A is a large vector (size M) that does not fit on the LLC, while $B0$, $B1$, $B2$, and E are vectors (size N) and all together fit on the L1 of any processor.

A parallel version of equation 3.1 can be written in OmpSs as Program 1 shows. The construct `#pragma omp task` includes `device(CastellW)` indicating that the code is to be executed on a Castell Worker. Tasks' inputs and outputs are used by the runtime to take advantage of the local memories. The `memalloc` allows the runtime to allocate space on the Workers local memories—renaming the output variables and making DMA transfers. Then, the local variables of the function `compute` do not point to main memory locations, but to local memory locations.

The execution of Program 1 is described in Figures 3.7 and 3.8 (which depicts a Castell implementation with one Master and 3 Workers). This example shows that the runtime can allocate and transfer data to the Workers; so programmers do not have to worry about managing the local memories, writing programs for a determined number of processors and size of local memories. The runtime allocates tasks variables in the local memoies while they fit on them.

The pseudocode on Figure 3.7 is an example of how the equation is executed on Castell, given the OmpSs code just described. Vectors $B0$, $B1$, $B2$, are packed on B. Figure 3.8 shows a snapshot of the state of the local memories and caches of the workers during computation. Each Worker has a local copy of the required vector (`b0`, `b1`, and `b2`) and a temporal buffer for their part of E (`e0`, `e1`, or `e2`), but they access the shared vector A (`A`) since it does not fit on their L1 cache. By allocating the small vectors on the local memories, and copying data to them, the vectors would be captured by the LLC, but local modifications would not generate coherency traffic. Similarly, shared vector `A` is present on the L1s and LLC, but it does not compete for space on the L1s with local variables. This way, the programmer knows that the variables of interest are close to the processing units, and they do not enter in conflict with the

Program 1 OmpSs skeleton code to compute Equation 3.1.

```
#pragma omp task device(CastellW) \
                 input(A) input(b0) output(e)
void f(float A[M], float b0[N], float e[N]) { }

#pragma omp task device(CastellW) \
                 input(A) input(b1) output(e)
void g(float A[M], float b1[N], float e[N]) { }

#pragma omp task device(CastellW) \
                 input(A) input(b2) output(e)
void h(float A[M], float b2[N], float e[N]) { }

#pragma omp task input(E0) input(E1) input(E2) output(E)
void sum(float E0[N], float E1[N], float E2[N], float E[N]) { }

compute (float* A, float* B, float* E)
{
  float E0[N],E1[N],E2[N]; //temp buffers
  f(A,&B[0],E0)
  g(A,&B[N],E1)
  h(A,&B[2*N],E2)
  sum(E0,E1,E2,E)
}
```

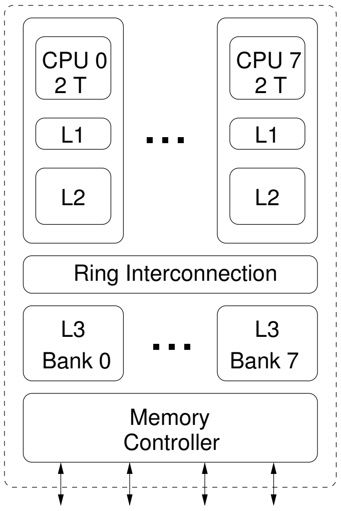
shared variables.

3.4 Conclusions

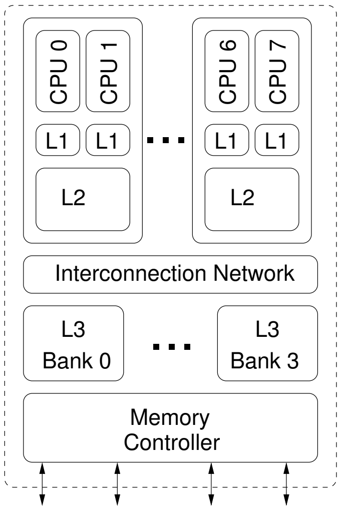
Castell uses a number of domain-specialized accelerators as Workers, to run the bulk of the application, and a few high-performance processors, as masters, to run the operating system and orchestrate the application. It contains a module for fast synchronization and a hierarchical interconnection network.

These features, combined with multiple physical, and single logical, address spaces that give users a unified memory view can be easily programmed with OmpSs. This ensures that OpenMP programs can be run on Castell and allows programmers to construct, from a familiar base, their programs.

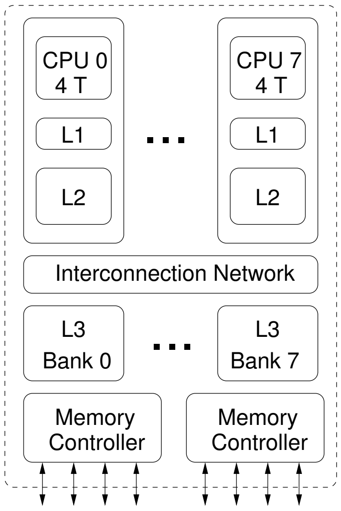
Since the introduction of the microprocessor, general-purpose processors have dominated the market, motivated by their programmability and the constant performance increase (based primarily on frequency and the ILP). However, this increase has reached its limit, and chip multiprocessors are starting to dominate the market. Because special-purpose processors can be faster and more efficient than general-purpose processors, the increase in transistors is making multicore systems heterogeneous, to provide balance between performance and programmability. This is motivating microprocessors with different types of processors even in the personal computer market. Castell is a proposal in this direction.



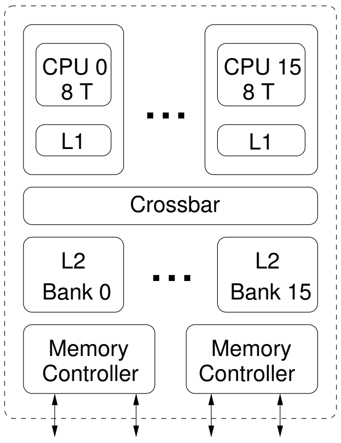
(a) Intel's 8 Core Sandy Bridge.



(b) AMD's Bulldozer.



(c) IBM's POWER 7.



(d) Oracle's SPARC 3.

Figure 3.1: SMT architectures. CPU is a general purpose processing unit. For all architectures the L1, L2 and L3 are the cache levels.

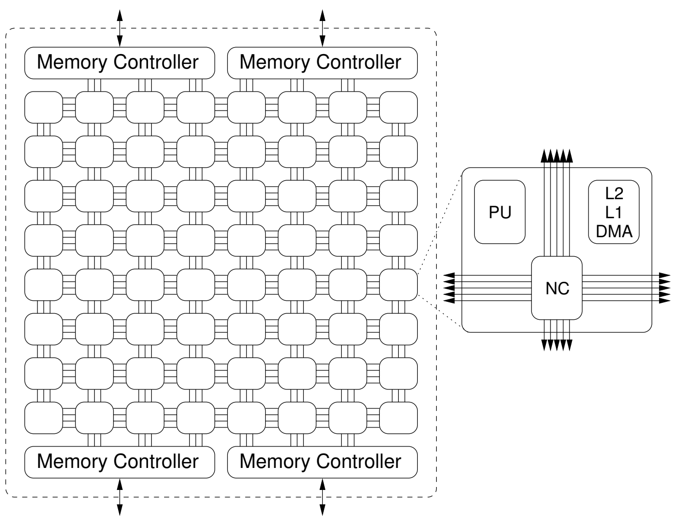


Figure 3.2: Tilera's Tile 64 microprocessor.

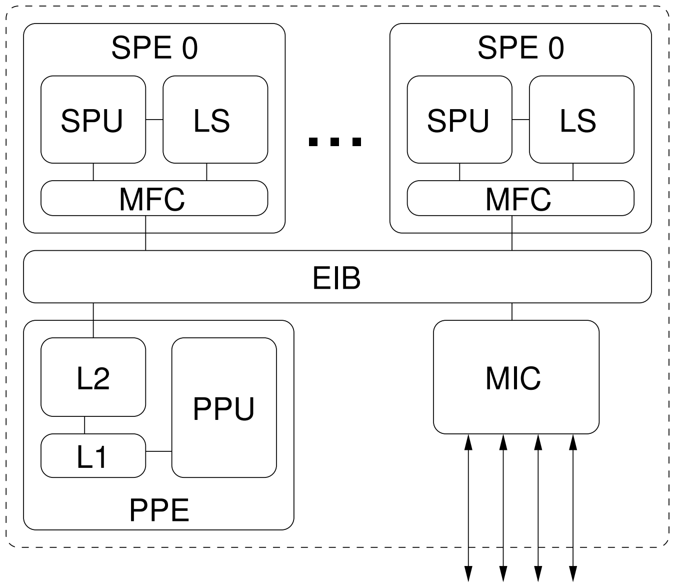


Figure 3.3: Cell/B.E. from Sony, Toshiba and IBM. SPE is synergistic processing element, and SPU synergistic processing unit; LS is local store; MFC is the memory flow controller, it contains the DMA controller; EIB is the element interconnect bus; PPE is the Power processing element, and the PPU is the power processing unit; MIC is the Memory interface controller.

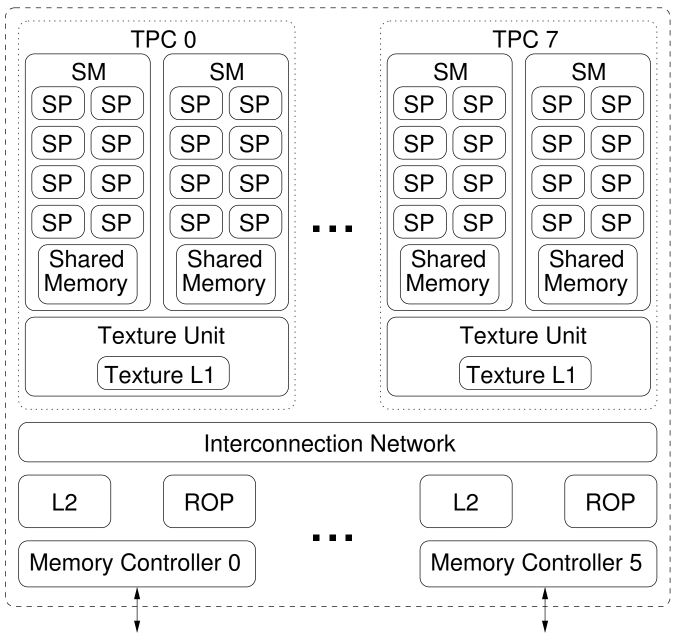


Figure 3.4: Nvidia's Tesla GPU architecture. SP are scalar processors, SM are streaming multiprocessor, ROP is raster operation processor, and TPC is texture processor cluster.

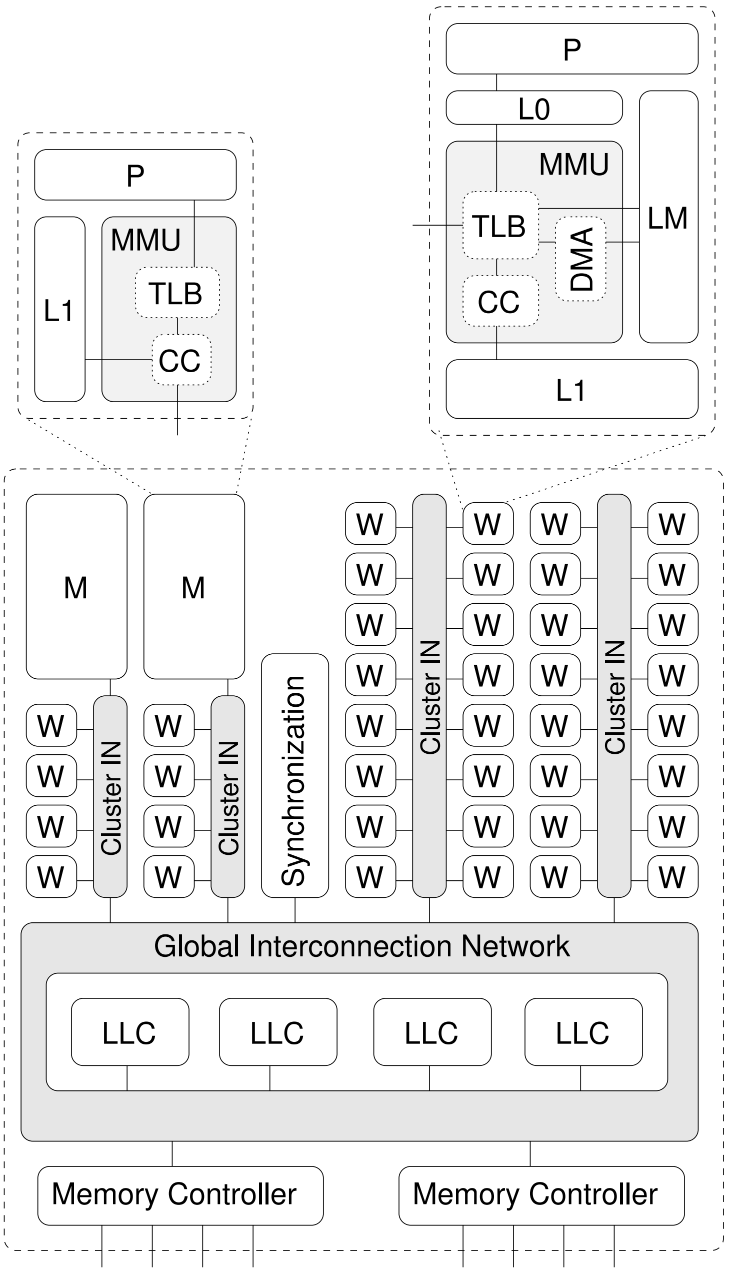
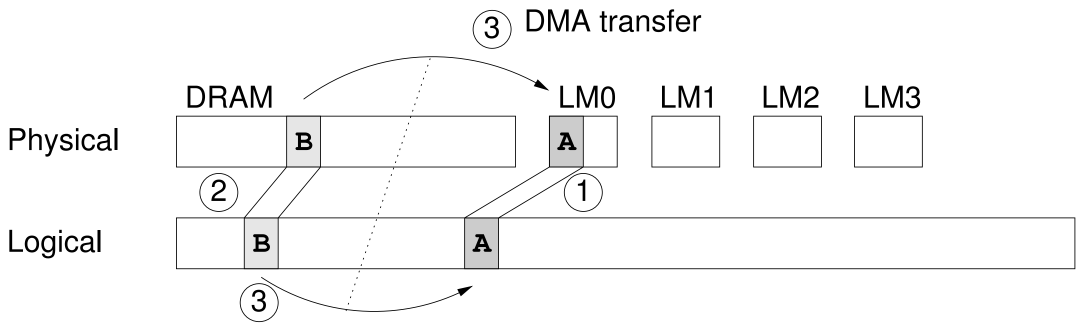
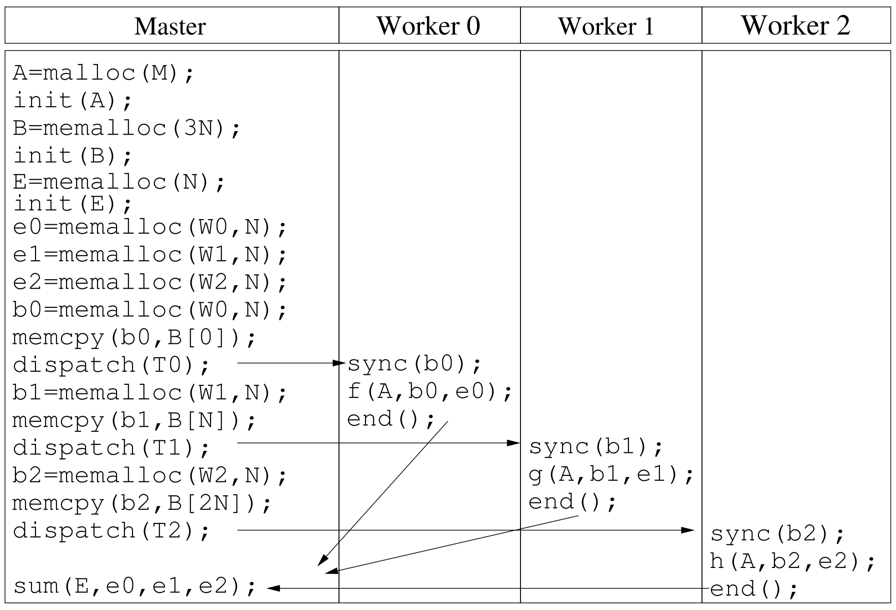


Figure 3.5: a 40-Worker Castell example. In this figure, M stands for Master, W is Worker, IN is Interconnection Network, P is processing unit, CC is cache controller, L0, L1 and LLC are cache level 0, 1 and last; MMU is Memory Management Unit, LM is Local Memory, DMA is a direct memory access unit, and TLB is the Translation Lookaside Buffer.



```
1. A = memalloc(LM0,128);
2. B = memalloc(128);
3. memcpy(A,B,128);  //Copies content of B into A
```

Figure 3.6: Castell's multiple physical and single logical memory spaces. The API is modified to allow software to indicate a physical memory target for allocation of the malloc. The copy between memory locations when there is more than one physical spaces is converted to a DMA transfer.



Master	Worker 0	Worker 1	Worker 2
`A=malloc(M);`			
`init(A);`			
`B=memalloc(3N);`			
`init(B);`			
`E=memalloc(N);`			
`init(E);`			
`e0=memalloc(W0,N);`			
`e1=memalloc(W1,N);`			
`e2=memalloc(W2,N);`			
`b0=memalloc(W0,N);`			
`memcpy(b0,B[0]);`			
`dispatch(T0);`	`sync(b0);`		
`b1=memalloc(W1,N);`	`f(A,b0,e0);`		
`memcpy(b1,B[N]);`	`end();`		
`dispatch(T1);`		`sync(b1);`	
`b2=memalloc(W2,N);`		`g(A,b1,e1);`	
`memcpy(b2,B[2N]);`		`end();`	
`dispatch(T2);`			`sync(b2);`
			`h(A,b2,e2);`
`sum(E,e0,e1,e2);`			`end();`

Figure 3.7: Execution example of Equation 3.1 with the use of `memalloc` and `memcpy` on Castell. `memcpy` are DMA transfers and they are asynchronous, so Workers require a `sync` instruction before they start computing with the data being transfered. The tasks are `T0`, `T1`,`T2`.

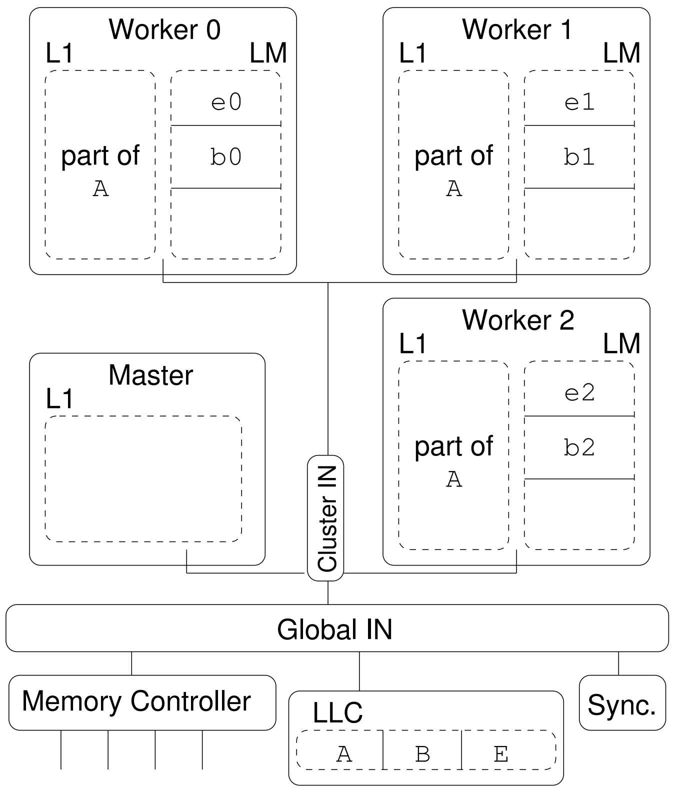


Figure 3.8: State of the local memories and caches during execution of Workers' tasks of the code on Figure.

Chapter 4
Experimental Methodology

There are two elements that are almost always present in computer architecture research: a simulation infrastructure and a set of applications. Most computer architecture research requires a simulation infrastructure to run experiments because analytical modeling has not been successful predicting processors performance. This is mainly because computer programs cannot be modeled as stochastic processes. So researchers have to use simulators and real programs to test their designs. These two elements are key to research and the success of an architecture. In this Chapter, we describe the set of applications and the multiprocessor simulator (TaskSim) used in this work.

Simulators are very useful to analyze and to improve the performance of software running on well known architectures; but they are also used to predict the performance of new architectures or new features of known ones. However, as we pointed out in the Introduction, single-thread performance is no longer improving, so even though systems are getting larger and faster, simulators are not getting any faster. The problem is that simulation speed determines the size of the architectures and the level of detail at which they can be tested in a reasonable time. Therefore, existing cycle-accurate simulation infrastructures of microarchitectures can not simulate large-scale systems.

At the same time, the design of a chip multi-core system requires a detailed evaluation of the memory system, the cache contention and the interconnection network. Otherwise, simulation of downsized versions will lead to incorrect scalability assumptions. TaskSim allows us to simulate hundreds of processors in a reasonable time modeling in detail critical scalability features.

4.1 Tasksim

TaskSim is a scalable event-driven simulator targeting large-scale accelerator-based architectures. For the experiments of this thesis, I have helped developing several modules; in particular, I have worked on the CPU, the cache and the DRAMs; I also helped making all modules work together and in the development of engine. The key for its scalability is the use of a task-level abstraction for accelerators simulation. Therefore, since our evaluations do not require accelerators microarchitectural modeling, we obtain cycle-accurate simulations only with the detailed simulation of data transfers and inter-accelerator synchronization on the shared resources in the architecture (caches, memory and interconnection). Because accelerators have task data loaded in their local memories and compute only locally, external events do not interfere with task execution (task execution isolation); therefore, no detailed timing of task execution is required, only task duration from the original execution.

On top of the task-level abstraction, TaskSim is built around an event-driven simulation framework that avoids the simulation of inactive hardware components and idle time. This is accomplished by skipping *empty* cycles (cycles with no activity) and selectively executing only the hardware components with scheduled activity or receiving external requests in a given cycle. This allows TaskSim to simulate hundreds of accelerators in minutes without loss of accuracy for macro-architecture scalability studies.

These characteristics allow TaskSim to target the simulation of large parallel applications coded in a master-worker task-offload computational model. As an example, Figure 4.1 shows a graphical representation of a chunk of a CellSs [4] application execution on a Cell/B.E. processor. Each one of the horizontal bars represents the state of an application thread along time. The different gray levels represent the computational phase types. For example, light gray on the MASTER (a thread on the PPE) is task generation, while light gray on the HELPER (another thread of the PPE) represents task scheduling. Light gray on the SPEs represents task execution. Darker grays levels in each processor type represents other execution phases. In addition, the trace contains information about the inter-task dependencies, shown as black lines between different threads. Tasksim uses all this information to schedule tasks satisfying their dependencies.

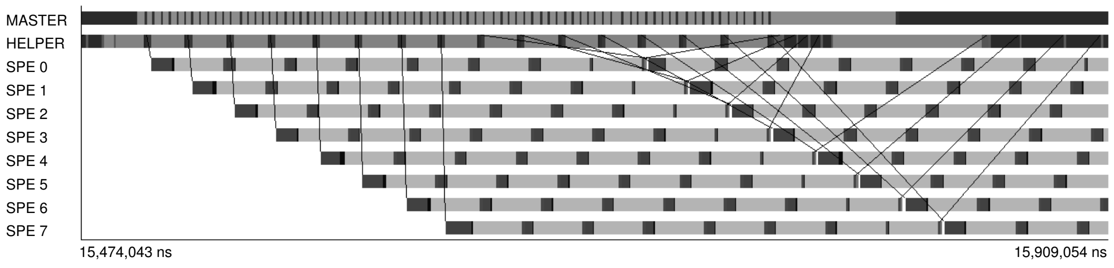
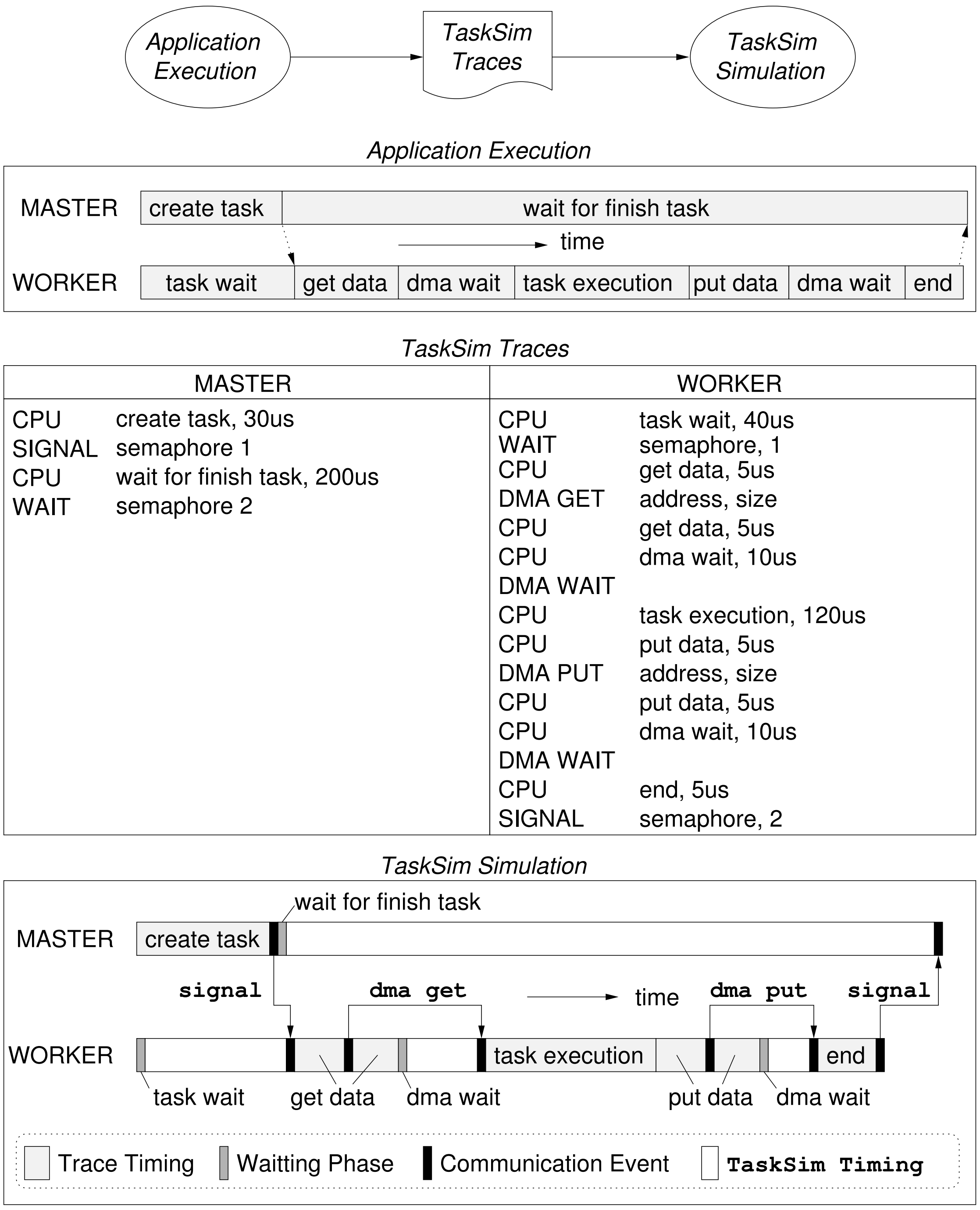


Figure 4.1: Graphical representation of a CellSs application trace from a Cell/B.E. execution. The different gray levels represent different computation phases of the processors and the black lines between processors represent communications and synchronizations.

As previously mentioned, computational phases, such as task execution, are not simulated in detail. Instead, they are simulated as a single instruction with duration obtained from the trace file. However, TaskSim ignores the duration of CPU phases involving access to shared resources in the architecture, such as waiting for DMA transfers, and their duration is given by the behaviour of DMA transfer during simulation. This is obtained by a detailed simulation of DMA controllers, caches, interconnection, memory controllers, and DRAM DIMMs.

The information contained on traces allows the simulator to group all tasks on a single task list, and dynamically schedule them to the processors. The dependence information is required to verify the execution correctness, so that no task is scheduled before all its dependencies are satisfied—even though the scheduling order may differ from that of the original trace because of the increased number of worker processors.



MASTER		WORKER	
CPU	create task, 30us	CPU	task wait, 40us
SIGNAL	semaphore 1	WAIT	semaphore, 1
CPU	wait for finish task, 200us	CPU	get data, 5us
WAIT	semaphore 2	DMA GET	address, size
		CPU	get data, 5us
		CPU	dma wait, 10us
		DMA WAIT	
		CPU	task execution, 120us
		CPU	put data, 5us
		DMA PUT	address, size
		CPU	put data, 5us
		CPU	dma wait, 10us
		DMA WAIT	
		CPU	end, 5us
		SIGNAL	semaphore, 2

Figure 4.2: TaskSim traces and simulation example.

As can be seen in Figure 4.2, *computational phases* are execution periods happening between two communication events. Communication events can be either synchronization events (such as semaphores) or data transfers (DMA get, put, and DMA wait). Note that the name of the phase can change between communications as *task execution* becomes *put data*. However, if a Worker performs a DMA transfer in the middle of a task, we consider two separate computation phases (as in *get data* and *put data*). TaskSim makes zero all original waiting phases timings—task wait, dma wait for finish task—and simulates data transfer, and synchronization latencies reproducing the execution of the application.

The computation phase abstraction requires to get the execution duration values from different sources. We consider three different sources where they can be obtained from:

1 Native execution on a real system where the application is instrumented to dump computation durations to a trace file.

2 An off-line simulation of the accelerator microarchitecture—computation duration can be obtained from a separate cycle-accurate model of the accelerator pipeline with simulation of just one processor, not all of them.

3 An analytical model which allows for quick approximations based on frequency scaling, superscalar issue width, etc.

All three methods, described above, need to be combined with an application-level trace describing the inter-task synchronization and DMA transfers, so that TaskSim can rebuild the execution on the projected muticore, and model the memory and interconnection systems in cycle-accurate detail.

The biggest benefit of abstract CPU simulation is that most of the simulated time does not consume any computational resources on the host machine. This is a consequence of skipping most of the simulated cycles which correspond to computational phases of accelerators.

For example, simulating a 4096×4096 matrix multiplication of a Cell/B.E.-like architecture incurs only a 400x slowdown compared to native execution. Moreover, simulating a target platform with 256 accelerators, 32 cache banks, 4 MICs, and 8 DRAMs (32x more processors than the Cell B.E.), is only 1.5x slower. As a result, simulation time is almost independent of the target architecture size.

Simulator validation

We configure TaskSim to model a single-chip Cell/B.E. system matching as close as possible its architecture, paying careful attention to the interconnection and memory parameters.

Figure 4.3 compares real execution time, on the Cell/B.E., with simulations for a set of applications. Details of the applications can be found in the following section.

The results, which are normalized to the real execution time, show an accurate simulation for most benchmarks. As mentioned before, the CPU abstraction does not introduce errors; on the contrary, it eliminates a big source of simulation error due to microarchitecture behavior simulation.

However, MatMul and Cholesky show significant differences for 1x simulation (no modification of the original phase durations is performed). The problem with these benchmarks is that

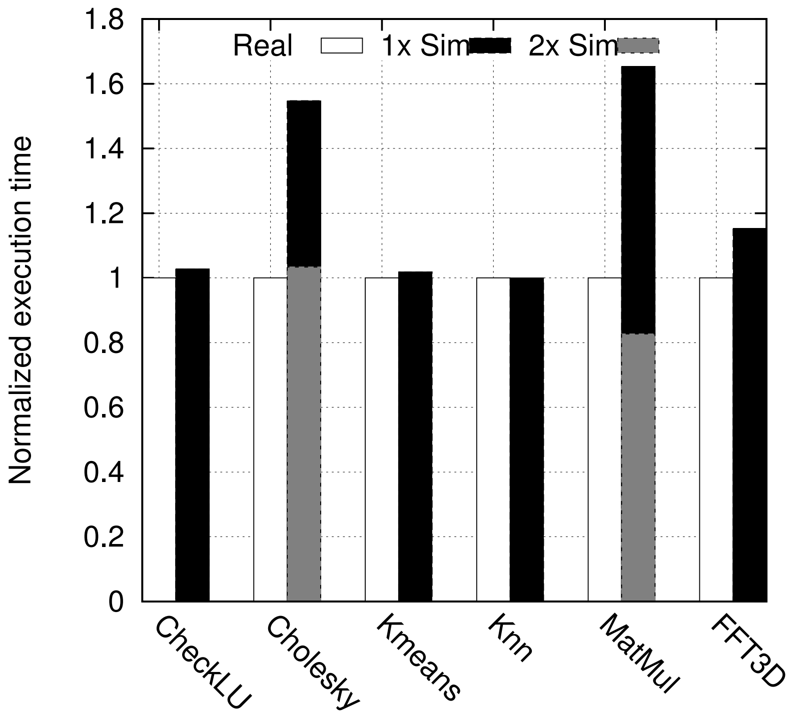


Figure 4.3: Comparison of real Cell/B.E. execution time and TaskSim simulations. 1x means that no change has been performed on the traces, while 2x means that computation phases have being speed up by this factor.

they are dominated by master thread speed. Therefore, a significant part of the prediction error is due to the execution overhead introduced by the instrumentation tracing mechanism—altering the real execution time.

In experiments where a comparison to a real execution is necessary (for a single application), we could instruct the simulator to apply a correction factor to some phases. In particular, we show simulation results for MatMul and Cholesky with a master thread twice as fast at the trace. It shows that the difference to the original is significantly reduced.

However, the important point from these results is one of the limitations of the Cell/B.E. architecture–the PPE speed.

4.2 Benchmarks

The traces used in this thesis were generated on a Cell/B.E. running at 3.2GHz (IBM BladeCenter QS22). The traces contain all the Workers' DMA transfers, computational phases and synchronization events. The CPU burst times obtained on the executions are used as the baseline timings for both Masters and Workers. To reduce OS noise, each benchmark was executed multiple times, and we selected the trace from the fastest execution.

We have selected six high-performance parallel applications which have been optimized for the SPEs and executed with the CellSs runtime using double buffering (overlap DMA traffic with task execution) in order to maximize their performance.

1 **FFT3D**: Fast Fourier Transform of a three dimensional cube. The kernel transforms a 256x256x256 cube of complex numbers (2 floats), and performs (1) a FFT on each row (FFT1D), (2) a rotation of the cube, (3) a second FFT, (4) a second rotation, and (4) a third FFT.

2 **MatMul**: Blocked matrix multiplication of 4096×4096 float matrices.

3 **Cholesky**: Blocked Cholesky factorization of a 4096×4096 float matrix. The matrix is traversed by columns to perform the factorization.

4 **Knn**: k-Nearest Neighbors algorithm. A distance based object classification algorithm, featuring lazy learning. The problem size consists of 100000 samples, 16384 points to label, 48 dimensions, with 30 neighbors, and 20 classes.

5 **Kmean**: k-Means algorithm. A distance based data clustering algorithm that performs iterative refinements. The problem size is 256K points, 64 dimensions, 64 centers, with the threshold set to 0.01.

6 **CheckLU**: Blocked Sparse LU decomposition, followed by a matrix multiplication verifying that $A = L \times U$.

Table 4.1 summarize the main characteristics of the applications: number of tasks, average task runtime, memory footprint, and estimated bandwidth required per task. Given that not all

Kernel	**Est. BW Per task**	**No. of tasks**	**Avg. Task runtime(μs)**	**Problem size(MB)**	**Task Block Size**
FFT3D	3.27GB/s	32768	13.9	128	64 × 64 sub matrix Transposition, 256 line FFT
MatMul	1.42GB/s	262144	25.8	192	64 × 64 sub matrix
Cholesky	1.68GB/s	357760	28.0	512	64 × 64 sub matrix
Kmean	1.56GB/s	335872	30.7	195	8192 vector
Knn	0.49GB/s	800768	7.9	36	8192 vector
CheckLU	1.11GB/s	54814	45.7	256	64 × 64 sub matrix

Table 4.1: Benchmarks main characteristics.

tasks (on a benchmark) request the same amount of data, and have different durations, the average bandwidth was obtained dividing total application bytes transferred and total task execution time. The estimated bandwidth is an average for all tasks of each application.

4.3 Additional applications

Some experiments were carried out using two additional applications. The H.264 decoder, from the multimedia domain, developed by Mauricio Alvarez; and Smith Waterman, from bio-informatics, developed by Friman Sanchez. These applications were programmed for the Cell/B.E., but the versions used were not coded in CellSs.

4.3.1 H.264/AVC Decoder

The parallel implementation of H.264 decoding (3D-wave) exploits intra-frame as well as inter-frame Macro Block (MB)-level parallelism. Inter-frame dependencies have a limited spatial range, since motion vectors are typically small. Therefore, It is possible to start decoding the next frame before the decoding of the current frame has finished, as soon as the reference macroblock has been decoded, as shown in Figure 4.4. This strategy increases the amount of available parallelism significantly beyond what a 2D-wave implementation provides, without increasing the decoding latency of individual frames Azevedo et al. [2].

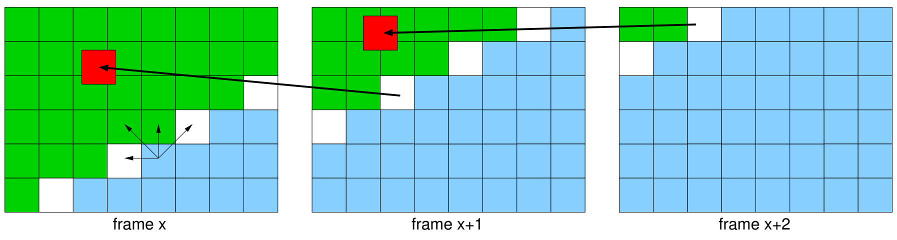


Figure 4.4: H.264-3D intra-frame wavefront extending multiple in-flight frames.

The amount of MB-level parallelism of the 3D-Wave is very large, ranging from 4000 to over 7000 MBs depending on the input movie and the number of in-flight frames. Furthermore, after the first frames, the amount of parallelism becomes stable.

4.3.2 FASTA Smith-Waterman

There are multiple levels of parallelism in the protein sequence alignment problem. The most frequently exploited is the embarrassingly parallel situation where a collection of query sequences has to be aligned to a database of candidate sequences [44]. The parallel strategy, to align one query sequence to a single candidate, is based on the FASTA Smith-Waterman code (ssearch) [35]. The most time consuming part of the algorithm computes the sum of the diagonals in a dynamically generated matrix, leading to a 2D wavefront parallelism.

The amount of parallelism depends on the size of the compared sequences. A pair of sequences of 4M and 30K symbols provides sufficient parallelism to keep 256 processors busy processing blocks of 16K elements. Longer sequences, like complete genomes, provide sufficient parallelism for even more cores.

4.4 Conclusions

The design of heterogeneous chip multiprocessors require a scalable simulation infrastructure that can execute to completion parallel applications, and that can model memory hierarchy in detail to understand data contention. For this thesis, we rely on TaskSim that provides us with these characteristics.

TaskSim is a cycle-accurately trace-driven simulator for heterogeneuous-multiprocessor architectures. The traces should describe the application as a combination of tasks, with their duration, and a series of synchronization information (including all memory transfers) for accurate modeling. It was shown that TaskSim can accurately model the execution of the Cell/B.E..

The set of applications described, and used in this thesis, represent some of the main features that will appear in scientific computing: dense and sparse matrix operations, frequency transformations, and, both, embarrassing parallel and complex task dependency problems.

Chapter 5

Off-Chip Memory

Memory bandwidth and latency have always been critical performance aspects of microprocessors, even for single processor architectures. The increased compute performance provided by smarter architectures, deeper pipelines, and higher clock frequencies has reduced computation time to the point where more and more applications have turned from compute-bound to memory-bound. The problem is that computing has become so fast, that data can not be read from memory fast enough to feed the processor's functional units.

Most computer systems today are built from commodity components, and that includes high-ranking supercomputers in the Top500 list like the Roadrunner at Los Alamos National Laboratory. This implies that whatever memory system is designed, it has to work with standard off-the-shelf memory devices, like current DDR2 and DDR3 DIMMs. The memory bandwidth provided by one DIMM is fixed by technology and JEDEC standards. Top specifications for DDR2 memory offer 16 bytes / cycle at 533 MHz for a total of 8.533 GB/s. Top specifications for DDR3 offer 16 bytes / cycle at 800 MHz, a total of 12.8 GB/s.

However, such maximum bandwidth can only be achieved under ideal conditions like a sequential access pattern (stride 1 accesses), or an access pattern that can be optimally distributed across the available banks. Optimizing data stream distribution across memory banks had been a great concern from the early days of supercomputing, when the memory system was the most expensive part of the dominant vector computers like the Cray-1 or the Cray-YMP.

Chip multiprocessors increase the pressure on the memory bandwidth, since many processors have to read from a shared memory system. There have been reports by Sandia National Labs of CMP systems becoming slower instead of faster after a number of cores have been integrated due to lack of memory bandwidth [45]. The Sandia press release reports insignificant performance gains when going from 4 to 8 cores, and performance slowdowns when going to 16 cores. They explicitly identify the need to design memory systems that "provide a dramatic improvement over what was available 12 months ago".

The bandwidth offered by a top DDR3 DIMM is not enough for the increasing number of processors per chip. Therefore, it becomes necessary to use more than one DIMM channel, and organize them to offer the same performance of a single, ideal, fast device.

On the other hand, memory latency cannot be reduced adding memory channels, because it depends on the time it takes to access a datum on the SDRAM cells—and this has not improved much in the last 20 years. Because storage structures cannot be made large and fast, architects use caches—which can be small and fast—to hold data commonly used or to fetch data before is required, in order to reduce memory stalls of processing units.

In this Chapter, we start by studying the memory latency and bandwidth requirements of Castell. Then, we evaluate multiple memory system organizations targeting high performance computing applications running on Castell with 8 and 32 on-chip processors, but without the LLC cache. By removing the LLC cache of Castell, we focus on the maximum off-chip memory bandwidth that can be achieved.

5.1 SDRAM State of the Art

Most computer systems use synchronous dynamic random access memory (SDRAM). They are dynamic random access memories (DRAMs) synchronized with the system bus. Jedec standards regulate the clock frequency, latency of the commands, pin distribution, voltage ranges. The main standards used today are SDRAM double data rate (DDR): DDR2 and DDR3. Double rate means that data is transmitted on the rising and falling edges of the system bus clock signal.

Figure 5.1 shows the structure of generic SDRAM arrays. They are formed by a 2 dimensional array of capacitor cells. Because the cells are capacitors, they require sense amplifiers that convert their charge in the corresponding digital value, the problem is that the read is destructive, so it also requires a buffer that remembers their value and restores it once it has being used. Furthermore, the array cells have to be refreshed periodically since capacitors loose their charge through leakage.

Physical addresses must be divided in Row and Column directions. The row address selects a page, which is loaded by the sense amplifiers, and it is stored on the row buffers. Then, the Column address can select the corresponding row where the data is located. Once a datum is in the row buffer, it is sent to the Memory controller using both raising and falling edges of the clock.

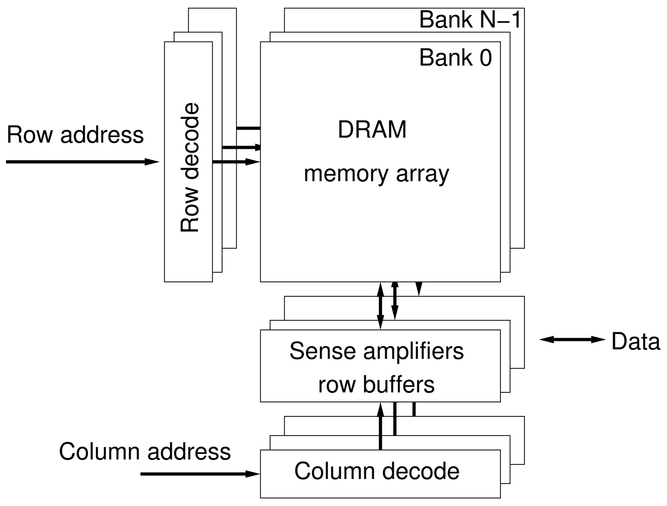


Figure 5.1: Internal structure of a modern DRAM.

DRAM organization and memory controllers work together to provide the data flux between DRAMs and microprocessors that common applications require. Figure 5.2(a) shows the basic access sequence. The memory controller sends an activate command (Act); then the SDRAM activates the corresponding row (n) and sends it to the sense amplifiers, which read the values from the array; and then, the row is stored in the row buffer. After a time t_{RCD}, the memory controller can send the READ command for bank n and column x. This initiates a column read from the row buffer and after a time t_{CL} the data would start arriving to the memory controller.

The row buffer is important, because, as it can be seen in Figure 5.2(b) if there are consecutive accesses to the same row and bank, the memory controller can schedule the DRAM commands such that there is a continuous burst. However, Figure 5.2(c) shows that if after a read, the next access falls to the same bank, but to a different row, it is necessary to close (precharge) the row before the other is opened. It is necessary to send the precharge command, and wait a time t_{RP} before the following activate command. This type of conflict creates a bubble, or gap, in the data port output stream, reducing the performance of the SDRAM.

In order to reduce the gaps that can be generated by row conflicts, the DDR standards duplicate the data arrays in several banks that can be accessed in parallel. Figure 5.2(d) shows that if two or more consecutive accesses fall on the same row, but in different banks, the memory controller can open the rows at the same time and schedule the READ commands such that there are no gaps on the data port.

5.2 Related Work

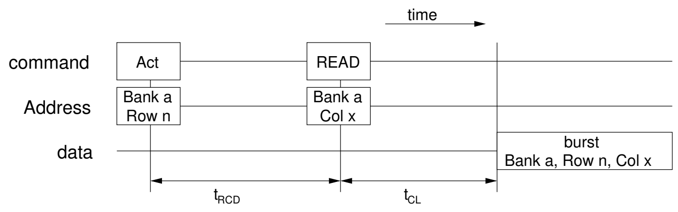
Conceptually, Castell is similar to a vector multiprocessor where the scratchpad memory is like the vector registers and DMA operations are like vector LOAD/STORE with stride 1. This means that there are some similarities in the access pattern and memory usage of these architectures.

The mechanism that distributes accesses across the installed memory modules is called storage scheme. The literature on vector processors describes three main storage schemes: interleaving, skewing [6], and linear transformations [14]. These basic schemes can provide conflict-free access for simple vector stride access patterns, like the stride 1 access of DMA operations.

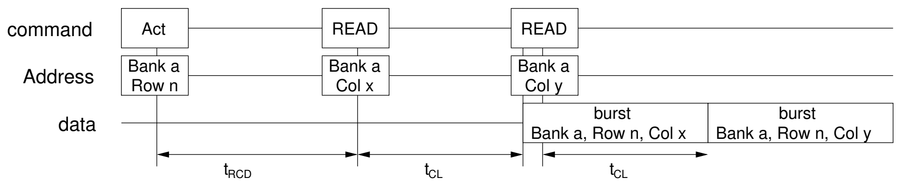
More complex stride access patterns can be optimized with variations of the interleaving scheme, as proposed by Harper III et al. [17]. Other strategies are required for conflict free accesses using general strides, like XOR strategies in Valero et al. [49], later extended at ISCA'92 by Valero et al. [50] to cope with more strides and unmatched memories. However, all these papers consider only a single vector processor.

A block interleaving method for conflict-free access on vector multiprocessors is proposed in Peiron et al. [36]. This scheme assumes a matched system where the number of memory sections (memory channels) is the same as the number of processors, and the total number of memory modules (banks) is equal or greater than the number of processor multiplied by the memory latency. In our current environment, where we consider 32 on-chip processors, and hundreds of cycles of memory latency, this proposal cannot be implemented.

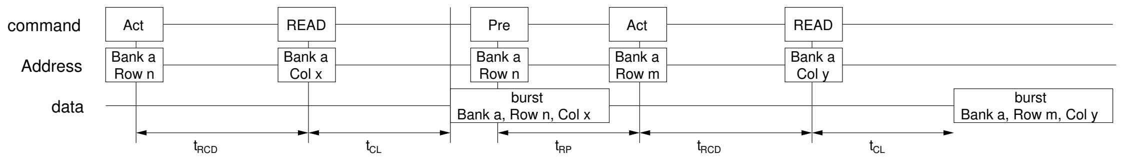
Some of these ideas have been used in systems with caches to improve the memory efficiency. Zhang et al. [59] and Burger et al. [25] it is shown that XOR address mapping can be



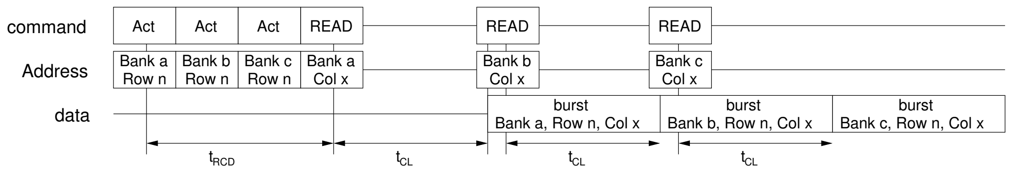
(a) Single access timing.



(b) Consecutive accesses to the same row.



(c) Row conflict on consecutive accesses.



(d) Consecutive accesses to same row different banks.

Figure 5.2: DRAM Command scheduling examples with basic timing restrictions. Act is Activate command, Pre is precharge command which loads back to the memory array the indicated row. The timing restrictions are: column access latency (t_{CL}), row address to column address delay (t_{RCD}), row to precharge latency (t_{RP}).

used to randomize bank ordering access in cache write backs, maintaining the locality properties of applications to access the row buffers on the DRAMs (contiguous row ordering). This allows the memory scheduler to use the banks more efficiently and, therefore, improve bandwidth.

XOR strategies are used to reduce conflicts on the installed devices and increase utilization; however, they can not change the granularity of the access pattern. Therefore, they can be used on top of the interleaving granularity of the system. In this chapter, we do not consider any XOR strategy since they would increase the complexity of the system and blur the interleaving granularity effects.

The interleaving strategy followed by Zurawski et al. [60] also seeks to improve the performance of memory systems with caches. It uses the cache index bits to select a DRAM page and bank, so that cache write-backs will fall in the same bank and page. The problem with this strategy is that it will not benefit from the spatial locality opportunities for the open row policy of the DRAM. Despite its simplicity, the architecture considered in this chapter does not contain caches, which makes it impossible to test it here.

Very aggressive channel and bank address mappings are proposed and used by Cuppu et al. [10] Jacob et al. [21] and by the Intel 955X Chipset [18]. The idea of these mappings is to distribute contiguous addresses on different channels in order to access them in parallel when consecutive addresses arrive to the memory controller, even if some opportunities of row locality on the DRAMs are lost, these are designed for maximum bandwidth efficiency use.

Given the simplicity of interleaving approaches, they are used as the starting point of our work. We extend previous work by examining the policies in the context of large scale CMP architectures where multiple address streams must be interleaved for fairness, and the pressure on the memory bandwidth is higher.

The aggressiveness of address mappings in this chapter is measured in terms of the placement of consecutive address words (128B) on the installed channels and banks. Considering that each DRAM page is 1KB (8 x 128 byte words), less aggressive address mappings will assign consecutive words to the same channel and bank, while more aggressive ones would spread consecutive words across different channels and banks.

The second important aspect of the memory controller design is the memory scheduler. It was shown by Valero et al. [50] and Peiron et al [37] for vector processors and by McKee et al. [30] and Rixner et al.[42] for streaming applications, that out-of-order servicing of accesses is a critical factor. Later, Rixner et al. [41] show the importance of memory reordering in web applications. In particular, they show that having a queue per bank and aggressive command reordering obtains the best bandwidth efficiency.

Ipec et al. [19] show that a CMP memory controller, on a 4-core (8-thread) processor, should adapt dynamically to the characteristics of the running applications. For such CMPs, they show that average data bus utilization for the best static controller only reaches 46 percent efficiency, and even an optimistic memory controller only achieves 80 percent.

The work by Corbal et. al. [9] proposes memory controllers for vector processors that use vector commands directly, instead of individual loads and stores. This allows the controller to reorder individual DRAM commands to exploit their locality properties efficiently.

Similarly, Mckee et al. [28, 29] show that close to 100 percent efficiency can be obtained if the access pattern is known, or if the memory requests are organized such that the memory scheduler can choose from them properly.

In this chapter, we consider a memory scheduler that chooses the first ready request from a separate queue per channel, similar to the one described by Rixner et al. [42, 41]: "To maximize request concurrency, the lowest-order bits after the DRAM page offset choose the DRAM channel, the next bits choose the bank, and the highest-order bits choose the row. Address bits are assigned so that the most-significant bits identify the smallest-scale component in the system, and the least-significant bits identify the largest-scale component in the system [11].

5.3 Architecture

Figure 5.3 shows the 32-Worker Castell evaluated in this chapter; the Workers are distributed in 4 clusters, with 2 memory controllers (MCs) connected to the global IN. Each one of the MCs manages up to 4 off-chip DDR2 memory devices (DIMM). Each one of the links in the figure is capable of transmitting 8 bytes per cycle at 3.2 GHz, 25.6 GB/s bandwidth per link. Dual-MC configurations using 8 DRAM channels can be found in some existing processors, like the IBM POWER6 [24]. This version of Castell does not include the cache hierarchy. Therefore, memory requests are sent to the off-chip memory directly; this facilitate the measurement of the applications required bandwidth.

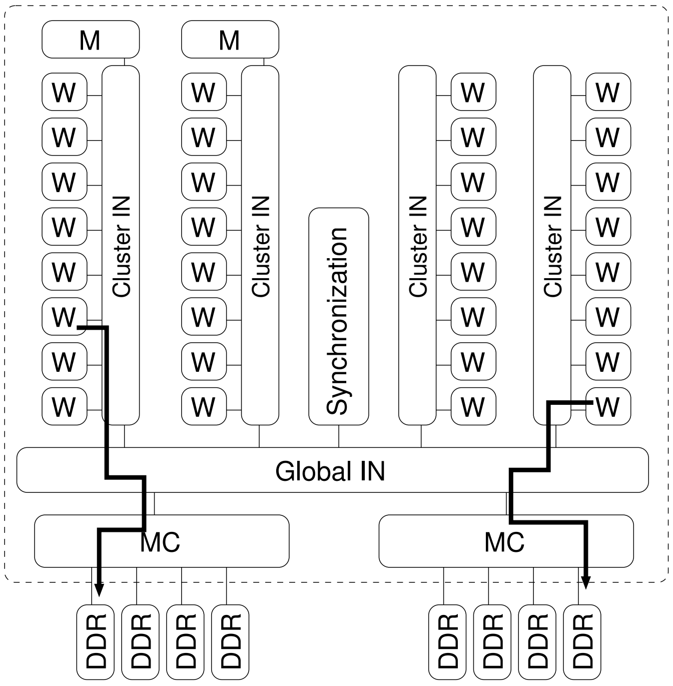


Figure 5.3: 32 processors architecture with 2 on-chip memory controllers managing 4 DRAM devices each.

The interconnection networks (both the Cluster and Global INs) allow two simultaneous communications, as long as they do not traverse the same links. For example, the figure shows a processor, from the first cluster, sending data to the first MC; and a processor, from the fourth cluster, sending data to the second MC in parallel. Accounting for two parallel data streams, the maximum bandwidth available is 51.2 GB/s. In order to achieve such 51.2 GB/s bandwidth, we use 8 DDR2-800 DIMMs of 6.4 GB/s each (4 of them connected to each MC).

Apart from the 32-Worker configuration just described, we also consider an 8-Worker configuration with only 1 cluster and a single MC with 25.6GB/s.

5.3.1 Memory System

Given the computation power available on current CMPs, and the low bandwidth of current commodity DRAM DIMM modules, it has been necessary to include several independent ports between memory and the microprocessor. However, in order to allow parallel applications to benefit from the extra bandwidth offered by the extra channels, it is necessary to use an adequate memory interleaving policy, so channels can be accessed in parallel.

The required bandwidth per task from Table 4.1 indicates that the fastest DDR3 DRAM, with 12.8GB/s bandwidth, is not enough for a parallel matrix multiply with 16 tasks, since it requires a 22.7GB/s bandwidth.

Figure 5.4 shows the projection of the estimated bandwidth (from Table 4.1) for different number of processors. It can be seen that only the FFT3D requires over 25.6GB/s for 8 processors and the other applications require much less than that. For 32 Workers, the 51.2GB/s maximum bandwidth is less than half of the required by FFT3D and barely satisfies the bandwidth of Cholesky, Kmean and MatMul.

In the first scenario, achieving 50% of maximum bandwidth is enough for most applications. However, in the second scenario, any bandwidth efficiency loss will degrade the performance of most applications. Only Knn has a large margin between the required and the maximum bandwidths.

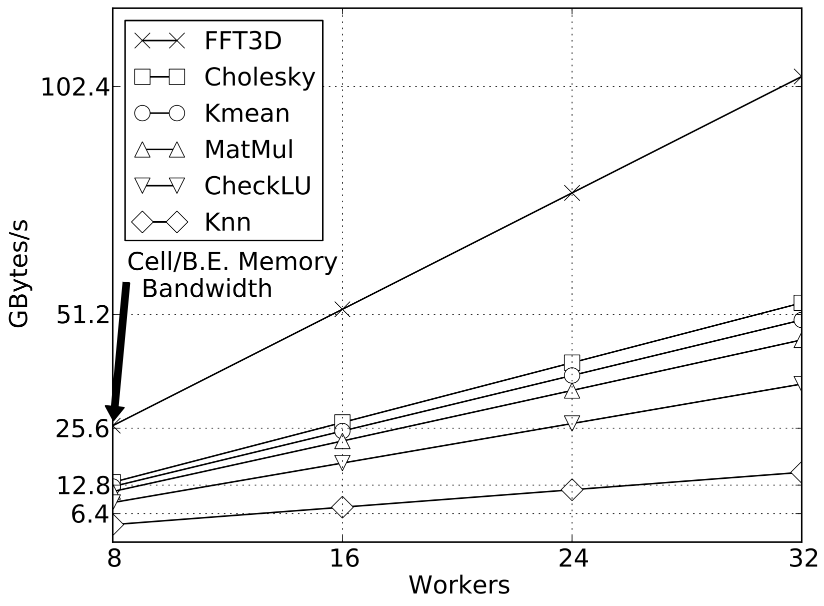


Figure 5.4: Bandwidth requirement projection for different number of on-chip processors.

5.3.2 Storage schemes

Based on previous works, as explained in Section 5.2, we use address-based interleaving policies as our memory storage scheme. The physical location of a datum is determined by bits from the address. The bits determine the memory controller (M), the device channel (H), the bank within the device (B), the row within the bank (R), and the column within the row (C). Note that the bits used to select the devices need to be consecutive in the datum's address.

DRAM modules have one row buffer for each bank. Usually the size of the buffers is 1KB or 2KB; therefore, the size of the pages of the DIMMs, which are composed of several DRAM modules, are 8KB or 16KB. There are two types of row operation modes supported by DRAMs: autoprecharge (AP) and not-autoprecharge (NAP). These modes are also called *close* and *open* page modes respectively. In NAP, the accessed page is left on a buffer, after an operation, so that future operations on the same page do not have to re-open the page. This strategy is efficient for high locality access patterns, but it would reduce performance when pages have to be closed constantly. In this later case, AP row operation is preferred.

Table 5.1 shows the interleaving policy, BHM-x, considered in this thesis. The policy indicates the address mapping to the different memory structures (MC, channel, bank, row and column). The number of bits in x determines the interleaving granularity, i.e. the number of consecutive bytes (from the least significant) mapped to a DRAM channel. For example, on a system with 2 MCs and 4 channels per MC and a BHM-4096 interleaving strategy, given a DMA transaction of 16KB, it would access just 4 DRAM channels (assuming the initial address is aligned to 4KB) as the mapping changes the channel every 4KB.

Family	Address Mapping	Description
BHM-x	R:C:B:H:M:x	Every X= 2^x bytes the MC and channel are changed

Table 5.1: Physical address bits used to determine the corresponding structure: MC (M), DRAM row (R), bank (B), channel (H), and column (C,x).

5.4 Impact of Memory Latency

Figure 5.5 shows the speedup of the applications against a real memory configuration for different memory latencies. The figure shows performance degradation of the benchmarks and applications for a 32-Workers Castell architecture as the memory latency is increased. Table 5.2 shows the parameters of the architecture used for the experiments in this Chapter. However, in this section for memory latency impact, we replaced the DDR memory system for an ideal conflict-free memory, and 102.4GB/s bandwidth provided by 4 MCs, with a configurable latency ranging from 1 cycle to 16384 cycles. The speedup is measured against an architecture with the DRAM parameters shown in the table, 2 MCS, 4 DIMMS, and 32 Workers.

The results show that for all applications performance does not degrade significantly until memory latency reaches 1024 cycles or higher. They all start degrading around 512 cycles. The FFT3D is the only application that improves performance with lower latency than a real memory module. Most DMA transfers are 16KB, which requires about 2048 cycles at 8 bytes/cycle. An additional 1024-cycle latency only increases total transfer time by 33%. Not only higher latencies, than the conventional DRAM, are tolerated (around 100 cycles), it also points out that adding a cache to reduce latency is useless, since no gain will be obtained from that. Double buffering is the main responsible for hiding latencies in a very effective way, since extra cycles are hidden by the execution of previous tasks. It is important to mention that it is not possible to compare directly the latency of real memory modules and the experiments; in the experiments

Parameter	Value	Parameter	Value
Clock	3.2GHz	DRAMs	DDR3-800
NoC Ports	25.6GB/s	DRAM BW	6.4GB/s
DIMMs	4 & 8	Page policy	closed
MCs	1, 2, 4	t_{CL}	12.5ns
MC Queue	512	t_{RAS}	37.5ns
MC sched.	in order	t_{RC}	50ns
processors	8 & 32	t_{RCD}	12.5ns
NoC Latency	1cy	t_{RP}	21.5ns
NoC BW	25.6GB/s	t_{WR}	15ns
NoC Rings	crossbar	t_{WTR}	10ns
t_{Burst}	20ns	t_{DQSS}	5ns

Table 5.2: Baseline architecture parameters.

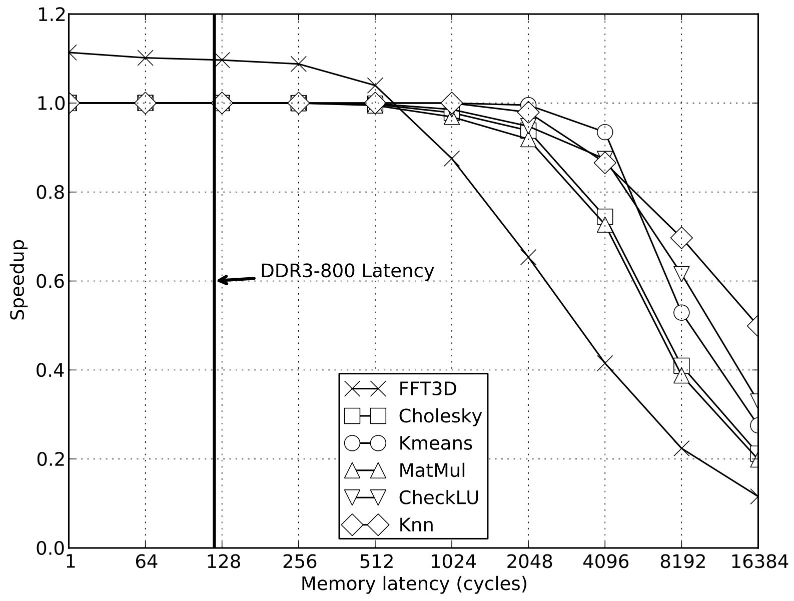


Figure 5.5: Impact of memory latency on a 32-Worker Castell.

all the accesses are conflict free while in the real systems there are bank and page conflicts that make the latency variable.

Lower latency than that of a real memory, only marginally improves performance of the FFT3D. This is because, the transpositions computations are so short that double buffering is not capable of hiding memory latency. Moreover, the latency perceived by the Workers, is not only determined by the actual latency of memory, but also on the network contention, and read and write interaction on the memory controller, generating a larger latency perception than that of the real system. FFT3D results show that performance degrades around 5% at 512 cycles latency. This is important because it is effectively 3 times the latency of real DDR3-800 memory module.

5.5 Impact of Memory Bandwidth

Once we have established that the applications tolerate latencies larger than those of real memory modules, we study the behavior of applications as we change the memory bandwidth. For example, the tasks of the matrix multiplication are 25.8μs, and require 64KB in-and-out data transfer from local memory. This translates to an average bandwidth requirement of 1.42 GB/s per worker, and an estimate of 45.4GB/s for 32 Workers.

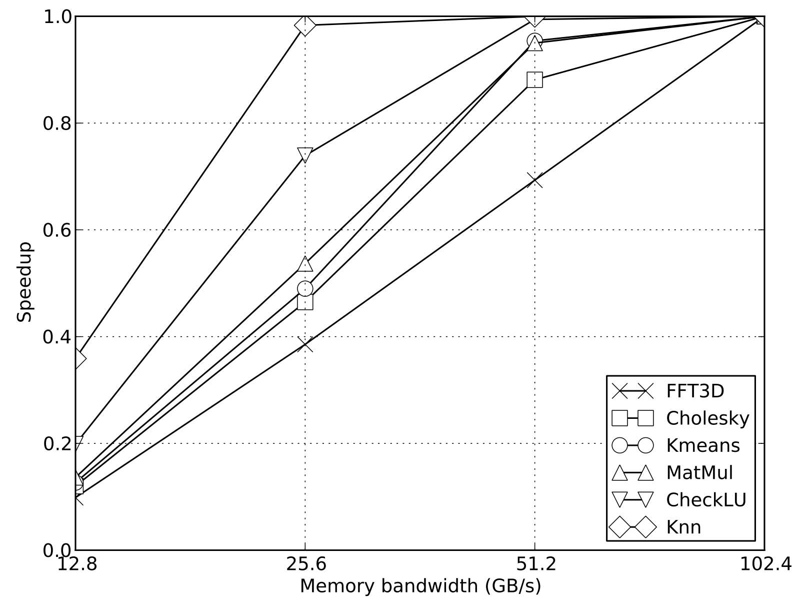


Figure 5.6: Impact of memory bandwidth on a 32-Worker Castell architecture.

Figure 5.6 shows that when the system bandwidth is lower than the required (on Figure 5.4), the performance drops quickly. For the FFT3D, a bandwidth below 102.4GB/s produces a performance drop, but the other applications only suffer a minor performance degradation until the bandwidth falls below 51.2GB/s. This result shows that applications are very sensitive to changes on the memory bandwidth.

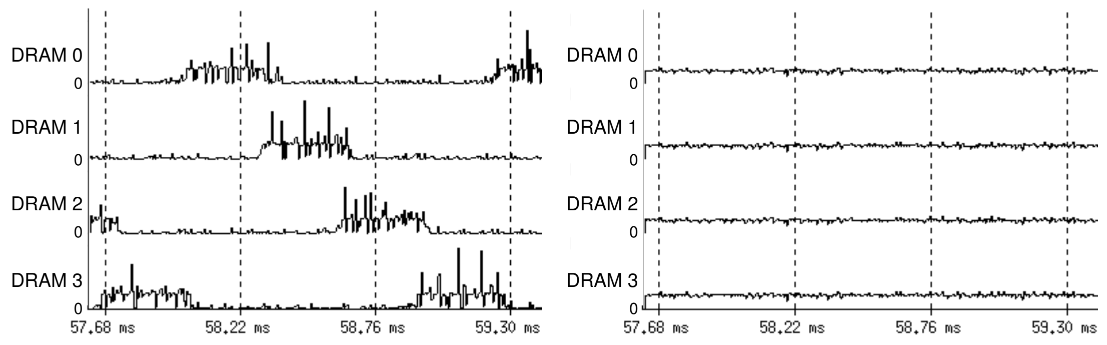
5.6 Impact of Memory Interleaving

Table 5.3 shows FFT3D simulations results for BHM-128 and BHM-4096 interleaving; it shows the FFT3D components (FFT-transpose-FFT-transpose-FFT) timing break down for an 8-Worker, 1-MC and 4-DRAM Castell architecture. The application behaves similarly for both interleaving granularities, except for the second transposition that takes 2.8 times longer for BHM-4096 than for BHM-128 interleaving.

Interleaving	FFT1D	Transposition	FFT1D	Transposition	FFT1D	total
BHM-128	20.1ms	14.7ms	20.1ms	14.1ms	20.2ms	89.2ms
BHM-4096	20.5ms	15.4ms	20.7ms	40.0ms	20.8ms	117.4ms

Table 5.3: FFT3D timings break down.

Figure 5.7 shows the number of requests per channel over time (during the second transposition) for the two interleaving strategies. The traffic that reaches the MC on the BHM-128 interleaving is nicely distributed among the 4 channels; while, in the BHM-4096 case the requests use only one channel at a time. The consequence is that, in BHM-4096, the maximum effective bandwidth is close to that of a single DRAM channel, leading to the poor performance shown for the second transposition in Table 5.3. These results show a case where the interleaving strategy has a major impact on performance for an access pattern.

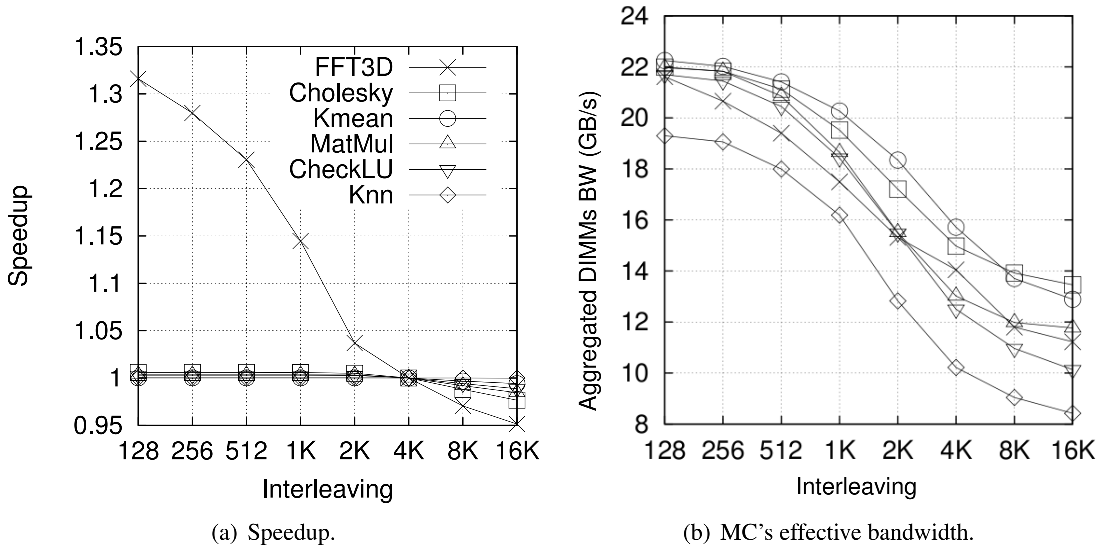


(a) Interleaving BHM-4096.

(b) Interleaving BHM-128.

Figure 5.7: Trace of the second transposition of the FFT3D for the 4 DRAM channels of the first MC, for 25.6GB/s peak bandwidth (4 DRAM channels of 6.4 GB/s). Each line shows the number of requests in a given time over time.

Figure 5.8(a) shows the speedup results for different interleaving granularities on an 8-Worker Castell with 1 MC and 4 DRAMs, against the BHM-4096 (BHM-4K) interleaving. Only the FFT3D significantly degrades with coarser granularities; but the FFT3D (Figure 5.4) is the only application that requires a bandwidth close to the maximum installed (25.6GB/s) and, as expected, is the only one that is penalized if the installed bandwidth is not efficiently used. Figure 5.8(b) shows the MC's effective bandwidth. It can be seen that when the effective bandwidth of the MC falls below the required, the applications performance degrades.



(a) Speedup. (b) MC's effective bandwidth.

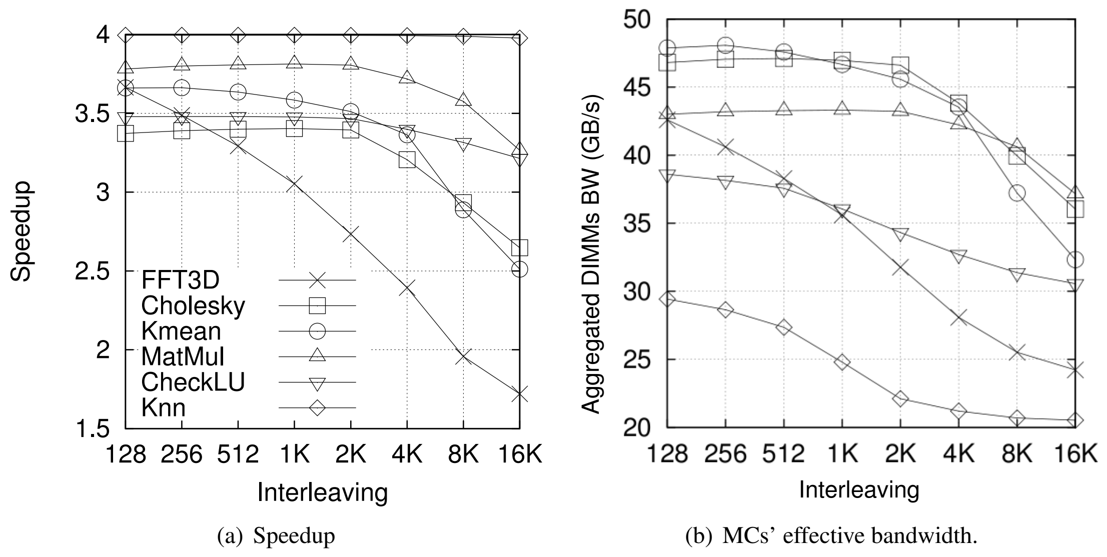
Figure 5.8: BHM-x interleaving simulation results for 8 Workers with 1 MC and 4 DRAM channels of 6.4 GB/s each. Speedup is measured against the 8 Workers and BHM-4096 interleaving.

However, as shown in Figure 5.9(a), when the installed bandwidth is close to the required by most applications, they present more performance degradation than in the 8-Worker experiments. In the case of the 32-Worker architecture, 51.2GB/s memory bandwidth (from 2 MCs) is less than the required by Cholesky (53.76GB/s) but more than the required by MatMul (45.44GB/s). The figure shows the speedup against the 8-Worker configuration with BHM-4K interleaving. It is clear that, as the interleaving granularity increases, the applications performance degrades. The MCs' bandwidth, Figure 5.9(b), shows a similar behavior as the presented for 8 Workers, where the granularity increase generates a general bandwidth decrease. Knn is not affected because it only requires a bandwidth of 15.68GB/s.

While, in the 8-Worker architecture, the bandwidth of the BHM-128 interleaving is the highest for all applications, in the case of 32 Workers, some applications have very similar bandwidth from BHM-128 until BHM-2K. This is due to the largest channels' traffic; while, in the first case, there are 2 processors per DRAM, in the second, there are 4 processors per channel. Then, the MCs additional requests per channel allow the MC scheduling to extract more bandwidth even if it is not evenly distributed—the traffic is more random.

In the case of the FFT3D with 32 Workers, when BHM-16K is used, the performance suffers a 2× slowdown compared to the one with BHM-128. Even though it has 4 times more Workers and twice the installed bandwidth than the experiment with 8 Workers, it gets less than 25.6GB/s, which is similar to the 21.8GB/s achieved by the simulation with 8 Workers and BHM-128.

Finally, it can be observed from Figure 5.8(b) that when the installed bandwidth is enough; the bandwidth usage, from fine- to coarse-grain interleaving, can decrease from 50% to 40% of the installed peak bandwidth. However, in Figure 5.9(b) we can see that when the installed bandwidth is close to the bandwidth required by applications, the bandwidth-efficiency use can decrease from 85% to 50% of the installed peak bandwidth, which can translate in 2× slowdown



(a) Speedup (b) MCs' effective bandwidth.

Figure 5.9: Simulation results BHM-x interleavings for 32 processors with 2 MCs with 4 DRAM channels of 6.4 GB/s each. Speedup is measured against the 8 processors and BHM-4096 interleaving.

(Figure 5.9(b)).

5.7 Conclusions

The current trend towards CMP architectures is increasing the memory system bandwidth requirements further than what a single off-chip channel can offer, in order to feed data to all the on-chip processors. The solution is to increase the number of DRAM channels per chip. The main concern of memory controllers for uni-processor systems with few DRAM channels, used to be bank conflicts, since this determines MC performance. However, with current multi-cores or many-cores requiring many DRAM channels the number of banks increases (for example, a 4-channel system with 8 banks each, contains 32 banks), which reduces the possibility of bank conflicts, and reduces its importance. We show that the main concern of multi-channel DRAM memory systems should be the interleaving granularity given that this determines the number of requests that can reach the channels in parallel and, therefore, the achievable bandwidth of the system.

We have explored a range of memory storage schemes, and have concluded that the most relevant factor in such storage schemes is how frequently data accesses change from one DRAM channel to the next. The interleaving frequency of 128 bytes is higher than the 4KB standard operating system page, meaning that the page allocator in the OS can not decide in which channel to map a page, because all OS pages are spread across many channels. This high interleaving frequency requirement implies that the number of channels must always be a power of 2, since all bit combinations must be valid.

Intuition and previous works show that fine grain interleaving would perform better than coarse grain ones, in terms of memory bandwidth usage. The objective of this chapter was not to propose the use of fine grain interleaving but to show the increasing importance of its use and, mainly, present the degree of performance degradation that can occur. When architecture provides enough bandwidth for applications, their bandwidth usage (from fine- to coarse-grain interleaving) can decrease from 50% to 40% of the installed peak bandwidth. However, when the bandwidth required is very close to the installed bandwidth, the bandwidth-efficiency use can decrease from 85% to 50% of the installed peak bandwidth, which can translate in $2\times$ slowdown. We showed that the degradation is due to the dynamic unbalance caused by the access pattern to the memory channels.

Chapter 6

Last Level Cache

From the previous Chapter, we can infer that a 32-Worker Castell implementation can be successful without LLC since 51.2GB/s of memory bandwidth is currently achievable, however, a 256-Worker Castell would require 8 times that bandwidth, which is not possible with current memory technology and available pins. The objective of the LLC Cache on Castell is to provide the bandwidth required by parallel applications. We have shown that applications in Castell can tolerate high memory latency when task execution is overlapped with DMA transfers, but, on the other hand, they require high memory bandwidth.

Caches are mainly used to reduce memory latency; however, multi-ported and multi-banked caches can also increase memory bandwidth as they support multiple reads or writes concurrently. Castell's LLC cache is multi-banked and fine-grained interleaved so it can provide the required bandwidth to the applications.

6.1 Related Work

Most multi-core architectures use shared and distributed last-level cache (LLC). For example, Intel's Sandy Bridge LLC is divided in as many slices (or banks) as the number of cores. The slices are connected through a multi-ring bus, so access to the core's slice is quicker than to other slices (NUCA). The target of this system is multiprogram applications with low data sharing and programming models that produce programs very sensitive to cache latency. IBM's POWER 7 LLC is shared and distributed with faster access to the part closest to each core; it also has an automatic mechanism that allocates private data as close to the core as possible, and clones some shared data on the core's close region. In both IBM's POWER 7 and Intel's Sandy Bridge, the LLC cache design prioritize single threaded applications. Similarly, the LLC of architectures like Tilera's Tile64 and AMD's are designed with the same philosophy.

SUN's Niagara 2 and Oracle's Niagara 3 contain a distributed LLC (second level) connected to the cores through a crossbar. Cores access to the different cache banks have the same latency and are equally distributed among cores; furthermore, the bank is fine-grained interleaved at cache line size, so consecutive lines are in different banks—this design favors bandwidth not latency. In a different domain, NVIDIA's Fermi GPU has a small distributed L2 cache for data sharing, which is designed for bandwidth.

6.2 Architecture

Figure 6.1 shows a 256-Worker Castell, with 32 clusters of 8 Workers each, 4 memory controllers (MCs), a 64-banks LLC cache, a synchronization module, all connected through global IN. The MCs manage 2 off-chip DDR3 memory devices (DIMM). Each one of the links in the figure is capable of transmitting 8 bytes per cycle at 3.2 GHz or 25.6 GB/s bandwidth. Each cluster also contains a Master processor.

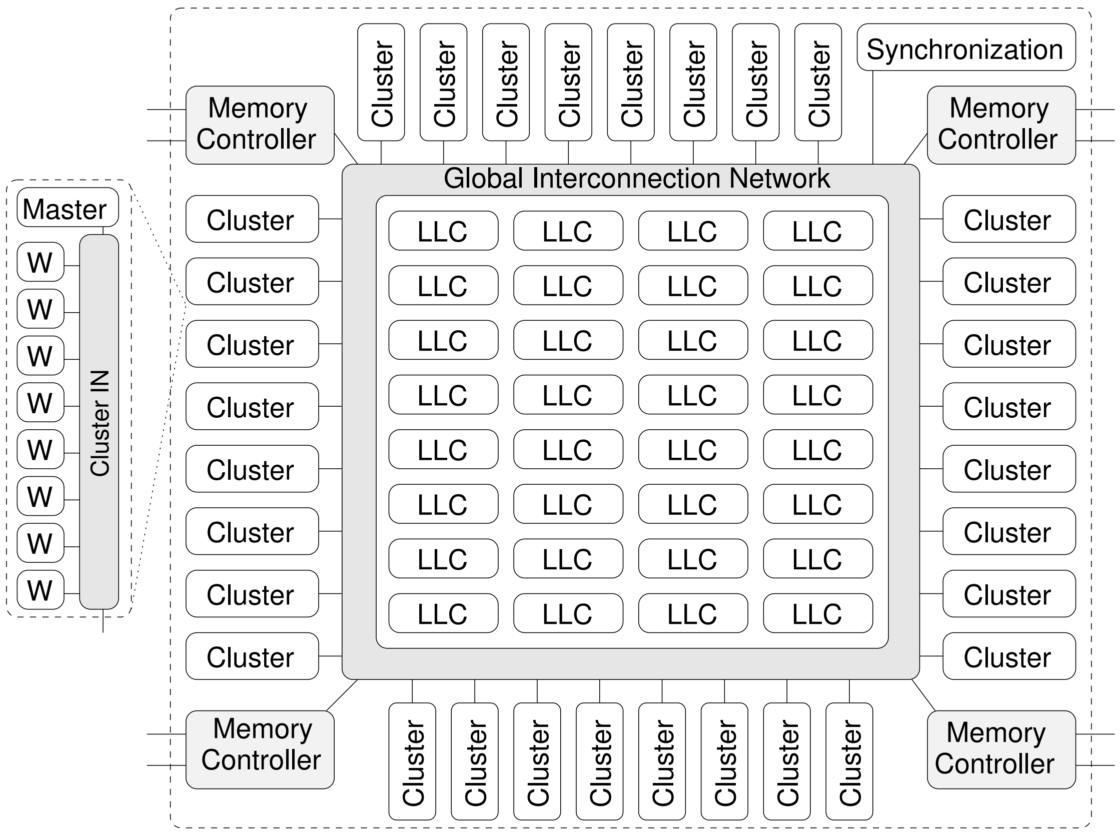


Figure 6.1: 256-Worker Castell with 2 on-chip memory controllers managing 2 DRAM devices each, and 64-banked LLC Cache.

Figure 6.2 shows the projection to 256 Workers of the estimated bandwidth for the applications (based on Table 4.1). It can be seen that FFT3D requires over 819.2GB/s for 256 Workers, which is not possible to provide only with off-chip bandwidth. Furthermore, most applications require around 409.6GB/s which is still much higher than the 150GB/s of NVIDIA's Tesla M2070—the highest memory bandwidth of a single chip today [1].

Table 6.1 presents the base parameters for the experiments performed in this chapter.

[1]To the best of our knowledge on April 2011.

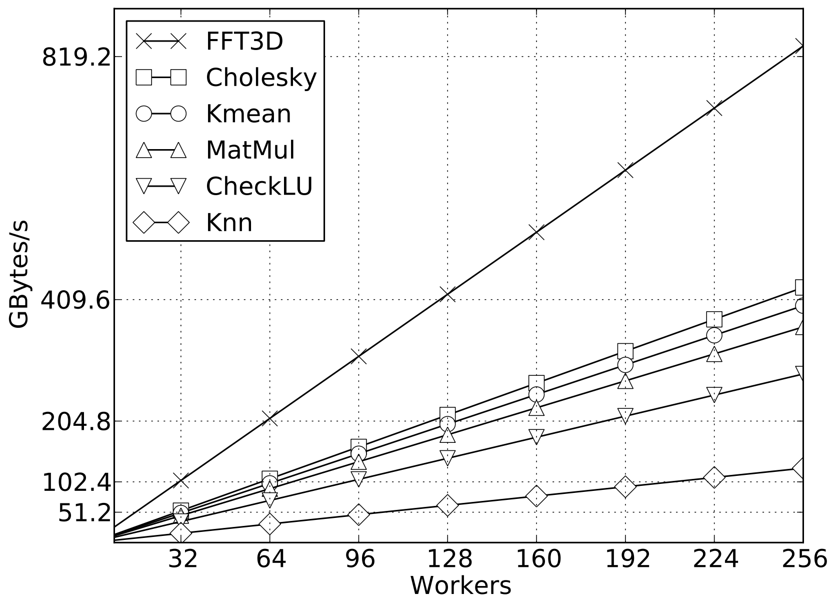


Figure 6.2: Estimated required bandwidth for 256 concurrent tasks.

6.3 Impact of Memory Latency

In Chapter 5, we show that applications, in a 32-Worker Castell, tolerate larger memory latencies than the latency of real SDRAM modules. To measure applications latency tolerance, on a 256-Worker Castell, we follow the same method than in the previous chapter; we removed the shared LLC and replaced the DRAM memory system for an ideal conflict-free memory with a configurable latency ranging from 1 to 16K cycles, with 32 MCs that provide 819.2GB/s bandwidth. In this case, though, we measure the speedup against a 256-Worker Castell, with 4 MCs each having 2 SDRAMs DDR3-1600, and 32 cache banks. Figure 6.3 shows applications speedup as memory latency is increased. As a reference, the DDR3-1600 used as a reference has an average 100 cycles latency.

The results show that the performance of all applications, except FFT3D and H.264, starts degrading when memory latency is higher than 1024 cycles. Because most DMA transfers are 16KB, they require 2048 cycles at 8 bytes/cycle. An additional 1K-cycle latency only increases total transfer time by 33%. Furthermore, double buffering also contributes to hide latencies in a very effective way. Not only higher latencies than the regular SDRAM are tolerated, it is also shown that the Castell cache latency itself, is completely irrelevant, since it will always be faster than the off-chip memory.

Comparing Figure 6.3 with Figure 5.5 from previous Chapter, we can see that in both cases the applications start loosing performance when the latency is larger than 1024 cycles.

The memory latency behavior of the FFT3D is similar to the 32-Worker latency experiments

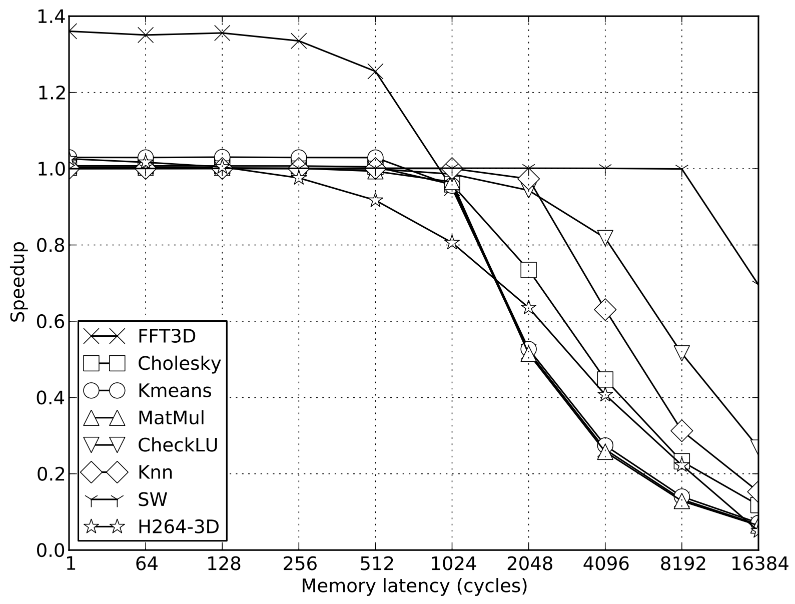


Figure 6.3: Impact of memory latency on a 256-Worker Castell. The speedup is measured against a real memory system with 819.2GB/s of bandwidth.

Parameter	Value	Parameter	Value
Clock	3.2GHz	DRAMs	DDR3-1600
NoC Ports	25.6GB/s	DRAM BW	12.8GB/s
DIMMs	8	Page policy	closed
MICs	4	t_{CL}	11.25ns
MIC Queue	512	t_{RAS}	35ns
MIC sched.	in order	t_{RC}	46.25ns
Clusters	32 (8 Workers each)	t_{RCD}	11.25ns
LLC Size	512MB	t_{RP}	11.25ns
LLC Lat.	20cy.	t_{WR}	15ns
LLC line	128B	t_{WTR}	7.5ns
LLC Assoc.	4-way	t_{DQSS}	2.5ns
LLC Banks	32 * 16 MB each	t_{Burst}	10ns

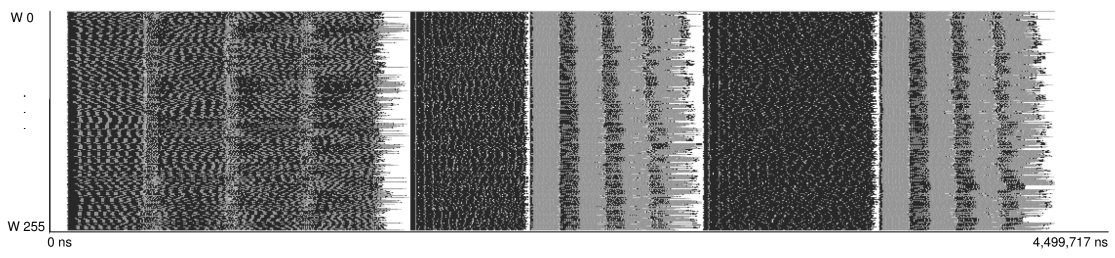
Table 6.1: Baseline architecture parameters.

of the previous chapter; while the memory latency is lower than 512 cycles, the FFT3D does not suffer any performance degradation. The performance improvement, compare to the reference, for memory latencies lower than 1024 cycles, can be understood from the traces of Figure 6.4. The FFT3D performs 3 lineal FFTs of the complete 3D matrix with 2 transpositions between them. The 3 traces of the figure clearly shows the five phases. In the trace of the reference (Figure 6.4(a)), the first FFT is very slow because the matrix is not on the cache, so the first time it is accessed, it pays the penalty of the cache plus the penalty of memory, which only has 102.4GB/s total combined bandwidth from the 4 MCs. In the figure, dark gray represents DMA waiting periods, while light gray represents execution. Comparing the duration of the first transposition of the three experiments, we can conclude that the latency of the cache banks, which is 47 cycles (8MB) from CACTI[58], produces the same performance of a no-conflict memory with 512 cycles latency, while the 1024 cycle memory takes around twice. The time difference in the second transposition between the reference and the 512 experiment is due to the queue size of the MCs in the no-conflict memory, which is 1024 entries long, while the MSHR of the cache banks is only 64 entries long.

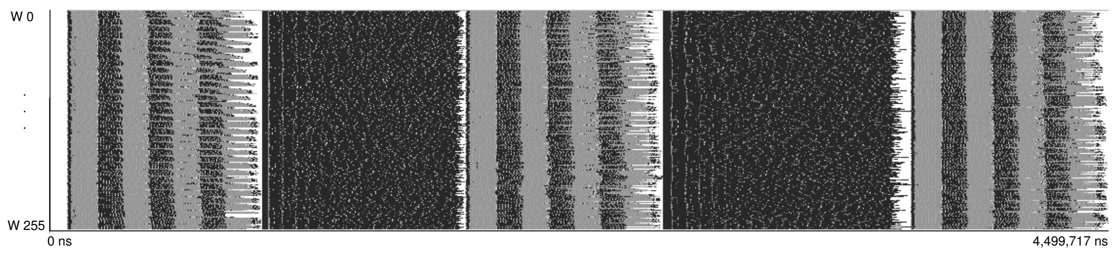
The latency tolerance of H.264 is much lower to the other applications, because it cannot fully benefit from double buffering. Not all the DMA transfers in the macroblock processing can be scheduled in advance, since the reference macroblock is not known until half-way through the decoding process. However, performance only degrades 15% for 512 cycles memory latency compared to 1 cycle latency.

6.4 Impact of Memory Bandwidth

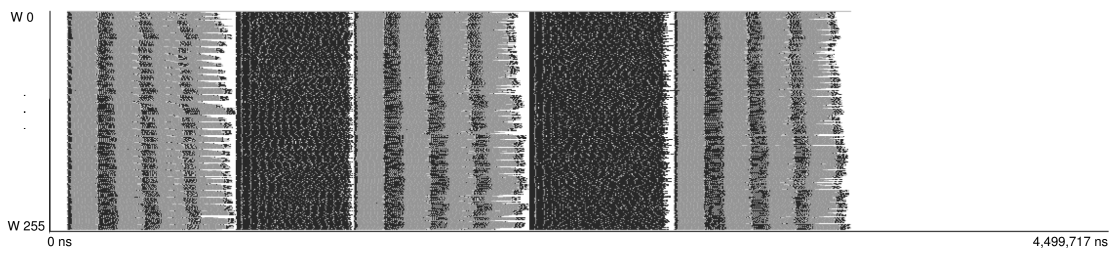
Once we have established that applications in a 256-Worker Castell can tolerate large memory latencies; we have to study the effect on the memory bandwidth. For example, Matrix multiply tasks take 25.8μs, and requires to transfer up to 64KB of data in and out of the local memory



(a) Reference experiment: 32 Cache banks, 4 MCs.



(b) 32 MCs with 1024 cycles memory latency.



(c) 32 MCs with 512 cycles memory latency.

Figure 6.4: FFT3D task traces of latency experiments. Dark gray represents DMA waiting phases, while light gray represents execution phases.

(depending on data reuse). This translates to an average bandwidth requirement of 1.42 GB/s per worker, and an estimate of 363 GB/s for 256 Workers. The estimated bandwidth required by the applications can be seen in Figure 6.2.

It is clear that we cannot provide such bandwidth from off-chip DRAM due to the limited pin count on a real system (largest systems today provide around 200GB/s). However, in order to establish the effect on the memory bandwidth, we performed experiments with no caches, Figure 6.5 shows the applications speedup for different memory bandwidths against the 32 cache banks and 102.4GB/s (from 8 off-chip DRAM channels). The experiments increase bandwidth adding DDR3-1600 memory modules. It is clear that most applications would benefit by a 819.2GB/s bandwidth and would need at least 409.6GB/s.

Even though there are some applications that do not require more than 204.8 GB/s, most applications require more bandwidth than what it is possible to provide from the number of pins and the memories available.

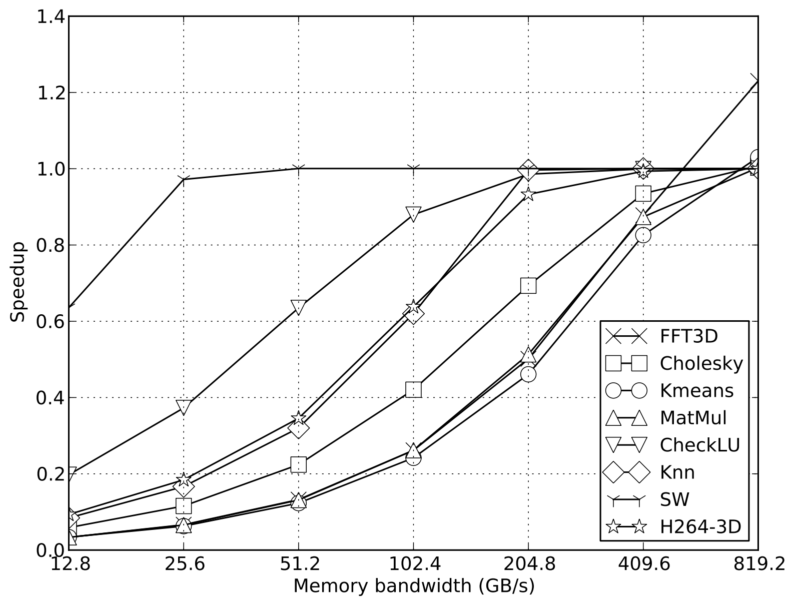


Figure 6.5: Impact of memory bandwidth on a 256-Worker Castell architecture.

6.5 Cache Banks

In order to provide the bandwidth, the best option is to add a cache that can provide the applications requirements. Since it is too expensive to create a single cache with a 819.2GB/s bandwidth, we have decided to split the cache in several banks. Figure 6.6 shows the speedup of the applications as the number of banks are increased—the memory bandwidth is 102.4GB/s, with 4 memory controllers. The speedup is measured against the 32-cache banks Castell. The results of Figure 6.2 explain why most applications reach their maximum performance with only 16 banks, which provide 409.6GB/s.

Beyond 16 banks, only MatMul and FFT3D improve performance. In MatMul this is because of task partitioning and execution; most of the time each Worker performs 8 dependent tasks, where in the first task three matrix blocks are required, in the following 7 tasks only two matrix blocks are required. So the peak required bandwidth is a bit higher than what the Figure 6.2 shows for 256 Workers.

From the figure, we see that only FFT3D improves beyond 819.2GB/s, and as seen in Figure 6.2 (estimated bandwidth for applications with 256 Workers, based on data from Table 4.1) it is the only application that requires more than that bandwidth.

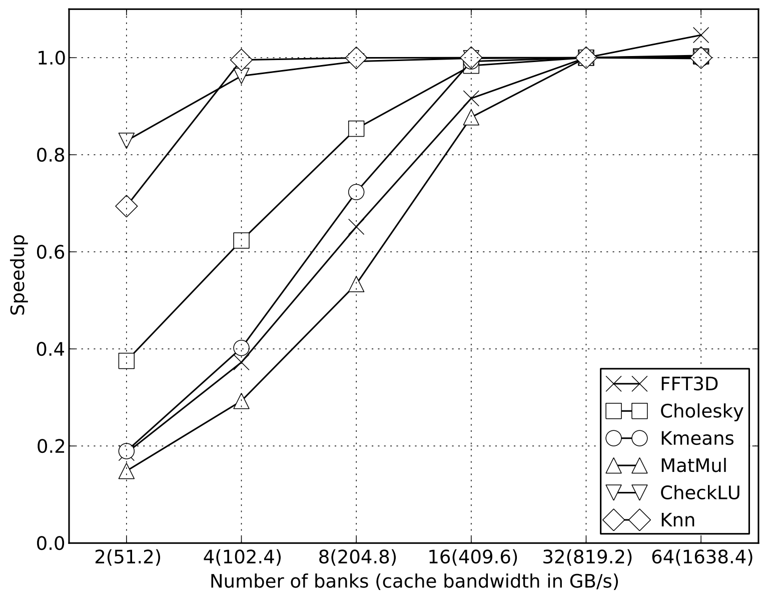


Figure 6.6: Impact of the LLC cache banks on a 256-Worker Castell and 102.4GB/s peak memory bandwidth.

6.6 Cache Interleaving

The cache interleaving is the address distribution strategy of data on the available cache banks. Figure 6.7 shows the effect of the different interleaving granularities on applications for 32-banks cache (819.2GB/s bandwidth from the caches) and 256 Workers. Only MatMul and FFT3D show significant performance degradation as the interleaving granularity increases. This shows that applications can be very sensitive to the interleaving.

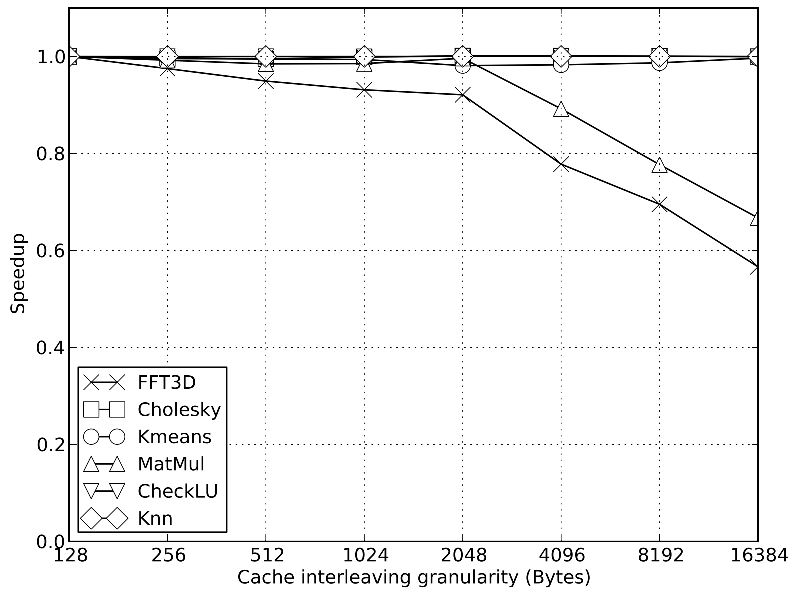


Figure 6.7: Impact of the LLC cache interleaving for 256-Worker Castell, and 1024GB/s memory bandwidth.

The interleaving on the FFT3D

The effect of interleaving can be better understood if we analize the FFT3D, however similar effects present MatMul. Figure 6.8 shows the simulation results for the FFT3D in a 256-Worker Castell, with 32 cache banks for 128 and 4096 bytes interleaving. Figure 6.8(a) top shows the task trace for the 4096 interleaving; where black represents data waiting, gray represents task execution and white regions are idle periods. There are 5 regions (separated by idle periods) corresponding to the FFT, transposition, FFT, transposition, FFT (see Section 4.2 for more details). The first FFT takes longer than the other two FFTs because the data is not on the cache at the beginning.

It can be seen that waiting periods (black) are very irregular for the FFTs. It can be explained with the bottom part of Figure 6.8(a); it shows cache banks requests number in gray scale: white represents periods of less than 10 requests, and black represents more than 4000 requests. The figure shows the irregular distribution of cache request access pattern to the cache banks. The irregular cache accesses pattern explains the behavior of the task trace. There are periods where most of the tasks are accessing the same bank, specially noticeable for the second transpose, and; therefore, some tasks have to wait a long time, if their accesses compete with the request of many other tasks. The second transposition takes several times more than the first one. This is because, the interleaving and the access pattern make the distribution of access concentrates, for some periods, on a very low number of banks.

Figure 6.8(b) shows task trace (top) and the cache-access trace (bottom) for the 128 cache interleaving experiments. They show that the waiting periods and execution times are distributed uniformly because the access pattern to the LLC banks is also well distributed. The first FFT takes much longer than the second and third FFTs because data is not in the cache the first time it is accessed.

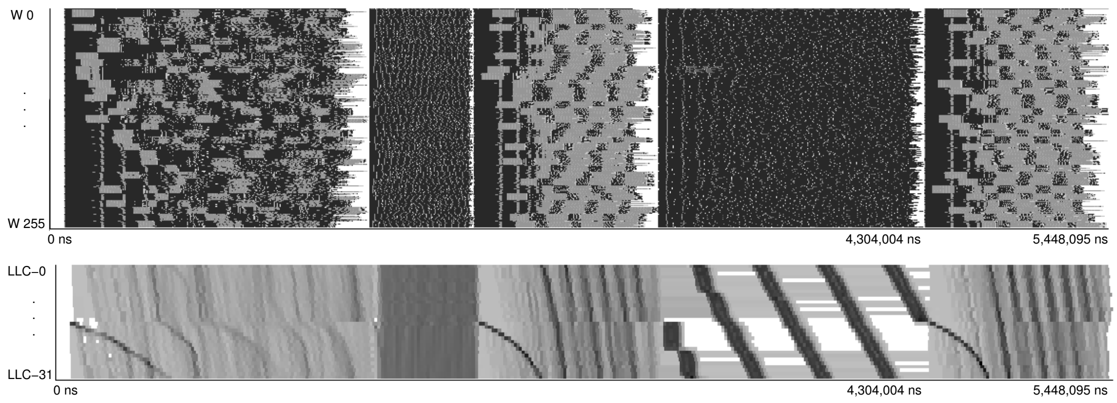
In both interleaving experiments, the first transposition take almost the same time, while the second transposition is several times longer for the 4096 interleaving experiment than for the 128 experiment. This result shows that data distribution among resources is very important, and the interleaving is critical.

6.7 The Cache as a bandwidth filtering

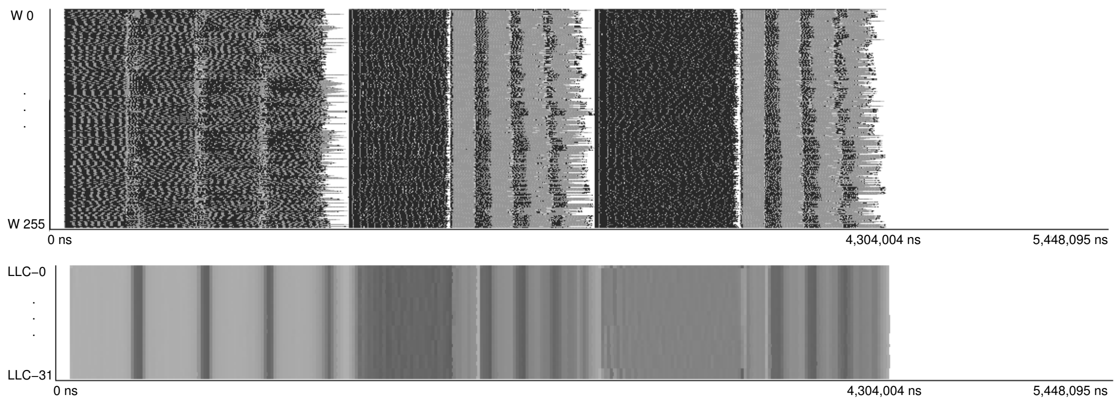
Figure 6.9 shows the performance degradation of the applications as the main memory bandwidth is decreased in a 32-bank and 256-Worker Castell. In can be seen that only Kmeans and the FFT3D improve performance beyond 102.4GB/s and only the FFT3D improves beyond 204.8GB/s. In both cases, it is due to cold cache start. This results show that a distributed and fine-grained interleaved LLC Cache is very effective as a bandwidth filter for Castell. A 51.2GB/s bandwidth, which is relatively low bandwidth for current state-of-the-art microprocessors with less than 10 processors, can still provide a performance within 10% of the maximum achievable for most applications for 256-Worker Castell.

6.8 Conclusions

As we increase Castell Workers to hundreds, we see that the applications still tolerate large memory latencies and are very sensitive to bandwidth. Moreover, given that the number of chip pins is limited, and the bandwidth of the SDRAM is relatively low, the required memory bandwidth can not be provided only with SDRAM channels. We added a distributed-shared last level cache that can provide the bandwidth required if enough banks are added. We show that it has to be fine-grain interleaved to be effective in most cases. Finally, the main result is that we could build a 256-Worker Castell with a large number of LLC banks and the memory bandwidth of today's processors.



(a) Task phases (top) and cache access pattern (bottom) for 4096 Mask Interleaving.



(b) Task phases (top) and cache access pattern (bottom) for 128 Mask Interleaving.

Figure 6.8: FFT3D task and cache-bank access pattern for 128 and 4096 interleaving masks. Dark gray represents DMA waiting phases, while light gray represents execution phases.

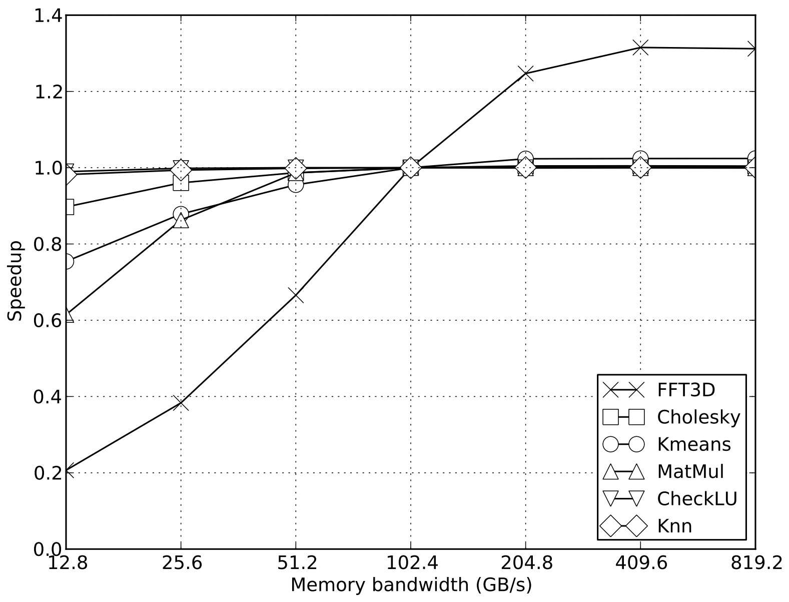


Figure 6.9: Impact of the off-chip memory bandwidth on a 256-Worker Castell with 64-banks LLC cache.

Chapter 7

On-Chip Synchronization

Synchronization mechanisms are critical for parallel applications. In previous chapters, we have assumed low synchronization-latencies, to avoid interference with the issues that we were dealing with. We showed that parallel applications in Castell scale to hundreds of processors considering current commercial SDRAM modules and pin counts that fit on state-of-the-art microprocessors. In this chapter, though, we show that synchronization latency is a problem and propose the use of hardware semaphore implementation that can provide the required synchronization latencies for Castell.

Low-level synchronization mechanisms like atomic instructions and spin-locks are hard to program, require active wait mechanisms that increase memory bandwidth pressure, and they do not scale to hundreds of processors for fine-grain tasks [47]. To solve this problem, there are several proposals; for example, transactional-memory attempts to solve this performing coarser-grain synchronization which avoids fine-grain synchronization.

Semaphores are a high-level synchronization mechanism used in many applications including the Linux OS. They are commonly used for controlling access to share resources from multiple processes. One of the problems with semaphores, is that they are high-level constructs that are programmed in software and are managed by the OS, so they are slow. Even though there are many patents for hardware semaphores, but solid-state implementations, to the best-of-out-knowledge, are only found in embedded (DSPs) systems. Figure 7.1 shows the importance of hardware semaphores on a 256-Worker Castell; it shows that software semaphores can reduce performance in more than 50% for many applications. The software semaphores latency is an estimation taken from [32].

Since synchronization delay has not become a bottleneck, General purpose CMPs do not require high-level constructs such as hardware semaphores. Furthermore, there are several problems that have to be solved for hardware semaphores on CMPs: virtualization and protection are required, limited number of physical semaphores, aliasing, and the limited size of queues. In this chapter, we address these problems.

7.1 Related Work

There have been hardware semaphores proposals since the mid 90s, but they have not reached mainstream processors. Even though most of the proposals are patents, there are also some

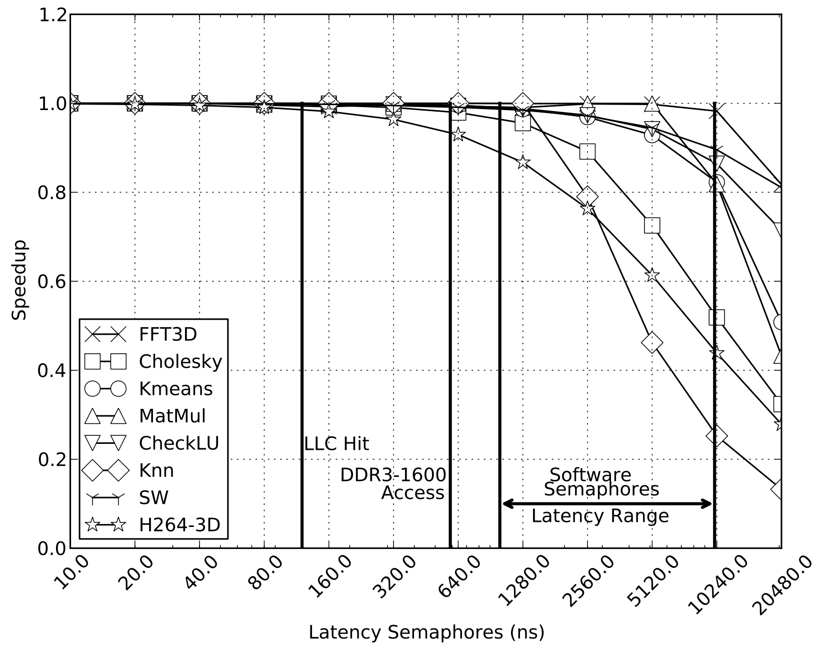


Figure 7.1: Impact of the semaphore latency for 256 Workers.

academic research that points in this direction, and at least one DSP processor with hardware semaphores.

Texas Instrument TITMS320TCI6487 3-core DSP processor contain 32 hardware general-purpose semaphores for shared resource access. They replace typical atomic operations. The semaphores are mapped in main memory, and software must map semaphores to resources. Each semaphore has 3 element queue, and there is an interruption per core for semaphore acquired notification. It contains two access types: (a) in the direct type, access is granted if the semaphore is free, otherwise nothing happens, and a load returns with busy or granted message; and (b) in the indirect type, the core is notified with an interruption when access granted, and a store is used to queue the CPU on the semaphore module.

A more theoretical proposal is the one proposed for hardware real-time-OS, Silicon TRON implementation [31], they study the OS speed requirement of embedded applications and propose the basic functionalities that require to be in hardware, and see that semaphores are one of them given that system calls are executed in hardware. The proposed implementation provides the speed and protection required.

Carter et. al. [8] show that hardware implementation of locks can reduce acquire and release times by 25-94% in distributed shared memory systems, which shows the potential benefits of hardware semaphores. Similarly, J. Santori and R. Kumar [47] proposed a barrier synchronization mechanism on hardware; they show that it is required as the number of cores increase since synchronization mechanisms speed requirement also increase.

Beside these proposals, there are several patents for hardware semaphores:

- Hardware semaphores in a multi-processor environment (U.S. Pat. 5,276,886, Jan. 4, 1994). It is a simple binary semaphore proposal that reads to a semaphore return the current state, if "clear", it is changed to "set", and returns "clear" (granted), otherwise returns "set" (not granted). No queue implementation so polling would be required.

- Circuit that implements semaphores in a multiprocessor environment without reliance on atomic test and set operations of the processor cores (U.S. Pt. 6,263,425 B1, Jul. 17, 2001). Binary semaphore implementation. A set operation tries to get a semaphore, with a priority circuitry that decides on conflicts. The test operation communicates if the semaphore was acquired. Polling is required.

- Circuit that implements semaphores in a multiprocessor environment without reliance on atomic test and set operations of the processor cores (U.S. Pat. 6,263,425 B1, Jul. 17, 2001).

- Generic Semaphore for concurrent access by multiple operating systems (U.S. Pat. 6,519,623 B1, Feb. 11, 2003). This implementation relies on a micro-kernel implementation.

- Method and apparatus for locking and unlocking a semaphore (U.S. Pat. 6,529,933 B1, Mar. 4, 2003).

- Hardware semaphores for a multi-processor system within a shared memory architecture (U.S. Pat. 6,892,258 B1, May 10, 2005). It was proposed for real-time OS.

- Method and apparatus for locking multiple semaphores (U.S. Pat. 7,143,414 B2, Nov. 28, 2006).

Other approaches, such as Intel's Carbon [23] implements the complete task scheduling on hardware. This way, they solve the contention problem of a synchronization mechanism and allow fine-grain tasks. It shows the importance of hardware solutions for parallel applications on CMPs. However, it only solves a particular Mutex problem (access to the task queue); it does not solve Mutex in general, for example, reductions and shared variables.

7.2 Hardware Semaphores

The Castell Synchronization module shown in Figure 3.5, implements a set of semaphores and their associated queues with protection provided by the OS through a common page table virtualization mechanism.

Given that hardware resources are implementation dependent and limited, when an application requires a larger number of semaphores than the implementation contains, or there are more waiting threads than places on the physical queue, main memory is used to increase their size. Furthermore, the proposed implementation requires special ISA instructions. API commands can be used to expose the hardware semaphores to developers similarly to the POSIX semaphores.

7.2.1 Virtualization

Similar to the virtual memory system, the OS must map logical to physical semaphores. As it is shown in Figure 7.2, when a processor accesses a logical semaphore, the semaphore translation lookaside buffer (STLB) is quoted for the corresponding physical semaphore. In case an entry is found the translation is made and the request is forwarded to the synchronization module. When the translation is not found on the STLB, the request is forwarded to the semaphore page table. The hardware managed replacement for fast miss resolutions, also avoids the need to call the OS when there is a STLB miss.

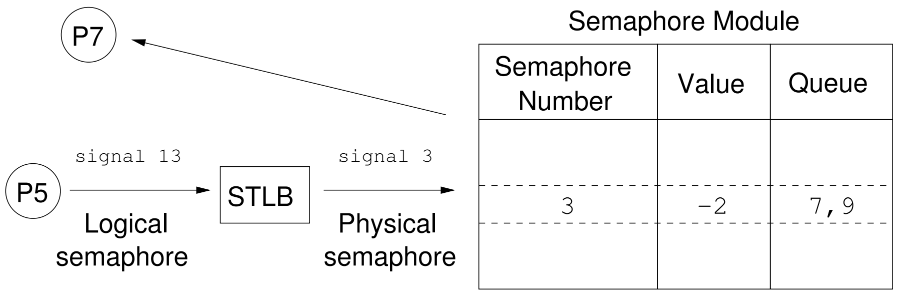


Figure 7.2: Semaphore virtualization.

In case the mapping is not found in the semaphore translation table, a segmentation fault interruption would happen. Therefore, applications must request the initialization of semaphores to the OS before they are used.

7.2.2 Active and Inactive Semaphores

Given that the number of physical semaphores is limited by the implementation, main memory is used to increase the number of semaphores. As shown in Figure 7.3, the semaphore table works like a direct-mapped cache. Each semaphore entry contains a tag, a value and its associated queue of waiting threads. The requests use the index to select the row and they test if the tag matches. On misses, the semaphore module would victimize the current entry and get the corresponding (inactive semaphore) entry form main memory.

Users can also request the OS to protect some semaphores such that no other semaphore is mapped to the same line and, therefore, would not be victimized.

7.2.3 Semaphore waiting queue

The fastest queue implementation would be one independent queue per semaphore; but this requires the knowledge of the maximum number of waiting threads at design time, as in the case of the Texas Instrument TITMS320TCI6487.

Another fast implementation would contain one queue per semaphore, but the size of the queue would be defined at semaphore creation. It would suffer form memory fragmentation as

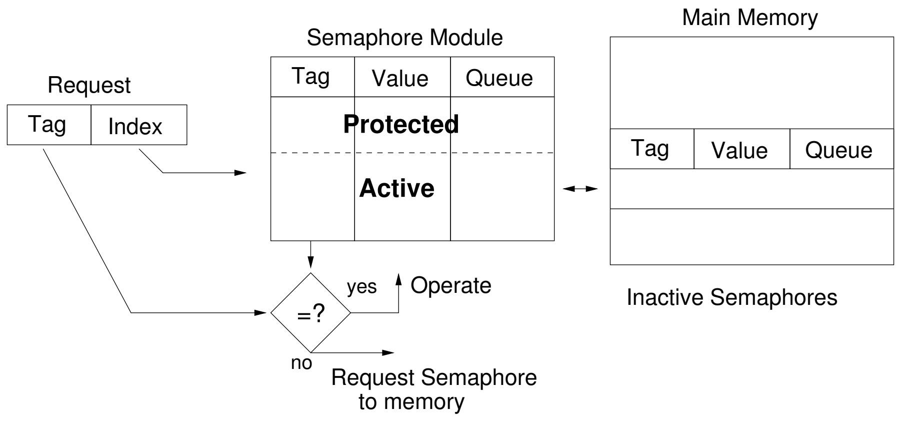
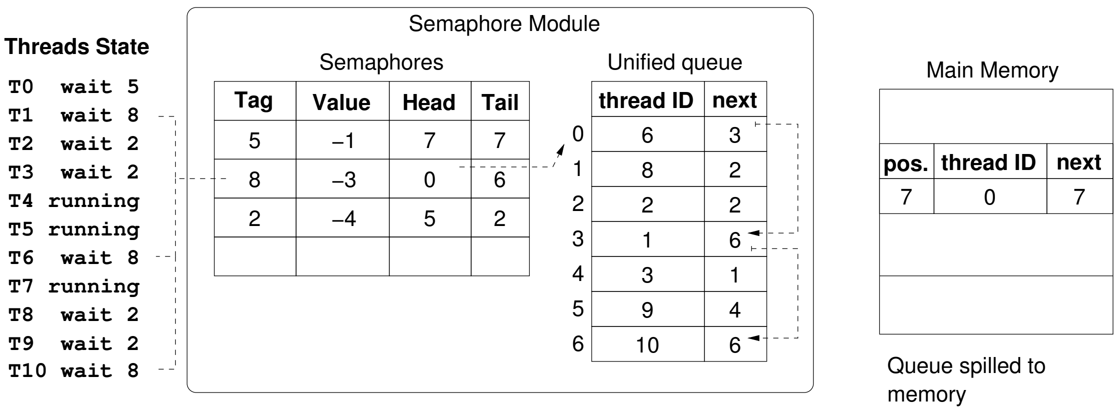


Figure 7.3: Active, inactive and protected semaphores.

space has to be assign dynamically, and semaphore initialization becomes slow since queue has to be assigned and checked.

A single shared-queue solves these problems without increasing access time significantly. Figure 7.4 shows the proposed FIFO queue implementation. Each semaphore contains its value, and pointers to the head and tail of their own queue—which makes part of the unified queue.

At the left of the figure, we show the state of 11 threads: eight are waiting and 3 are running. The dotted lines indicate the relevant information about semaphore 8. The -3 indicates that there are 3 threads waiting for semaphore 8. The head pointer has a value 0, which means that position 0 in the unified queue has oldest thread waiting for this semaphore; in this example is the thread 6. The “next” field indicates the following thread on the semaphore queue; in this case is the position 3 corresponding to thread 1; followed by position 6 which corresponds to thread 10. The “next” position of 6 is 6 itself, which means is the last position in the queue; which corresponds to semaphore’s 8 “Tail" field entry.



Threads State

```
T0  wait 5
T1  wait 8
T2  wait 2
T3  wait 2
T4  running
T5  running
T6  wait 8
T7  running
T8  wait 2
T9  wait 2
T10 wait 8
```

Tag	Value	Head	Tail
5	−1	7	7
8	−3	0	6
2	−4	5	2

	thread ID	next
0	6	3
1	8	2
2	2	2
3	1	6
4	3	1
5	9	4
6	10	6

pos.	thread ID	next
7	0	7

Figure 7.4: Example of the semaphores module unified queue, and overflow spilled to memory.

In case of queue overflow, the memory is used to hold part of the queue, as it is shown for semaphore 5 which did not have space on the unified queue. Therefore, the “Tail” points to main memory. This would make the response time much longer, but it is important for programmability reasons as it reduces segmentation faults due to lack of space for waiting threads.

7.2.4 Required Hardware

The semaphore module is a storage container similar to a cache. Assuming a 128KB of available storage for the module we can divide it half and half between the Semaphore table and the Unified queue. It is, then, possible to have 8192 queue positions which contain 4-Bytes ID and 4-Bytes “next” field. Similarly, the semaphore table can contain 4096 physical entries for Tags, Value, Head and Tail. This distribution would provide 32 bit Tag plus 12 bit Index semaphores, which gives 44 bit physical semaphores or 2^{44}.

For the experiments carried out in this thesis, the queue only requires 256 positions, and the maximum number of semaphores (with no reuse during the application) was the FFT3D with 32768. In this case, the fields were 1 Byte, except for the Tag that has to be 2 Bytes; the queue would require 256 positions and 2 Bytes each for a total of 512 Bytes. While the semaphore would be 32768 positions, requiring 5 Bytes (2 for the Tag and 1 for Value, head and tail). Therefore, a 160KB storage is required, and 160.5KB total including the queue.

7.2.5 API

In order to use Castell hardware semaphores the API has to include the following:

- `Sem-create`: system call that fills the translation table and assigns the required protection.
- `Sem-init`, `sem-wait`, `sem-post`, and `sem-try-wait` POSIX-like semaphore application interface.
- `Sem-destroy` to free physical semaphore assignment at the end execution.

7.2.6 ISA

Semaphore initialization and destruction do not need special instructions since the mapping from logical to physical is done by the OS writing on the page table. Furthermore, given that the synchronization module is mapped on main memory, some initialization functionalities, like clearing previous values of the semaphore, or resetting the queue are performed by writing on special registers. However, the following ISA semaphore instructions are required for fast access to the semaphores by the applications:

- `SEM-POST`: Store-like instruction that are routed to the semaphore module, and can be retired as soon as they are sent.
- `SEM-WAIT`: Load-like instruction that would not be retired until the semaphore answers.
- `SEM-TRY`: Load that returns true or false value depending on the state of the semaphore.

7.3 Use of Semaphores

In general, users do not have to manually program semaphores, since the runtime uses the described API for queuing and scheduling tasks. Because there are many Masters, they use the semaphores to access the task graph. The runtime inside the Workers also use the semaphores to request work on the task queues. However, users can use them manually, as in the case of the transpositions on the FFT3D, where the semaphores synchronize data read and write between tasks.

7.4 Conclusions

Synchronization is a key element for parallel applications and high level hardware synchronization facilities that can improve programmability and enable parallel applications with fine-grain tasks.

The proposed implementation, which combines a hardware semaphore module with ISA and API commands, provides Castell with a simple, but fast, synchronization mechanism for parallel applications.

Relatively small storage of about 160.5KB would be required for Castell given the applications tested without requiring any spill to memory, either for semaphores or the queue.

Chapter 8
Network On Chip

The scalability of chip multiprocessors (CMPs) is tightly coupled with that of the network-on-chip (NoC). Specifically, the NoC must be able to provide sufficient concurrency and bandwidth to transfer promptly data between the computational units and the memory system.

The hierarchical K-ring topology used in *Castell* can efficiently manage hundreds of processing units, and the concurrency and bandwidth requirements imposed on such NoC.

In this chapter, we present an evaluation of the requirements imposed on the hierarchical Castell NoC in terms of the number of simultaneous data transfers that should be supported in a given cycle. We also evaluate the impact on connectivity and performance of various aspects of the Castell architecture: the number of workers in each NoC cluster, the bandwidth of each of the interconnection links, and the number of components connected to the global K-Ring.

8.1 Related Work

Multicore chips including few high-performance processors, such as the 4-core Intel Core i7, the 6-core AMD Opteron, and the IBM POWER 7, they all use a simple bus or crossbar interconnect. The Sun Microsystem's Ultrasparc T2 has a shared, 8-bank, 4MB L2 cache and 8 processors connected through a crossbar. As observed by Asanovic et al. [1], they are not likely to scale to more than 32 cores as their increasing parallelism is already experiencing diminishing returns.

More aggressive CMP implementations like Tilera's TILE64 chip [3] and Intel's 80-core Polaris [52] use a 2D mesh interconnect. However, Grot et al. [15] show that such mesh organizations are not scalable to hundreds of processors.

Finally, heterogeneous CMP architectures such as Intel's Larrabee [48] and the Cell/B.E. [16] use a single global K-Ring interconnection where everything is connected.

Castell's NoC is similar to the one used in Rigel [22]. A detailed comparison of the Castell hierarchical NoC organization with these previous proposals is beyond the scope of this thesis. We will show that our organization does scale to an architecture with 256 worker processors, and concentrate on evaluating the parameters that impact the requirements of such hierarchical NoC design.

Parameter	Value	Parameter	Value
Clock	3.2GHz	DRAMs	DDR3-1600
NoC Ports	25.6GB/s	DRAM BW	12.8GB/s
DIMMs	8	Page policy	closed
MICs	4	t_{CL}	11.25ns
MIC Queue	512	t_{RAS}	35ns
MIC sched.	in order	t_{RC}	46.25ns
Cluster	4 to 32 workers	t_{RCD}	11.25ns
LLC Size	512MB	t_{RP}	11.25ns
LLC Lat.	20cy.	t_{WR}	15ns
LLC line	128B	t_{WTR}	7.5ns
LLC Assoc.	4-way	t_{DQSS}	2.5ns
LLC Banks	32 * 16 MB each	t_{Burst}	10ns
L1 Size	32KB(I&D)		
L1 Lat.	5cy.	NoC Latency	0-8cy
L1 Assoc.	2-way	NoC BW	25.6GB/s
L1 Line	128B	NoC Rings	4 to crossbar

Table 8.1: Baseline architecture parameters.

8.2 Design Space Exploration

This section explores the design space of Castell's global interconnect, evaluating the effect of key parameters on the overall speedup achieved, as well the concurrency imposed on the interconnect. The latter figure provides an estimate on the number of rings the interconnect requires. To explore the interconnect concurrency, we replaced the global interconnect with a full crossbar.

The evaluation explores the key parameters one-by-one, using the values from Table 8.1 for the other parameters. Specifically, all measurements are performed with 256 workers (See Figure 6.1), 32 cache banks, and 4 MICs with 2 channels each. The parameters evaluated are the workers per cluster, the link width, interconnect latency, and the number of interconnect rings.

8.2.1 Workers per Cluster

The number of Workers per-cluster determines the trade-off between the load on the global interconnect and bandwidth provided to each worker, since an entire cluster shares a single 25.6GB/s link to the global interconnect.

Figure 8.1 shows the speedups, against a 256-Worker Castell, with 4 MCs each having 2 SDRAMs DDR3-1600, and 32 cache banks and 8 workers per cluster, as we change the number of Workers per cluster. As expected, the performance decreases as we increase the number of processors per cluster as the available bandwidth for each worker is reduced. The only exceptions are Knn and CheckLU. For Knn, this is explained by its low bandwidth requirements; with per task bandwidth of 0.49GB/s (Table 4.1), 32 Workers only require 15.68 GB/s, which underutilized the per-cluster link (25.6GB/s). In the case of CheckLU, although its per-task bandwidth requirements are higher than Knn's, its internal data dependencies prevent it from scaling to 256 workers.

In general, we see that grouping 8 Workers per cluster is sufficient to allow all benchmarks to scale well as it provides each worker with ample bandwidth. The only exception is FFT3D, whose excessive bandwidth benefits from few Workers per cluster. We, therefore, determine that assigning 8 workers in each cluster achieves the best trade-off between the number of ports on the global interconnect, and the overall system performance.

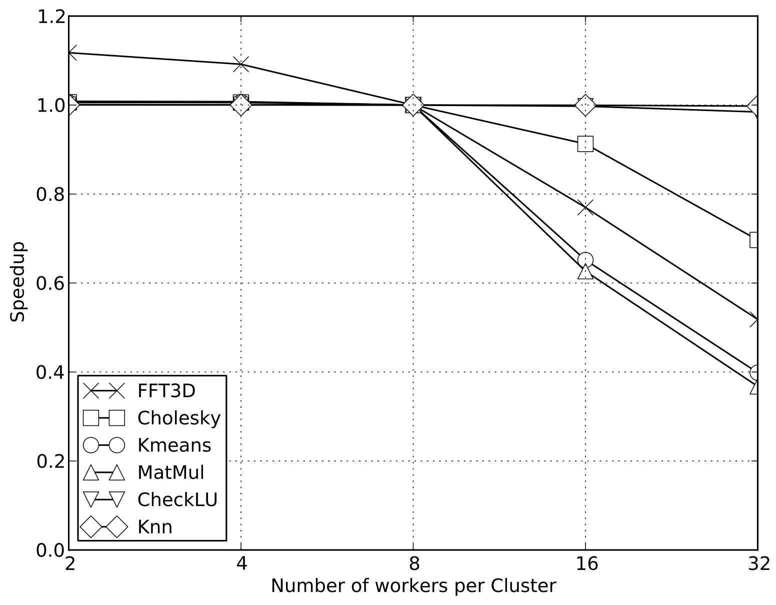


Figure 8.1: Performance impact due to the cluster size.

8.2.2 Number of Rings

We explore the effect of the number of rings on the system performance. For these experiments, we use a K-ring network to evaluate the effect of limiting the interconnect's concurrency on the overall system performance.

Figure 8.2 shows how the number rings affects the performance of the applications, compared to a cross bar network. If number of rings falls below 16 (which gives the same number of concurrent connections), most applications' performance starts to degrade, except for Knn and CheckLU that are not bandwidth limited.

These results show that a 256-Worker Castell, requires an interconnection that can keep at least 16 concurrent requests in flight. This is a small number, if we consider that this Castell configuration contains 32 clusters, 32 cache banks, and 4 MICs, which can have a maximum of

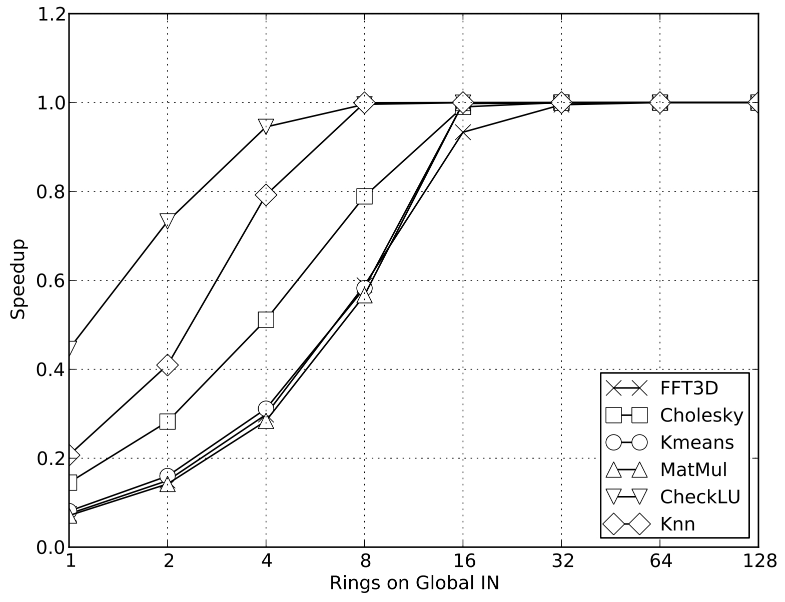


Figure 8.2: Rings

72 concurrent requests. Another way of seeing this result is that most of the applications require 16 connections multiplied by 25.6 GB/s per port gives us 409.6 GB/s sustained bandwidth. This could be seen as a contradiction with the effect of memory bandwidth on applications of Figure 6.5, which shows that at 409.6GB/s most applications suffer between 5 and 20% slowdown and the FFT3D around 30%. In these results, only the FFT3D looses around 10% while the other applications do no loose any performance. However, lets remember from Figure 6.2 that 409.6GB/s is the estimated bandwidth requirement for most applications, which corresponds to the results of this section.

Also, note that in the design there is no dedicated connection between the caches and the MICs, and they have to compete with the other connections. This simplifies the architecture since only two types of INs are required: one for the clusters and the other one for the global IN. There is no need for 4 more networks to connect 8 cache banks and a MIC. This also allows to change the memory interleaving or other characteristics of the connection between caches and MICs.

8.3 Conclusions

In this chapter, we have evaluated the main parameters of the Castell Network on Chip, a 2-level hierarchical network composed of intra-cluster interconnection rings, and a global segmented interconnection ring that connects clusters with cache banks, synchronization module, and memory controllers.

Our results show that the hierarchical organization is critical and that 8 Workers per cluster seems optimal, since increasing the number of Workers per cluster reduces performance, and reducing them does not produce many benefits.

All together, we show that a hierarchical NoC connecting 256 Workers in clusters of 8, 32 cache banks, and 4 memory controllers should support at least 16 concurrent data transfers on the global interconnect to avoid becoming the bottleneck.

Chapter 9
Research Impact

In this chapter, we review the works that have used Castell as a base architecture either to propose a particular implementation or to propose an architecture improvement. I have contributed the architecture definition and some architecture insights for their proposals.

9.1 DMA++

Architectures that contain processors with private local-memories require DMA engines to transfer data between different memories. In Castell, when data is allocated on a local memory, and there is a memory copy operation, the DMA engine transfers data between main memory and the local memory.

When the Workers contain SIMD units, data alignment becomes a problem to programmers. DMA transfer engines require that data is aligned (in the sender and receiver side) for efficient transfer; furthermore, Workers with SIMD units, also, require either aligned data to the size of the unit or special instructions to align data at execution time. The DMA++ proposal of Vujic et. al [57, 56] (where we have provided the base architecture for this work) can solve these problems by adding features that can copy data and make realignment transformations while transferring data. Figure 9.1 shows the hardware addition placement of the DMA++ Engine.

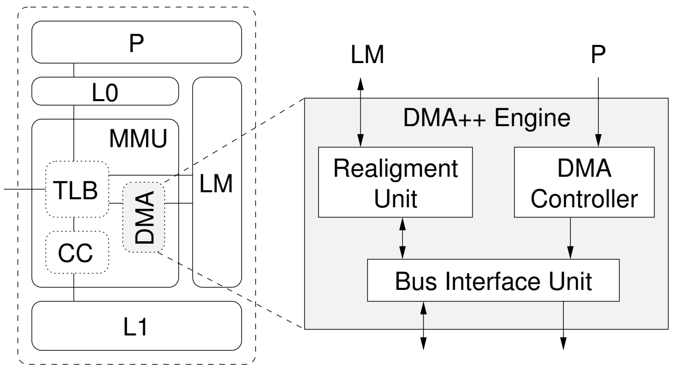


Figure 9.1: DMA++ Engine placemant.

9.2 TaskSs

Master processors speed can become a bottleneck in Castell if tasks cannot be created fast enough to keep Workers busy. Figure 9.2 shows the architecture proposed by Yoav et. al [13]. The Masters program the task pipeline module, which will be in charge of managing tasks. This design reduces the throughput requirement of the Masters the area they require.

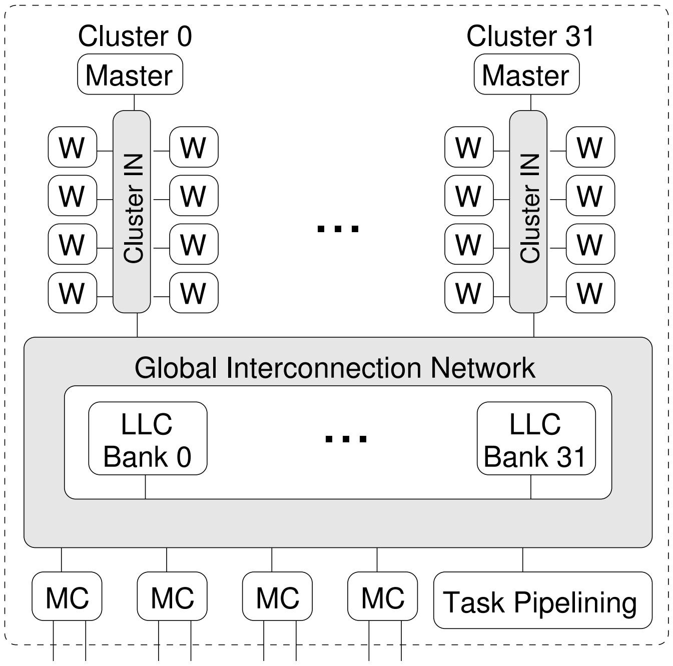


Figure 9.2: 256-Core Castell implementation with the Task Pipeline module.

In Figure 9.3, we see that the task pipeline module improves Castell scalability compared to the pure software task management.

9.3 Bioinformatics

Bioinformatics is a very important research field. Its purpose is to develop computer tools to study information from biology that can impact human health. One of the most interesting tools is DNA sequence alignment. It helps scientist to discover the relation between different DNA sequences; this helps to understand the role of genetics in some diseases.

The importance of sequence alignment, the size of sequences, and the amount of information of DNA sequences are worth considering special computer architecture design, exclusively for them. Castell framework has been used for two sequence alignment algorithms: Smith Waterman and ClustalW.

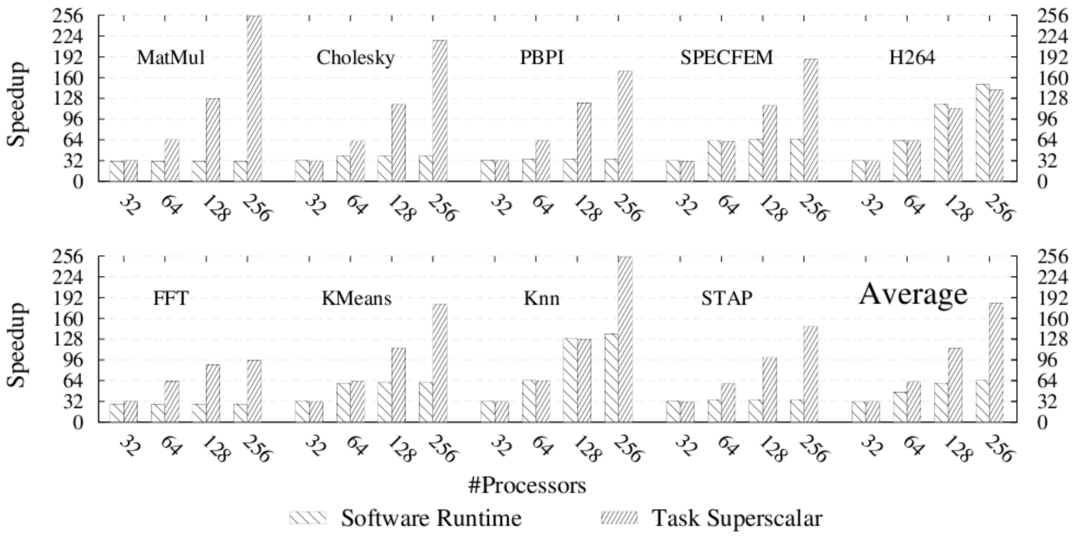


Figure 9.3: Applications speedup with task pipeline versus software management tasks.

9.3.1 Smith Waterman

The Smith-Waterman algorithm (SW) finds the best local alignment of sequences. Sanchez et. al [43] present a computer architecture for this algorithm and show that it scales to 128 Workers as shown in Figure 9.4.

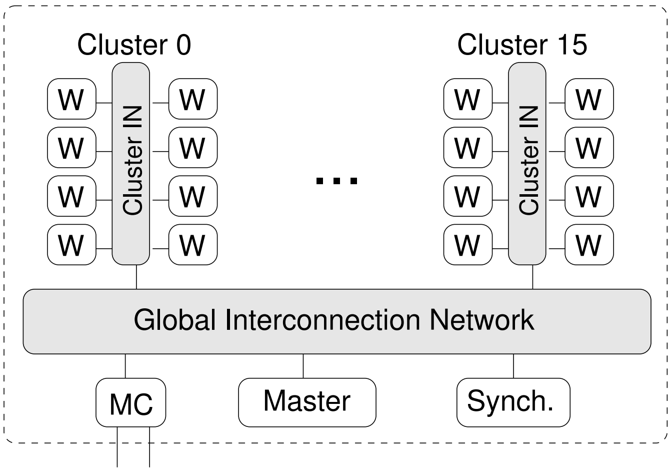
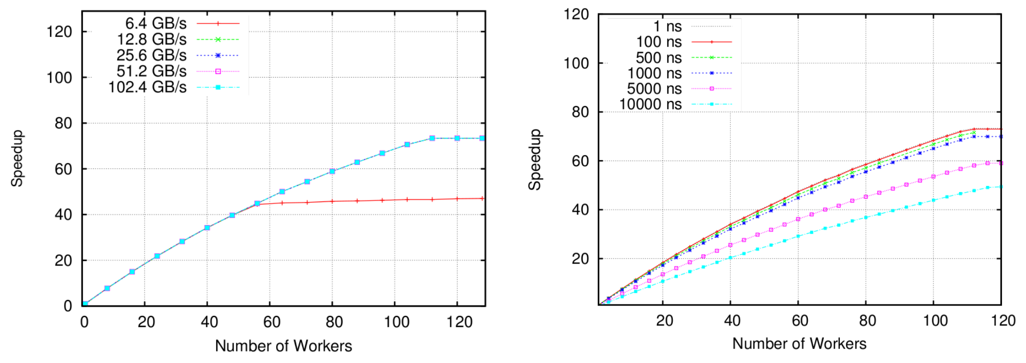


Figure 9.4: 128-Worker Castell implementation for Smith Waterman.

Note that the architecture does not include cache. The paper presents two algorithm implementations: one that uses main memory to share data between tasks, and a second one which



(a) Bandwidth requirement. (b) Synchronization latency sensitivity.

Figure 9.5: Speedup results of Smith Waterman processor architecture design exploration.

transfers data between Workers. As it is shown in Figure 9.5(a), the algorithm that transfer data between Workers' local memories only requires 12.8 GB/s of main memory bandwidth; therefore, no cache is required to obtain the algorithm highest performance.

Similarly, the paper demonstrates the need for a synchronization module; Figure 9.5(b) shows that a hardware semaphore implementation with latency below 1000 cycles is required, since this is not possible (as shown in Chapter 7) with software semaphores.

9.3.2 ClustalW

The ClustalW is a multiple sequence alignment computer program. Sebastian Isaza et. al [20] propose a Castell-based architecture that scales to 1024 Workers as shown in Figure 9.6.

In Figure 9.7 we can see some selected results from the paper. It is shown that 4 cache banks are enough; but the application is very sensitive to the synchronization latency, suggesting, as in the Smith-Waterman implementation, the need for a synchronization module.

9.4 Conclusions

In this chapter, we have shown that Castell has been used in bio-informatics algorithms and also for two architecture improvements; one concerning the DMA engines and the other for the task creation speed. Castell is an architecture framework that is useful for many types of scientific research problems and also can be useful as a base architecture for microprocessor research.

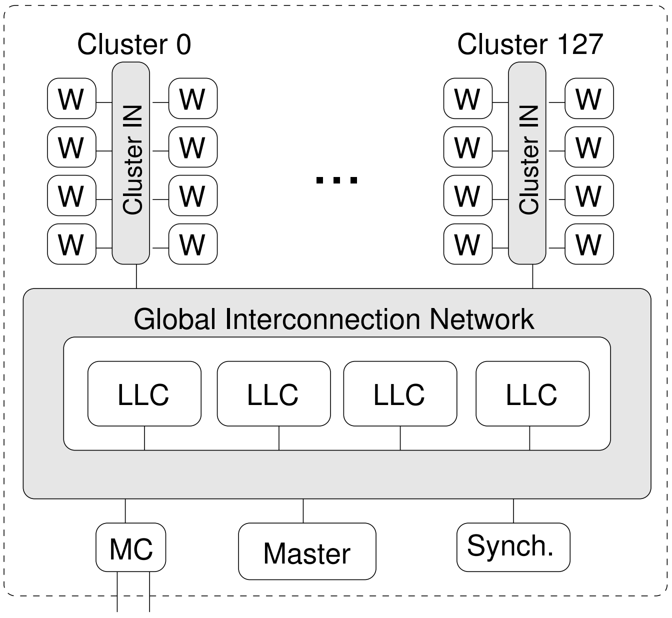
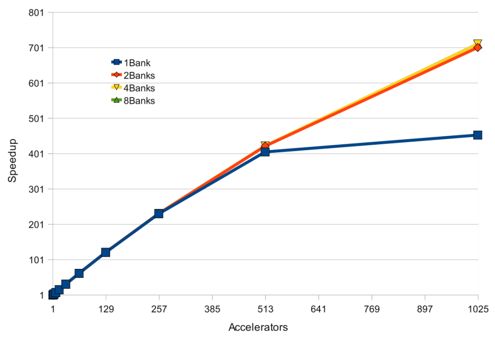
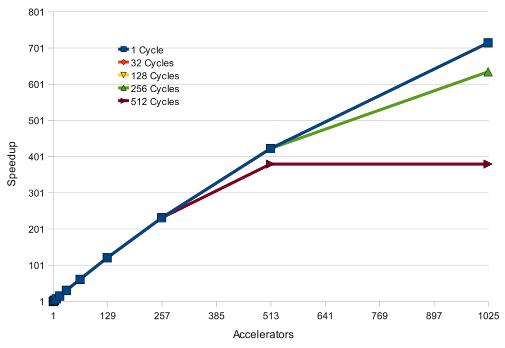


Figure 9.6: 1024-Worker Castell implementation for ClustalW.



(a) Cache bank requirements.



(b) Synchronization latency sensitivity.

Figure 9.7: Speedup results of the Clustalw architecture design exploration.

Chapter 10 Conclusions

In this thesis, we propose a scalable computer architecture framework for multicore processors. We provide a design space exploration of many of the features that can limit their scalability for parallel applications, and provide features for a runtime system to provide performance and programmability.

Castell

The main contribution of this thesis is a heterogeneous chip multiprocessor architecture framework; we show it scales to hundreds of cores for scientific applications and could be built with current pin count and SDRAM memory modules. The architecture structure facilitates the programmability using a master-worker programming model.

- Alex Ramirez, Felipe Cabarcas, Ben Juurlink, Mauricio Alvarez Mesa, Friman Sanchez, Arnaldo Azevedo, Cor Meenderinck, Catalin Ciobanu, Sebastian Isaza, and Georgi Gaydadjiev. The SARC architecture. *IEEE Micro*, 30:16–29, 2010. [39]

We developed Castell to be programmed using a task-based programming model; but different to other architectures that target a language, and have failed because of this, Castell does not contain features exclusive for task-based programming models. On the contrary, it naturally maps to parallel processing with different types of processing units, with storage modules for private and shared variables, and with a fast synchronization mechanism.

The architecture on TaskSim

We provide the definition of the architecture that TaskSim implements. Furthermore, I helped implementing some of its modules for the experiments in this thesis.

Memory organization for efficient off-Chip bandwidth use

We showed that the memory interleaving granularity has a very big impact on applications performance and prevents applications to extract the potential SDRAM performance. These results were presented in the paper *Interleaving granularity on high bandwidth memory architecture for CMPs* in SAMOS [7].

- Felipe Cabarcas, Alejandro Rico, Yoav Etsion, and Alex Ramirez. Interleaving granularity on high bandwidth memory architecture for cmps. In *Proceedings of the 2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2010)*, pages 250–257, Samos, Greece, July 2010. [7]

Last level cache organization to increase bandwidth

Caches can be used to increase the bandwidth, not only to reduce memory latency as it is done in most systems. In Castell the last level cache acts as a bandwidth filter. In order to be effective, it is necessary that last level caches are fine grain interleaved and distributed. In this thesis we show only the general effect of the caches and interleaving, however a complete study of the cache hierarchy using the Castell architecture framework was performed in the following papers.

- Augusto Vega, Alejandro Rico, Felipe Cabarcas, Alex Ramírez, and Mateo Valero. Comparing last-level cache designs for cmp architectures. In *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies*, IFMT '10, pages 2:1–2:11, New York, NY, USA, 2010. ACM. [54]

- Augusto Vega, Felipe Cabarcas, Alex Ramirez, and Mateo Valero. Breaking the bandwidth wall in chip multiprocessors. In *Proceedings of the 2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2011), Samos, Greece, July 18-21, 2011*. IEEE, 2011. [53]

Hardware semaphores to reduce synchronization latency

We showed software synchronization mechanisms are too slow for large Castell architectures. We propose a hardware semaphore module that reduces synchronization latency. We did not submit this work for peer review because the work by Vallejo et. al "Architectural Support for Fair Reader-Writer Locking" [51] was being developed in parallel at the Barcelona Supercomputing Center. However, their work shows the growing importance of hardware support for synchronization.

Hierarchical organization

As the number of cores on a chip increase, it is hard to keep them on a single ring base interconnection, we have observed that several scalable proposals use a hierarchical organization as in GPUs of Rigel. We show that this type of organization is effective for Castell, however, this is not a contribution of this work.

Impact

Castell is a computer architecture framework that has been used for a number of projects and special-purpose implementations. The objective of any computer architecture design is to have an impact in the community. We consider that the following examples, where Castell has been used, are valuable contributions of this thesis.

Castell is the base architecture of three important European projects:

- SARC: Scalable computer ARChitecture. [46]
- ENCORE: ENabling technologies for a programmable many-CORE. [12]
- IBM and BSC's Mareincognito [27].

We also provided the architecture base to the following works:

- Friman Sánchez, Felipe Cabarcas, Alex Ramirez, and Mateo Valero. Long dna sequence comparison on multicore architectures. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, pages 247–259, Berlin, Heidelberg, 2010. Springer-Verlag. [43]
- Sebastian Isaza, Friman Sánchez, Felipe Cabarcas, Georgi Gaydadjiev, and Alex Ramirez. Parametrizing multicore architectures for ClustalW. In *Proceedings of the 8th ACM conference on Computing Frontiers*, Ischia, Italy, May 2011. [20]
- Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Task superscalar: An out-of-order task pipeline. In *Proceedings of the 43rd annual ACM/IEEE international symposium on Microarchitecture*, Atlanta, GA, USA, December 2010. [13]
- Nikola Vujic, Marc Gonzalez Tallada, Felipe Cabarcas, Alex Ramirez, Xavier Martorell, and Eduard Ayguade. Dma++: On the fly data realignment for on-chip memories. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA '10)*, January 2010. [57]

Bibliography

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, EECS Department, University of California, Berkeley, Dec 2006.

[2] Arnaldo Azevedo, Ben Juurlink, Cor Meenderinck, Andrei Terechko, Jan Hoogerbrugge, Mauricio Alvarez, Alex Ramirez, and Mateo Valero. A Highly Scalable Parallel Implementation of H.264. *Transactions on High-Performance Embedded Architectures and Compilers*, 4(2), 2009.

[3] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, Matthew Mattina, Chyi-Chang Miao, Carl Ramey, David Wentzlaff, Walker Anderson, Ethan Berger, Nat Fairbanks, Durlov Khan, Froilan Montenegro, Jay Stickney, and John Zook. Tile64 processor: A 64-core SoC with mesh interconnect. In *Digest of Technical Papers of the IEEE International Solid-State Circuits Conference*, pages 88–598, 2008.

[4] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55 – 69, 1996.

[6] Paul Budnik and David J. Kuck. The organization and use of parallel memories. *IEEE Transactions Computers*, 20(12):1566–1569, December 1971.

[7] Felipe Cabarcas, Alejandro Rico, Yoav Etsion, and Alex Ramirez. Interleaving granularity on high bandwidth memory architecture for cmps. In *Proceedings of the 2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2010)*, pages 250–257, Samos, Greece, July 2010.

[8] John B. Carter, Chen-Chi Kuo, and Ravindra Kuramkote. A comparison of software and hardware synchronization mechanisms for distributed shared memory multiprocessors. Technical report, University of Utah, Salt Lake City, UT, USA, 1996.

[9] Jesus Corbal, Roger Espasa, and Mateo Valero. Command vector memory systems: High performance at low cost. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 68–77, Paris , France, 1998.

[10] Vinodh Cuppu and Bruce Jacob. Organizational design trade-offs at the dram, memory bus, and memory controller level: Initial results. Technical Report UMD-SCA-TR-1999-2, University of Maryland Systems and Computer Architecture Group, November 1999.

[11] Vinodh Cuppu and Bruce Jacob. Concurrency, latency, or system overhead: which has the largest impact on uniprocessor dram-system performance? In *Proceedings of the 28th annual international symposium on Computer architecture*, ISCA '01, pages 62–71, Göteborg, Sweden, June 2001.

[12] ENCORE: ENabling technologies for a programmable many-CORE. `http://www.encore-project.eu/`.

[13] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Task superscalar: An out-of-order task pipeline. In *Proceedings of the 43rd annual ACM/IEEE international symposium on Microarchitecture*, Atlanta, GA, USA, December 2010.

[14] Jean Marc Frailong, William Jalby, and Jacques Lenfant. XOR-schemes: A flexible data organization in parallel memories. In *Proceedings of the International Conference on Parallel Processing*, pages 276–283, August 1985.

[15] Boris Grot, Joel Hestness, Stephen W. Keckler, and Onur Mutlu. Express cube topologies for on-chip interconnects. In *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture*, pages 163–174, Raleigh, North Carolina, USA, February 2009.

[16] H. Peter Hofstee. Power efficient processor architecture and the Cell processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, 2005.

[17] David T. Harper III and Darel A. Linebarger. A dynamic storage scheme for conflict-free vector access. In *Proceedings of the 16th annual international symposium on Computer architecture*, ISCA '89, pages 72–77, Jerusalem, Israel, June 1989.

[18] Intel. *Intel 955X Express Chipset*, April 2005.

[19] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 39–50, Beijing, China, June 2008.

[20] Sebastian Isaza, Friman Sánchez, Felipe Cabarcas, Georgi Gaydadjiev, and Alex Ramirez. Parametrizing multicore architectures for ClustalW. In *Proceedings of the 8th ACM conference on Computing Frontiers*, Ischia, Italy, May 2011.

[21] Bruce Jacob, Spencer W. Ng, and David T.Wang. *Memory systems: cache, DRAM, disk.* Morgan Kaufmann Publishers, Burlington, MA 01803, USA, 2008.

[22] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *Proceedings of the 36th annual International Symposium on Computer Architecture*, pages 140–151, 2009.

[23] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 162–173, 2007.

[24] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. Ibm power6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, November 2007.

[25] Wei-Fen Lin, S.K. Reinhardt, and D. Burger. Reducing dram latencies with an integrated memory hierarchy design. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 301–312, 2001.

[26] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: a modular reconfigurable architecture. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 161–171, New York, NY, USA, 2000. ACM.

[27] Mareincognito. `http://www.zurich.ibm.com/sys/servers/mare.html`.

[28] S.A. McKee and W.A. Wulf. A memory controller for improved performance of streamed computations on symmetric multiprocessors. In *Parallel Processing Symposium, 1996., Proceedings of IPPS '96, The 10th International*, pages 159–165, Apr 1996.

[29] S.A. McKee, W.A. Wulf, J.H. Aylor, R.H. Klenke, M.H. Salinas, S.I. Hong, and D.A.B. Weikle. Dynamic access ordering for streamed computations. *Computers, IEEE Transactions on*, 49(11):1255–1271, Nov 2000.

[30] Sally A. McKee. Hardware support for dynamic access ordering: Performance of some design options. Technical report, University of Virginia, Charlottesville, VA, USA, 1993.

[31] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai. Hardware implementation of a real-time operating system. *TRON Project Symposium,*, 0:34, 1995.

[32] Kevin M. Obenland. The use of posix in real-time systems, assessing its effectiveness and performance. Technical report, The MITRE Corporation, September 2000.

[33] Kunle Olukotun, Lance Hammond, and James Laudon. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2007.

[34] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, pages 2–11, New York, NY, USA, 1996. ACM.

[35] W. R. Pearson. Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms. *Genomics vol 11(3)*, pages 635–650, 1991.

[36] Montse Peiron, Mateo Valero, and Eduard Ayguadé. Synchronized access to streams in SIMD vector multiprocessors. In *Proceedings of the 8th international conference on Supercomputing*, pages 23–32, Manchester, England, 1994.

[37] Montse Peiron, Mateo Valero, Eduard Ayguade, and Tomas Lang. Vector multiprocessors with arbitrated memory access. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 243–252, 1995.

[38] Francisca Quintana, Jesus Corbal, Roger Espasa, and Mateo Valero. Adding a vector unit to a superscalar processor. *Proceedings of the 13th international conference on Supercomputing*, pages 1–10, 1999.

[39] Alex Ramirez, Felipe Cabarcas, Ben Juurlink, Mauricio Alvarez Mesa, Friman Sanchez, Arnaldo Azevedo, Cor Meenderinck, Catalin Ciobanu, Sebastian Isaza, and Georgi Gaydadjiev. The SARC architecture. *IEEE Micro*, 30:16–29, 2010.

[40] Alejandro Rico, Alex Ramirez, and Mateo Valero. Available task-level parallelism on the cell be. *Scientific Programming*, 17(1-2):59–76, January 2009.

[41] S. Rixner. Memory controller optimizations for web servers. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 355–366, Dec. 2004.

[42] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, and J.D. Owens. Memory access scheduling. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 128–138, 2000.

[43] Friman Sánchez, Felipe Cabarcas, Alex Ramirez, and Mateo Valero. Long dna sequence comparison on multicore architectures. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, pages 247–259, Berlin, Heidelberg, 2010. Springer-Verlag.

[44] Friman Sánchez Castaño, Alex Ramirez, and Mateo Valero. Quantitative analysis of sequence alignment applications on multiprocessor architectures. In *Proceedings of the 6th ACM conference on Computing Frontiers*, pages 61–70, 2009.

[45] Sandia National Laboratories, `http://www.sandia.gov/news/resources/releases/2009/multicore.html`. *More chip cores can mean slower supercomputing*, January 2009.

[46] SARC: Scalable computer ARChitecture. `http://www.sarc-ip.org/`.

[47] John Sartori and Rakesh Kumar. Low-overhead, high-speed multi-core barrier synchronization. In *HiPEAC*, pages 18–34, 2010.

[48] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, August 2008.

[49] M. Valero, T. Lang, J. M. Llaberia, M. Peiron, Navarro, J. J., and E. Ayguade. Conflict-free strides for vectors in matched memories. *Parallel Processing Letters*, 1(2):95–102, 1991.

[50] Mateo Valero, Tomás Lang, José M. Llabería, Montse Peiron, Eduard Ayguadé, and Juan J. Navarra. Increasing the number of strides for conflict-free vector access. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 372–381, New York, NY, USA, 1992. ACM.

[51] Enrique Vallejo, Ramon Beivide, Adrian Cristal, Tim Harris, Fernando Vallejo, Osman Unsal, and Mateo Valero. Architectural support for fair reader-writer locking. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 275–286, Washington, DC, USA, 2010. IEEE Computer Society.

[52] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28tflops network-on-chip in 65nm CMOS. In *Digest of Technical Papers of the IEEE International Solid-State Circuits Conference*, pages 98–589, 2007.

[53] Augusto Vega, Felipe Cabarcas, Alex Ramirez, and Mateo Valero. Breaking the bandwidth wall in chip multiprocessors. In *Proceedings of the 2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2011), Samos, Greece, July 18-21, 2011*. IEEE, 2011.

[54] Augusto Vega, Alejandro Rico, Felipe Cabarcas, Alex Ramírez, and Mateo Valero. Comparing last-level cache designs for cmp architectures. In *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies*, IFMT '10, pages 2:1–2:11, New York, NY, USA, 2010. ACM.

[55] Carlos Villavieja, Isaac Gelado, Alex Ramirez, and Nacho Navarro. Memory management on chip-multiprocessors with on-chip memories. In *Workshop in the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, pages 1–7, Bejing, China, 2008.

[56] Nikola Vujic, Felipe Cabarcas, Marc Gonzalez Tallada, Alex Ramirez, Xavier Martorell, and Eduard Ayguade. Dma++: On the fly data realignment for on-chip memories. *IEEE Transactions on Computers*, 99(PrePrints), 2010.

[57] Nikola Vujic, Marc Gonzalez Tallada, Felipe Cabarcas, Alex Ramirez, Xavier Martorell, and Eduard Ayguade. Dma++: On the fly data realignment for on-chip memories. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA '10)*, January 2010.

[58] Steven J. E. Wilton and Norman Jouppi. CACTI: An enhanced cache access and cycke time model. *IEEE Journal of Solid-State Circuits*, 31:677–688, 1996.

[59] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM International Symposium on*, pages 32–41, 2000.

[60] John H. Zurawski, John E. Murray, and Paul J. Lemmon. The design and verification of the alphastation 600 5-series workstation. *Digital Tech. J.*, 7(1):89–99, 1995.

List of Figures

1.1 Number of transistors evolution of Intel's microprocessors and Moore's law. . . 2
1.2 Intel's microprocessors frequency evolution. 3
1.3 Intel's microprocessors power density evolution. 4
1.4 Heterogeneous versus Homogeneous cores using Pollack's rule. Assuming a fix chip area, the graphs show a performance potential for different serial and parallel code mix of applications. we compare three types of architectures: big cores, small cores, and heterogeneous. 7
1.5 Thesis Overview. 8
1.6 Castell environment and impact. 8

2.1 Inter-task dependence graph of a small 6x6 Cholesky decomposition. 11

3.1 SMT architectures. CPU is a general purpose processing unit. For all architectures the L1, L2 and L3 are the cache levels. 22
3.2 Tilera's Tile 64 microprocessor. 23
3.3 Cell/B.E. from Sony, Toshiba and IBM. SPE is synergistic processing element, and SPU synergistic processing unit; LS is local store; MFC is the memory flow controller, it contains the DMA controller; EIB is the element interconnect bus; PPE is the Power processing element, and the PPU is the power processing unit; MIC is the Memory interface controller. 23
3.4 Nvidia's Tesla GPU architecture. SP are scalar processors, SM are streaming multiprocessor, ROP is raster operation processor, and TPC is texture processor cluster. 24
3.5 a 40-Worker Castell example. In this figure, M stands for Master, W is Worker, IN is Interconnection Network, P is processing unit, CC is cache controller, L0, L1 and LLC are cache level 0, 1 and last; MMU is Memory Management Unit, LM is Local Memory, DMA is a direct memory access unit, and TLB is the Translation Lookaside Buffer. 25
3.6 Castell's multiple physical and single logical memory spaces. The API is modified to allow software to indicate a physical memory target for allocation of the malloc. The copy between memory locations when there is more than one physical spaces is converted to a DMA transfer. 26

3.7 Execution example of Equation 3.1 with the use of `memalloc` and `memcpy` on Castell. `memcpy` are DMA transfers and they are asynchronous, so Workers require a `sync` instruction before they start computing with the data being transfered. The tasks are `T0`, `T1`,`T2`. 26
3.8 State of the local memories and caches during execution of Workers' tasks of the code on Figure. 27

4.1 Graphical representation of a CellSs application trace from a Cell/B.E. execution. The different gray levels represent different computation phases of the processors and the black lines between processors represent communications and synchronizations. 30
4.2 TaskSim traces and simulation example. 31
4.3 Comparison of real Cell/B.E. execution time and TaskSim simulations. 1x means that no change has been performed on the traces, while 2x means that computation phases have being speed up by this factor. 33
4.4 H.264-3D intra-frame wavefront extending multiple in-flight frames. 35

5.1 Internal structure of a modern DRAM. 38
5.2 DRAM Command scheduling examples with basic timing restrictions. Act is Activate command, Pre is precharge command which loads back to the memory array the indicated row. The timing restrictions are: column access latency (t_{CL}), row address to column address delay (t_{RCD}), row to precharge latency (t_{RP}). 40
5.3 32 processors architecture with 2 on-chip memory controllers managing 4 DRAM devices each. 42
5.4 Bandwidth requirement projection for different number of on-chip processors. . 43
5.5 Impact of memory latency on a 32-Worker Castell. 45
5.6 Impact of memory bandwidth on a 32-Worker Castell architecture. 46
5.7 Trace of the second transposition of the FFT3D for the 4 DRAM channels of the first MC, for 25.6GB/s peak bandwidth (4 DRAM channels of 6.4 GB/s). Each line shows the number of requests in a given time over time. 47
5.8 BHM-x interleaving simulation results for 8 Workers with 1 MC and 4 DRAM channels of 6.4 GB/s each. Speedup is measured against the 8 Workers and BHM-4096 interleaving. 48
5.9 Simulation results BHM-x interleavings for 32 processors with 2 MCs with 4 DRAM channels of 6.4 GB/s each. Speedup is measured against the 8 processors and BHM-4096 interleaving. 49

6.1 256-Worker Castell with 2 on-chip memory controllers managing 2 DRAM devices each, and 64-banked LLC Cache. 52
6.2 Estimated required bandwidth for 256 concurrent tasks. 53
6.3 Impact of memory latency on a 256-Worker Castell. The speedup is measured against a real memory system with 819.2GB/s of bandwidth. 54

6.4 FFT3D task traces of latency experiments. Dark gray represents DMA waiting phases, while light gray represents execution phases. 56
6.5 Impact of memory bandwidth on a 256-Worker Castell architecture. 57
6.6 Impact of the LLC cache banks on a 256-Worker Castell and 102.4GB/s peak memory bandwidth. 58
6.7 Impact of the LLC cache interleaving for 256-Worker Castell, and 1024GB/s memory bandwidth. 59
6.8 FFT3D task and cache-bank access pattern for 128 and 4096 interleaving masks. Dark gray represents DMA waiting phases, while light gray represents execution phases. 61
6.9 Impact of the off-chip memory bandwidth on a 256-Worker Castell with 64-banks LLC cache. 62

7.1 Impact of the semaphore latency for 256 Workers. 64
7.2 Semaphore virtualization. 66
7.3 Active, inactive and protected semaphores. 67
7.4 Example of the semaphores module unified queue, and overflow spilled to memory. 67

8.1 Performance impact due to the cluster size. 73
8.2 Rings 74

9.1 DMA++ Engine placemant. 77
9.2 256-Core Castell implementation with the Task Pipeline module. 78
9.3 Applications speedup with task pipeline versus software management tasks. . . 79
9.4 128-Worker Castell implementation for Smith Waterman. 79
9.5 Speedup results of Smith Waterman processor architecture design exploration. . 80
9.6 1024-Worker Castell implementation for ClustalW. 81
9.7 Speedup results of the Clustalw architecture design exploration. 81

List of Tables

3.1 Homogeneous vs. heterogeneous systems for parallel applications. 15

4.1 Benchmarks main characteristics. 35

5.1 Physical address bits used to determine the corresponding structure: MC (M), DRAM row (R), bank (B), channel (H), and column (C,x). 44
5.2 Baseline architecture parameters. 45
5.3 FFT3D timings break down. 47

6.1 Baseline architecture parameters. 55

8.1 Baseline architecture parameters. 72