Extreme Multicores
Programming of Systems with Thousands of CPUs

Daniel Jorge Rodrigues Ferreira

Thesis to obtain the Master of Science Degree in
Information Systems and Computer Engineering

Supervisor: Prof. José Carlos Alves Pereira Monteiro

Examination Committee
Chairperson: Prof. Ernesto José Marques Morgado
Supervisor: Prof. José Carlos Alves Pereira Monteiro
Member of the Committee: Prof. Alberto Manuel Ramos da Cunha

October 2014
“What it boils down to is one per cent inspiration and ninety-nine per cent perspiration.”

Thomas Edison
Acknowledgments

I would like to thank my family for their constant support on all the steps of my education, and for providing the conditions to make this thesis possible.

To Joana I would like to address my thank you for her support thorough the development of this thesis.

Also, I would like to acknowledge the friendship and attention of my friends that have stood by me through all these years.

To my supervisor Doctor José Monteiro I would like to address my thank you for helping me to complete this thesis successfully, and for giving me the opportunity of presenting this work at the Design Automation Conference.

Finally, I would like to thank Doctor Luís Silveira for assisting me in every possible way on my visit to the Design Automation Conference.
Abstract

Current trends predict many-core processors with thousands of cores by the end of the decade. Yet currently used interconnection networks and cache coherence protocols fall short in harvesting all their promised performance. Furthermore, programming on these systems poses a new set of challenges, but also opportunities.

These problems motivate research on the topic of many-core processors, trying to provide a set of insights about the best options on interconnection networks, on cache coherency, on the memory architecture, and on the parallelization strategy. All this with the goal of providing the maximum throughput of the processor.

In this document an in-depth analysis about the previous issues is presented, giving a comparison on the available options and providing some early conclusions about which are the best ones and what characteristics should they present. The effective evaluation of these options is dependent on an accurate testing platform and a set of standardised tests to evaluate the system performance. To achieve this, it is also evaluated multiple simulators and benchmark suites.

With the conclusions withdrawn from this analysis, the first stepping stone of the experimental work is concluded, defining which tests should be defined and which parameters should be tuned.

The systematic execution of standardised tests on different architectures, followed by a detailed analysis of the results obtained, leads to verifiable conclusions about the afore mentioned options.

This research indicated the Sniper simulator and the PARSEC benchmark suite as the best options. Using them it was possible to perform experiments that pointed to the use of a 2D Torus Cluster interconnection network, and to the use of a shared L3 cache coupled with a directory as the best options for building a many-core processor.

Keywords

Many-core processors; Interconnection networks; Cache coherence; Thousands of CPUs; Simulators; Parallel programming.
Resumo

As tendências actuais permitem antever processadores com um número elevado de núcleos no final da década. No entanto, as redes de interligação e os protocolos de coerência de cache existentes não permitem extrair toda o potencial prometido para estes novos processadores. Para além disto, programar adequadamente para estes novos sistemas impõe uma série de desafios, bem como muitas oportunidades.

Estes problemas motivam investigação sobre as arquitecturas deste tipo de processadores. Esta deve tentar definir quais são as melhores opções na escolha de redes de interligação, protocolos de coerência de cache, arquitectura de memória, e na estratégia de paralelização. Isto com objectivo final de obter o maior desempenho possível do processador.

Neste documento é apresentada uma análise abrangente das questões levantadas, fornecendo uma comparação detalhada das múltiplas opções existentes, culminado com uma discussão sobre a selecção das melhores opções.

A avaliação efectiva destas opções depende de uma plataforma de testes fidedigna, um simulador, e de um conjunto de testes padronizados para avaliar o desempenho do sistema de uma forma credível. Para escolher o melhor simulador e o melhor conjunto de testes padronizados, é apresentada uma avaliação de múltiplos simuladores e de dois conjuntos de programas de teste usados na indústria.

Esta pesquisa permitiu escolher o simulador Sniper e o conjunto de testes padronizados PARSEC para a realização da investigação. Esta permitiu identificar a rede de interligação 2D Torus Cluster e a utilização de uma cache L3 partilhada com um directório como as melhores opções para a construção de um processador com um número elevado de núcleos.

Palavras Chave

Processadores de muitos núcleos; Redes de interligação; Coerência de cache; Simuladores; Programação paralela.
Contents

1 Introduction .................................................. 1
  1.1 Motivation .................................................. 2
  1.2 Objectives .................................................. 2
  1.3 Thesis Outline ............................................ 2

2 Many-core Processors .......................................... 5
  2.1 Interconnection Networks ................................ 6
     2.1.1 Electrical Networks .................................... 8
     2.1.2 3D Interconnection Networks ......................... 10
     2.1.3 On-chip Optical Nanophotonic Networks .......... 11
  2.2 Cache Coherence ........................................... 13
     2.2.1 Snooping ............................................... 13
     2.2.2 Directory .............................................. 13
  2.3 Memory Architecture ....................................... 16
  2.4 Parallel Programming ...................................... 17

3 Simulators ..................................................... 19
  3.1 Types of Simulators ...................................... 20
  3.2 Emulation vs Instrumentation .............................. 21
  3.3 Full-system Simulation vs User-level Simulation .... 22
  3.4 Simulator Comparison .................................... 22
     3.4.1 gem5 .................................................. 22
     3.4.2 CMPSim ............................................... 23
     3.4.3 Graphite .............................................. 23
     3.4.4 Sniper ............................................... 23
     3.4.5 Hornet ................................................ 24
     3.4.6 SlackSIM ........................................... 24
     3.4.7 ZSim .................................................. 24
  3.5 Simulator Comparison Conclusion ......................... 25
  3.6 Result Validation .......................................... 27
     3.6.1 SPLASH-2 ............................................ 27
     3.6.2 PARSEC .............................................. 27
List of Figures

2.1 Quad-core chip multiprocessor layout. .......................................................... 6
2.2 16-core tiled CMP, using a 2D mesh interconnect network with a distributed coherence
     directory [1]. ................................................................................................. 7
2.3 2D Torus layout [2]. ....................................................................................... 9
2.4 Spidergon-Donut layout [3]. ........................................................................... 9
2.5 Butterfly to Flattened Butterfly layout [4]. .................................................... 10
2.6 Multiple 3D interconnection networks (a) 2-D IC–2-D NoC. (b) 2-D IC–3-D NoC. (c) 3-D
     IC–2-D NoC. (d) 3-D IC–3-D NoC. [5]. ....................................................... 11
2.7 ATAC architecture [6]. .................................................................................. 12
2.8 Sparse Directory Slices structure [7]. ............................................................. 15
2.9 ACKwise Directory Line structure [6]. ......................................................... 15
4.1 Pi simulated execution time comparison. ....................................................... 37
4.2 Pi interconnection network contention. ........................................................ 37
4.3 Pi DRAM access delay. ................................................................................ 38
4.4 Pi host simulation execution time. ............................................................... 38
4.5 N-body simulated execution time. ............................................................... 39
4.6 N-body interconnection network contention. ............................................. 39
4.7 N-body DRAM access delay. ..................................................................... 40
4.8 N-body host simulation execution time. .................................................... 40
4.9 blackscholes simulated execution time. ....................................................... 41
4.10 blackscholes simulated execution time proportion relative to two cores. ..... 42
4.11 blackscholes host execution time. .............................................................. 42
4.12 blackscholes host execution time proportion relative to two cores. ............. 43
4.13 fluidanimate simulated execution time. ...................................................... 44
4.14 fluidanimate simulated execution time proportion relative to 2 cores. ......... 44
4.15 fluidanimate simulated execution time proportion relative to 8 cores. ......... 44
4.16 fluidanimate host execution time. ............................................................. 44
4.17 fluidanimate host execution time proportion relative to 2 cores. ............... 45
4.18 fluidanimate host execution time proportion relative to 8 cores. ............... 45
4.19 streamcluster simulated execution time. .................................................... 45
6.4 OpenMP Simulated Execution Time Comparison ........................................ 80

A.1 Section of the 2D Torus next to a Spidergon ........................................ A-3

B.1 Poster presented at the Design Automation Conference ........................ B-2
List of Tables

2.1 Network comparison for 1024 cores. ................................. 10

3.1 Simulator comparison. ................................................. 26
3.2 PARSEC benchmark workloads. ....................................... 27
3.3 PARSEC's workloads characteristics. ................................. 28

4.1 Architectural capabilities of gem5, Sniper, and Graphite. .......... 32
4.2 PARSEC Benchmark workloads that function on the simulators. .... 33
4.3 fluidanimate host execution time on gem5, Sniper, and Graphite in minutes. 34
4.4 PARSEC benchmark workloads that function on the simulators. .... 35
4.5 Handcrafted workloads. ............................................... 36

5.1 2D Mesh interconnection network architecture specifications, used in the simulation. .... 53

6.1 1D Heat workload description. ....................................... 77
Abbreviations

AMD  Advanced Micro Devices
ccNUMA  Cache Coherence Non-Uniform Memory Access
CMOS  Complementary Metal-Oxide-Semiconductor
CMP  Chip multiprocessor
CPU  Central Processing Unit
DES  Discreet Event Simulation
DRAM  Dynamic Random-Access Memory
DSM  Distributed Shared Memory
DSP  Digital Signal Processors
FPGA  Field-Programmable Gate Array
GPU  Graphical Processing Unit
HDL  Hardware Description Language
HPP  Heterogeneous Parallel Programming
HR  Hyper Ring
I/O  Input/Output
ILP  Instruction Level Parallelism
IOCOOM  In-order Core Out-of-Order Memory
IP  Intellectual Properties
IPC  Instruction Per Cycle
KNIF  Knights Ferry
LLC  Last Level Cache
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIC</td>
<td>Many Integrated Core</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>MPSoC</td>
<td>Multiprocessor System on Chip</td>
</tr>
<tr>
<td>NoC</td>
<td>Network on Chip</td>
</tr>
<tr>
<td>NUCA</td>
<td>Non-Uniform Cache Architecture</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non-Uniform Memory Access</td>
</tr>
<tr>
<td>ONet</td>
<td>On-chip Optical Broadcast Communication Network</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PDES</td>
<td>Parallel Discrete Event Simulation</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>SD</td>
<td>Spidergon-Donut</td>
</tr>
<tr>
<td>SoC</td>
<td>System on Chip</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static Random-Access Memory</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission Control Protocol/Internet Protocol</td>
</tr>
<tr>
<td>UMA</td>
<td>Uniform Memory Access</td>
</tr>
<tr>
<td>WDM</td>
<td>Wavelength Division Multiplexing</td>
</tr>
</tbody>
</table>
1 Introduction

Contents

1.1 Motivation ................................................................. 2
1.2 Objectives .............................................................. 2
1.3 Thesis Outline .......................................................... 2

1
1.1 Motivation

Today's multicore processors are the result of the pursuit of more computational power. These are part of the tendency to increase the number of cores per processor, that has already lead to some sixty-four core high-end processors. Also, it is predicted that by the end of the decade, there will be thousand core processors [8].

This new paradigm brings a set opportunities and challenges, namely on the internal processor network and on the programming model, and multiple solutions exist that try to extract the as most performance as possible. The internal processor network must be prepared to handle the high number of cores, by providing the adequate inter-core communication and cache coherence. On the other hand it is necessary to prepare today's programs for this new paradigm, by taking advantage of program parallelisation and optimisation. Both topics are on-going research topics.

The research of both this topics is desirable, but a problem emerges: How to perform tests on non-existing processors? The answer is many-core simulators. The use of simulators allows the test of possible future architectures through the execution of programs on them, thus enabling the adaptation of programs to this new paradigm. Furthermore, this type of simulations allows the identification of the most promising architectures without actually having to build them, decreasing the time and cost needed to research them.

1.2 Objectives

The purpose of this work is to develop an in-depth study of how to handle architectures with thousands of Central Processing Unit (CPU)s, by providing an analysis of these architectures, and of the parallelization of algorithms on them. In the end, a set of conclusions are drawn over what are the best choices of internal processor interconnection network, cache coherence mechanism, and on program parallelization.

In order to reach these goals three initial steps have to be taken: gather and analyse information about what are the best options on the internal structure of a processor with thousands of cores and on the parallelisation of programs; choose an accurate and reliable simulating environment that can execute programs and change the underlying architecture as freely as possible; and choose a set of benchmarks that can provide a means of performance measure of the simulated architecture.

With the previous research completed, an extensive set of simulations of different representative problems on multiple architectures are to be performed, allowing the production of performance measurements to be gathered and analysed, providing the previously mentioned conclusions and some insights on these new architectures.

1.3 Thesis Outline

This document is divided in four key chapters: Many-core Processors, Simulators, Simulator Evaluation, Architectural Exploration, and Execution Enhancement.
The Many-core Processors, Chapter 2, contains the state of the art of the many-core processors. Here it is possible to find a detailed explanation of the issues and advantages associated with them. Also, key issues such as a proper interconnection network, cache coherence mechanism and programming model are discussed.

The Simulators, Chapter 3, contains the analysis of the simulating platform that enables the correct simulation of the afore mentioned processors. Furthermore, the a set of benchmark tests is chosen.

In the Simulator Evaluation, Chapter 4, an evaluation of the simulators previously chosen is made. Here it can be found a comparison of their constraints and of the quality and consistency of their results.

In the Architectural Exploration chapter, Chapter 5, multiple options regarding the Interconnection Network and the Cache Coherence mechanism are tested and analysed, allowing to draw conclusion and extract insights about which are the most promising options for many-core processors.

In the Execution Enhancement chapter, Chapter 6, an analysis of the thread creation and mapping process is made, and an analysis of the constraints that OpenMP must abide when it is used to develop programs for many-core processors.

Finally, in Chapter 7 the overall conclusions and future work is discussed.
2 Many-core Processors

Contents

2.1 Interconnection Networks ........................................ 6
2.2 Cache Coherence .............................................. 13
2.3 Memory Architecture ........................................ 16
2.4 Parallel Programming ......................................... 17
Many-core processors [9] form a new category of multicore processors, having the high number of cores as their main characteristic, from a few hundred to several thousands of cores.

These present a set of challenges that conventional multicore processors do not. When dealing with such a high number of cores it becomes hard to provide a cache coherence scheme that is able to handle all of them efficiently, with today’s implementations being insufficient to answer all the requests and to take advantage of this new paradigm, becoming a serious bottleneck in the processor performance. On the other hand, the interconnect network of today’s multicore processors is also not suitable for this new paradigm, not being possible to use an usual global bus, based on making all the cores share a common bus, without having some serious penalties.

These two problems are the main topics of research, constituting the key to harvest the full potential of this new type of multicore processors.

2.1 Interconnection Networks

When discussing interconnection networks, or topology, the main problem at hand is: How to provide fast communication between the cores? There is no perfect answer for this question, nevertheless there are multiple interconnection networks that perform different trade-offs.

Before advancing to the discussion of the multiple interconnection networks available, and their advantages and disadvantages, it is important to provide an overview of the elements and concepts related with the usual function of the processor, explained in detail next.

A processor can be defined as the hardware part of a computer than can take an input and provide an output, by performing some operation that can be arithmetical, logical or other. To do so it has three main components: one or more cores, a cache hierarchy, and one or more buses to connect the caches to the main memory, through a memory controller. For example, Fig. 2.1 shows the layout of a typically quad-core processor.

![Figure 2.1: Quad-core chip multiprocessor layout.](image)

This type of multicores presents a small set of cores, each one is connected to its own Level 1 (L1) cache, that is connected to a Level 2 (L2) private or shared cache, that is connected to a shared bus. On newer processors there are also Level 3 (L3) private caches or one single L3 shared cache. The bus must be connected to every processor cache to be able to get input and provide output, and perform cache coherence (explained in detail in Section 2.2). When the number of cores reaches some dozens the contention becomes too high, also the shared bus becomes excessively long, be-
coming necessary to insert repeaters and amplifiers, to compensate the signal integrity degradation. This leads to an increase in power consumption and delay.

On newer processors, and on future processors, this layout is likely to change. Commodity processors nowadays are built to do more than just provide raw processing power, they provide a increasing set of hardware parts as: a Graphical Processing Unit (GPU), which may include some dedicated memory; Digital Signal Processors (DSP); generic memories; Input/Output (I/O); mixed signal modules; application specific hardware; Intellectual Properties (IP); and peripherals. These new processors are called System on Chip (SoC) [10], and they bring advantages such as higher processing capability, smaller size, lower power consumption, and lower cost.

Having an overview of the processor elements it is now possible to analyse in further detail the interconnect network options at hand.

On a SoC if a shared bus solution is used, there is a centralised arbitration unit which organises the communication between the modules. As the number of connected modules increases so does the contention. Also, the complexity of the arbitration unit, and the transmission latency increases, so it becomes obvious that a shared bus is not suitable for this situation. Hence this a good solution only for a limited number of modules.

![Figure 2.2: 16-core tiled CMP, using a 2D mesh interconnect network with a distributed coherence directory][1].

To solve the previously mentioned shared bus problems a new paradigm has been proposed, the Network on Chip (NoC) [10], as shown on Fig. 2.2. In this new paradigm the multiple modules in the processor are connected by a more complex interconnection layout forming a network, using routers to send messages, since messages may be forwarded, and not just received and sent. The presence of routers eliminates the need of a centralised arbitration unit, because the messages sent across it are now able to contain the information about their destination.

To connect the cores on a many-core Chip multiprocessor (CMP), or cores and other modules on a Multiprocessor System on Chip (MPSoC), there are multiple available technologies: electrical networks; on-chip optical nanophotonic networks [11]; 3D interconnects [12]; and on-chip wireless communication [13]. The last three face some design and manufacturing challenges. In all of them the core is not considered a sole element as in Fig. 2.1 it is considered in a tile.

A tile, as shown in Fig. 2.2 is usually constituted by the core itself with L1 instruction and data caches, both of them private, a L2 cache that can be shared or private, and a router. The latter is responsible for communicating with the interconnection network that it is connected to, not only serving as a means of input and output, but also as a forwarder of packets, using flits (flow control...
digits), that were sent by other tiles.

Next electrical networks, on-chip optical nanophotonic networks, and 3D interconnect networks are discussed. The on-chip wireless communication networks are not discussed in detail, because they present too many issues. Despite reducing the latency by a minimum of 20% on the tests performed [13], it increases the power consumption significantly, and the issue of interference with other devices is not tackled. This would make possible for the processor to stop working properly when some other external device was functioning, something that is unacceptable.

2.1.1 Electrical Networks

The electrical networks [14], [15], [16] are the most well known. They consist in having each tile connected, by one or more buses, to other tiles. These have the advantage of usually being simple and of using known manufacturing methods, but they have the disadvantage of the interconnection network usually having a big hop count from one end to the other (distance measurement). Also, having multiple crossing connections on a chip presents some problems, usually only being possible to have ten crossing communication wires [9], which may be a limitation.

There are multiple electrical networks that make different trade-offs trying to minimise these disadvantages. To evaluate and compare them the following set of key aspects should be considered:

- diameter: the greatest number of hops from one end to the other of the network;
- average inter-node distance: the average number of hops between any two tiles;
- bisection width: the minimum number of wires need to cut to split the network into two;
- cost: the number of links needed to build the network;
- router degree: the number of ports on the routers.

One key aspect for many-core processors, SoC or not, is that the network diameter is small, otherwise they will not be able to run real-time applications [9].

The multiple existing electrical networks can be divided in Direct and Indirect. The former is characterised for having every router attached to a tile, and the latter may have routers that are not connected to tiles, notwithstanding the tiles maintain their structure.

Direct topologies:

- 2D Mesh: consists in having each tile connected to its closest tiles in a grid like manner, each tile connects to a maximum of four other tiles;

- 2D Torus: similar to the previous but, seen in a three dimensional manner, forms a torus with the connections. It has the property of having “circular” buses, i.e. in the end tile it connects to the first one of that that line and/or column. (Shown in Fig. 2.3);

- Hypercube: consists in each tile having a connection for each one of the hyper cube’s dimensions;
Hyper Ring: consists in a hierarchical ring network with dual rings;

Spidergon: seen as a circle, consists in having every tile in the circumference connected to the tile at each side, and to the tile across the circle, forming a network similar to a spider’s web. (Shown in one of the sub-structures shown in Fig. 2.4);

Spidergon-Donut: consists in having multiple Spidergon structures, connected by one or more buses to allow more tiles, as shown in Fig. 2.4.

Indirect topologies:

Crossbar: consists in having a single switch that directly connects \( n \) inputs to \( m \) outputs;

Tree: consists in having switches, not belonging to any tile, connecting multiple tiles (leafs) and connecting other switches in a hierarchical manner. When a tile wants to send a message, it sends the message up from his leaf to closest common ancestor, and then it is forwarded down to recipient leaf;

Fat Tree: improves upon tree by trying to solve the problem of high contention near the root, by adding multiple roots and high order switches, increasing the bandwidth;

Butterfly: consists in having multistage, i.e. nodes at ends and switches in the middle, there is exactly one path between each pair of nodes, and each node sees a tree rooted at itself. (Shown on Fig. 2.5);

Flattened Butterfly: this has the same principle as Butterfly but instead of fewer simpler switches with more connections, it has more complex switches with fewer connections, which can increase the scalability due to having the need of fewer overlapping connections. (Shown on Fig. 2.5);

Star: consists in having every tile connected to a central router/switch responsible for forwarding the packet to the correct destination.

Table 2.1 contains information about the key aspects mentioned for each one of the topologies presented before.
Table 2.1: Network comparison for 1024 cores.

<table>
<thead>
<tr>
<th>Network</th>
<th>Diameter (inter-router hops)</th>
<th>Average Distance (hops)</th>
<th>Bisection Width (links)</th>
<th>Link Cost (links)</th>
<th>Router Degree (ports)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D Mesh ($2^5\times2^5$)</td>
<td>62</td>
<td>21.33</td>
<td>32</td>
<td>1984</td>
<td>4</td>
</tr>
<tr>
<td>2D Torus ($2^5\times2^5$)</td>
<td>32</td>
<td>16</td>
<td>64</td>
<td>2048</td>
<td>4</td>
</tr>
<tr>
<td>32-ary 2-flat Flattened Butterfly</td>
<td>1</td>
<td>1</td>
<td>512</td>
<td>1.5K</td>
<td>63</td>
</tr>
<tr>
<td>8-ary 16-fly Butterfly</td>
<td>15</td>
<td>15</td>
<td>512</td>
<td>17K</td>
<td>16</td>
</tr>
<tr>
<td>Fat Tree (32 nodes/leaf router)</td>
<td>11</td>
<td>9.07</td>
<td>1024</td>
<td>6144</td>
<td>33</td>
</tr>
<tr>
<td>Hypercube HC($2^{10}$)</td>
<td>10</td>
<td>5</td>
<td>512</td>
<td>5120</td>
<td>10</td>
</tr>
<tr>
<td>Star HC($2^{10}$)</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>HR3($N2=2^4$, $N1=2^4$, $N0=2^4$)</td>
<td>28</td>
<td>14</td>
<td>4</td>
<td>1154</td>
<td>4</td>
</tr>
<tr>
<td>Spidergon SG($2^{11\times x}$)</td>
<td>256</td>
<td>128.62</td>
<td>514</td>
<td>1536</td>
<td>3</td>
</tr>
<tr>
<td>SD($N1=2^3$, $N0=2^6$)</td>
<td>24</td>
<td>12.89</td>
<td>128</td>
<td>2056</td>
<td>5</td>
</tr>
<tr>
<td>4xconcentrated SD(16,16)</td>
<td>12</td>
<td>6.87</td>
<td>32</td>
<td>640</td>
<td>9</td>
</tr>
</tbody>
</table>

From the analysis of Table 2.1 the Spidergon-Donut (SD) [9] stands out. This network increases the router degree to five and lowers the diameter further, in relation to the Hyper Ring (HR). There are multiple possible SD configurations with various values of N1 and N0. The lowest SD diameter is achieved when it has $N1 \leq N0$, i.e., when the donut is fatter and has a smaller perimeter. With an increasing number of nodes, the SD’s diameter scales better than other popular networks.

2.1.2 3D Interconnection Networks

3D interconnection networks [5] improve upon the usual electrical networks, by allowing multiple layers of electrical interconnection networks to be stacked on top of each other. This capability is the stepping stone for the creation of a new set of interconnection networks that can increase performance by 46%, and decrease the power consumption by 62% when compared with a 2D mesh. One of its key aspects is the general reduction of the minimum latency, achieved by the reduction of the number of hops per packet and the length of the communication channels.
The research on this technology is at a very early stage [17], [18], [5], [19], but it has been already made clear that it supports and enhances the previously mentioned networks. This is due to it using the same base technology, but also allowing the existence of multiple layers of networks, which can be themselves connected in multiple fashions.

Currently there are a very limited set of interconnection networks studied for application on many-core processors, Figure 2.6 shows four of them. The 2-D IC-2-D NoC is equivalent to the 2D mesh discussed in the previous section, taken as the baseline for comparison purposes. The other three are more interesting examples. The 2-D IC-3-D NoC increases the router degree from four to five, as two 2D meshes are stacked on top of each other. On the other hand 3-D IC-2-D NoC maintains the NoC structure but has multiple Processing Element (PE), which can be cores or other devices in a SoC. Finally, the 3-D IC–3-D NoC has multiple PE in a multilayered mesh.

From a theoretical evaluation of the presented networks, it is possible to state that 3-D IC–2-D NoC constitutes the best option for networks with thousands of cores, because it uses less wires and routers while still retaining computational power, allowing an decrease in power consumption and in heat generation.

To conclude, this technology is still in a very premature stage, lacking further testing and analysis before it can be considered as a true alternative to the well known electrical networks. Nevertheless, it provides a new set of possibilities that, if confirmed, may make the transition to thousand core processors simpler, because the manufacturing technology in place today is very similar.

Furthermore, it is stated that “this paradigm enables the integration of Complementary Metal-Oxide-Semiconductor (CMOS) circuits with disparate technologies which can be non-silicon or even electro-mechanical” [18]. This leaves open the possibility of further modification, and improvement of the processors, possibly making it possible scaling this to multiple thousands of cores.

2.1.3 On-chip Optical Nanophotonic Networks

There is an entire new possible set of networks based on advances in electronic-photonic integration technology, making possible on-chip optical nanophotonic networks with greater integration,
smaller distances, and higher bandwidths \[20, 21, 22, 23\]. Furthermore, research \[24\] has proved the viability of building the needed processor optical devices with standard CMOS processes allowing the replacement of wires and buses \[23\].

The networks presented in the electrical networks section are implementable with this technology, however the existence of overlapping optical buses in a chip is still hard to achieve, making most the previous networks a poor choice when using this technology.

An interconnection network that successfully uses this new technology is the ATAC \[6\], Fig. 2.7. This includes an on-chip optical broadcast communication network in a mesh based tiled multicore improving the performance, energy consumption, and ease of programmability \[9, 22, 6\]. It gathers the tiles firstly in hubs, and then connects them with a 2D electrical mesh (EMesh) for intra-hub tile communication and a broadcast network (BNet) that is used to handle all the communication from and to the tiles. The BNet connects each hub to the On-chip Optical Broadcast Communication Network (ONet), forming all of this the ANet, here the main advantage of this topology is made clear. The ONet leverages the on-chip optical advances that eliminate all the inter-hub contention by using Wavelength Division Multiplexing (WDM). This allows a single optical waveguide to simultaneously carry multiple independent signals on different wavelengths, thus allowing concurrent hub communication with each other or with the memory controller rendering obsolete all the contention problems that exist on conventional buses.

Also this provides higher communication speed than standard buses, mainly due to the underlying physical speed of the photons being much higher than the electrons, has a lower energy consumption that an equivalent bus based solution, and the manufacturing problems that affect the on-chip optical devices may no longer pose a problem because standard processes have been developed to build this devices \[6\], allowing the replacement of today’s electrical bus to on-chip optical communication networks.

ATAC presents itself as the proof of concept for this new technology. Although it is still an undergoing topic of research, it already was shown that it can be manufactured using current techniques. So, ATAC, and this technology, must be considered as an true alternative to the electrical networks.
2.2 Cache Coherence

When dealing with multiple cores, containing individual caches, the problem of cache coherence emerges. In order to produce the correct results it must be guaranteed that every core is able to read the last value written to any memory position. This problem is already addressed in current single and multiple core processors, with strategies also valid for many-core processors. However, there are multiple variations of cache protocols that assure this, with different performances in many-core processors, that are analysed next.

To provide an adequate cache coherence scheme the interconnection network must also be taken into account. So, for the scope of this section the interconnection network to be considered, unless explicitly stated in the text, is a 2D mesh due to it being one of the most well known [9], providing a good mean of comparison.

This section gives an overview of the existing cache coherence protocols, and what is their viability on many-core processors, divided in: Snooping, and Directory.

2.2.1 Snooping

The snooping protocol [7] consists in having each core monitoring a bus shared by all the cores. It carries the information about each read and write of all the cores. So, using a Write Invalidate protocol, a core will invalidate a cache line when a write operation is observed to the memory location that the cache has a copy of. For example, if two cores $C_1$ and $C_2$ have a cached copy of a memory position $M_1$, if $C_1$ writes to memory a value on $M_1$, $C_2$ will invalidate its cached copy, assuring that when it is accessed an access to main memory has to be made, to retrieve the latest value. If a Write Update protocol is being used, when a cache value is updated, its new value is multicasted to all other cores, allowing them to update their local value.

Both of these protocols are part of the possible solutions, for today’s common two and four cores multicore processors. These have the advantage of being straightforward solutions that provide a good performance. Nevertheless, they have the major disadvantage that they do not scale properly. To implement them a shared bus must be guaranteed between all the cores, so that the operations made by each one can be snooped by the other ones. This is manageable between a few cores but it does not work for hundreds or thousands of cores.

So, it can be concluded that the use of a snooping based cache coherence protocol is not viable for many-core processors.

2.2.2 Directory

The directory based cache coherence protocol [7] consists in having a shared memory that explicitly controls the coherence between the caches. It acts as a filter so that a core must ask for its permission to load data from a memory position to its cache. The directory receives information about each cache copy, updating or invalidating cache lines accordingly.

This bypasses the problem of the snooping protocol that requires all the cores to share a bus.
So, despite its performance penalties, it constitutes the best solution for many-core processors that is currently available.

To implement this directory, on a processor, it has to be either one single memory block, or a distributed memory block. The former has the advantage of being a simple solution. It however creates a bottleneck to the processor performance, due to the fact that all the cores have to communicate with it. Also, it is built based either in on-chip Static Random-Access Memory (SRAM) or off chip Dynamic Random-Access Memory (DRAM) which imposes a slowdown to the caches. There is also the in-cache directory, Fig. [2.2] It takes advantage of the tile structure on many-core processors, by using the L2 shared caches containing a sub-set of the directory entries, leveraging the cache access speed and not imposing an additional memory block.

Also, an interesting solution that also takes advantage of caches is Non-Uniform Cache Architecture (NUCA) [7], that uses all the L2 caches as a global cache, accessible by any core. There are two options regarding the data placement techniques: static and dynamic. When a static data placement is used, the L2 cache where the cache line is going to be placed, “home core”, depends on the lower-order bits of the accessed data. This has the advantage of providing an easy way to find the “home core” and of allowing parallel access to cache lines in different L2 caches. However due to the data necessities in executions being inherently dynamic, it may incur in high remote-cache access rate and introduce many indirect coherence messages.

In case a dynamic data placement is used, if a certain cache block is used more often by a tile it will be mapped to its L2 cache, including a directory line, reducing the time needed to access it. Also, this makes it possible to leverage data locality between tiles [25]. However it imposes difficulties on the determination of the “home core”, and it needs a Operating System (OS)-assisted data placement algorithms in order for it to work properly.

In any case, the use of a directory based cache coherence protocol has the advantage of reducing the bus contention, making it more suitable solution for thousand core processors, but it still presents disadvantages [7]:

1. The automatic data replication of shared data, when a distributed directory is being used, results in one address being stored in multiple core caches, reducing the cache capacity left for other data, therefore increasing the cache miss rate;

2. The directory causes an indirection which leads to an increase to the cache miss access latency. In a typical directory-based protocol, consulting the directory of a line on the home node is on the critical path for a number of situations;

3. A write to shared data or an eviction of on-chip directory cache requires invalidation of all shared copies of the data, resulting in higher cache miss rates and protocol latencies;

4. The memory overhead introduced by the directory structure may not scale gracefully with the number of cores.

There are multiple directory based cache coherence protocols that try to minimise the previous
disadvantages: Fullmap directories, Duplicate-tag-based directories, Sparse directories, In-cache directories, and ACKwise, as described next.

Fullmap directories store the state of cache lines of all cores of the corresponding cores, with a bit vector that states in which caches there is a copy of that memory position, allowing every cache to hold a copy of the memory position [26]. This has the disadvantage of imposing a heavy communication traffic because, if a cache line is shared among all the caches a single update to main memory can cause an update to every cache. Another aspect is that this scheme does not explore data locality, introducing too much area cost due to redundant information.

Furthermore, it does not scale gracefully because if a bit per core is used on a 1024 core many-core processor, each directory line will at least have 128 Bytes which becomes unmanageable if the directory needs to handle a high number of cache lines. The key advantage of this protocol is that it allows that any number of cores share the same cache line, capability that is not present on all the protocols.

Sparse directories [7], as shown in Fig. 2.8, improves upon the Fullmap directory by reducing the space of directory storage needed, and by reducing the directory associativity. Fig. 2.8 depicts the behaviour of sparse directory. When it overflows the eviction of cached blocks is forced, because they cannot be any longer tracked by the directory.

In-cache directory [7], also known as static cache bank directory, takes a different approach at the directory by distributing by all the tiles the directory, which avoids a separate directory either in DRAM or on-chip SRAM. It saves directory tag storage, but over-provisions the sharer storage, due to the number of tags in the lower-level cache is greater than the number of tracked blocks in the private caches [26], [14], [15], [7].

Unfortunately this directory scheme also does not scale well. If a bit per sharer is chosen to keep track of the shared cache blocks it will become dominated by vector storage when the number of tiles is greater than 128. If a 256 core configuration is used more than 256 MB of on-chip memory storage would be consumed on the L1 directory storage, due to the over-provision.

ACKwise [6], Figure 2.9, is a cache coherence protocol created to leverage the advantages of the
ATAC interconnection network (described in previous section). This is based on the MOESI directory protocol \[16\]. Figure 2.9 presents the structure of a directory cache line. It contains three fields: State, specifying the state of the cached blocks associated with the directory line; Global(G), specifies if there are more block sharers than the vector capability; Sharer\(_1\)-Sharer\(_k\), specifies which tiles are sharing the cache block, although not existing one bit per tile to save space.

There are two possibilities when implementing this in hardware, to provide a sharer bit for each tile, or limiting the number of sharers on the vector to save space, being the last named ACKwisek. If the Global bit is set any modification on the referenced cache block on the directory must be broadcast to every tile, even if it does not have that cache block, which has a low overhead due to the ATAC’s ONet. Furthermore, this protocol uses a static data placement scheme.

Finally, all the previous protocols exist with multiple variants. For example a fullmap directory can be used with a MSI or a MOESI protocol.

\section*{2.3 Memory Architecture}

The predominant memory architecture in use in today’s multicore processors is the shared-memory architecture, or Uniform Memory Access (UMA). It consists on the system having a large set of contiguous data that is accessible by every element in the system. On the common multicore systems the accesses made to the memory have guarantees of coherence and consistency given by the cache coherence mechanism put in place, as described in Section 2.2.

As discussed before, when dealing with many-core processors there are issues with keeping the caches coherent and reducing the access time to memory. So, the standard shared memory architecture may fall short to provide the best performance for this new paradigm. This stems from the cost of maintaining the caches coherent.

If the cache coherence is not longer a requirement there are two immediate drawbacks: the cache coherence now has to be controlled explicitly, and programs that were developed to execute with shared memory have to be adapted.

Alternative solutions to shared memory are Non-Uniform Memory Access (NUMA), Cache Coherence Non-Uniform Memory Access (ccNUMA), and Multicomputers. The NUMA or Distributed Shared Memory (DSM), architecture consists in having the a single address space accessible by all the cores, but it may not be all physically in same place. Therefore different memory positions may have very different access delays, factor that should be considered to improve performance.

A slightly different architecture from the one presented before is ccNUMA. Here the difference is that the architecture itself deals with the issue of cache coherence bypassing the main issue of not using a shared memory, and reducing the effort of adapting legacy programs to this new architecture. An example of a ccNUMA architecture already in place on a processor is the Advanced Micro Devices (AMD) Opteron processor. Also, this architecture can be used on clusters, or grids.

Finally, there is the Multicomputers approach that consists on treating each core as an independent computer, with its own private address space. Despite each core having its own private address
space, nothing stands in the way of compromising and including a segment of shared memory across all cores. Also, if this latter solution is not used, all data sharing as to be performed through the use of message requests using the interconnection network. A example of an solution that includes this architecture is the Threaded Many-core Memory Model [27].

The solutions that have already been proposed for many-core processors, the use of an interconnection network and a directory, already enforce the leap from the shared memory architecture to the ccNUMA architecture. But, for the work presented ahead, the ccNUMA architecture uses an address space physically on the same space, i.e., without inherently different access delays. So, the assumptions taken to develop applications for the shared memory architecture can be kept when developing for this new architecture, simplifying the adaptation process.

2.4 Parallel Programming

Traditionally, application development follows the sequential programming model, that works well on single core processors. However, as discussed before, multicores are here to stay and the many-core processors are predicted to the end of the decade, so in fact it can be said that “sequential programming is dead” [28], [29]. A new programming model is needed together with abstraction of the computer system architecture [30], that can harvest the power of the current multicore and of future many-core processors.

When parallelizing an application there are two main approaches: parallel programming [31] and auto-parallelization. The first puts the work of parallelizing the program entirely on the programmer, while the second does it automatically. The latter may seems to be the perfect solution but it can only make a very limited set of parallelizations at compile time due to most of them being hard to identify. Furthermore, Instruction Level Parallelism (ILP) is always performed, if the processor supports it.

The parallelism in programs can be at the level of task, data, recursive and pipeline [32], [33], [30]. Multiple algorithms for parallel execution [34], [35], designs of parallel programs [30], [36], [37], [38], and different patterns for exploiting parallelism [32], [30] exist.

Nowadays there are two de facto standards when discussing parallelization in shared and distributed systems: OpenMP and Message Passing Interface (MPI), respectively. In this sense the many-core processors can use OpenMP to parallelize programs, but in many-core SoC there is a different problem. In these processors usually a GPU is integrated in the chip, being it usually more powerful than the CPU. To harness the power of the GPU a new programming model has been brought up the Heterogeneous Parallel Programming (HPP) model.

The focus on the rest of the section will be on shared memory, due to its use in the many-core processors. There are two main parallel programming models for shared memory systems: Pthreads and OpenMP.

Pthreads, or Portable Operating System Interface (POSIX) Threads, form a set of types and procedure calls for C/C++ programming language [31], [39], [40]. These allow the programmer to create, destroy, and manipulate threads, a lightweight process with a program counter and execution
stack [33]. It also provides mutual exclusion (mutex) and semaphores [41], allowing the handling of shared data. This model is a poor choice because the programs constructed with it are hard to develop, maintain, and very error prone. This causes the appearance of races, deadlocks, and starvation of threads. Another important aspect of this model is that it does not distribute the work by all the available tiles, it just creates the threads and waits for them to be executed, leading to a poor load balancing. Due to all this, this model is not a good solution for developing parallel applications, not allowing the applications to scale properly [42].

OpenMP [43] is an application programming interface to make it easier to develop parallel shared memory programs, portable across multiple architectures and available for Fortran, C/C++. It consists in taking a sequential program and placing pragmas (code annotations) on it. It can handle the creation of threads, synchronisation and the management of the shared memory. It is particularly easy to parallelize loops, one of the main sources of parallel code. These characteristics combined with the high abstraction-level provided make OpenMP the best programming model for shared memory systems.

On the many-core SoCs the answer is not the same as in the many-core CMPs. While they can execute programs parallelized with OpenMP and achieve good speedups, the fact of a GPU being present in the same chip as the CPU is not taken into account. This CPU+GPU or Accelerated Processing Units (APU) may be better used if a both of the available processors are used. Using the GPU to perform general processing, also named GPGPU (General Purpose Graphics Processing Unit), has the most important contribute done by NVIDIA with its CUDA (Compute Unified Device Architecture). To use them both there are programming languages as Brook [44] or Cg [45], and more recently a standard has been developed, the Open Computing Language (OpenCL) [46], [47] standard for parallel programming CPUs, GPUs and other processors. It provides APIs for C/C++.

So due to the fact that the work is going to be done over shared memory many-core CMP the obvious choice of parallel programming model is OpenMP. The future processors are most likely to be SoC but that does not make the results obtained with OpenMP unreliable. The OpenMP parallel programming is still applicable in the parallelization of the programs on the CPU of an SoC.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Types of Simulators</td>
<td>20</td>
</tr>
<tr>
<td>3.2 Emulation vs Instrumentation</td>
<td>21</td>
</tr>
<tr>
<td>3.3 Full-system Simulation vs User-level Simulation</td>
<td>22</td>
</tr>
<tr>
<td>3.4 Simulator Comparison</td>
<td>22</td>
</tr>
<tr>
<td>3.5 Simulator Comparison Conclusion</td>
<td>25</td>
</tr>
<tr>
<td>3.6 Result Validation</td>
<td>27</td>
</tr>
<tr>
<td>3.7 Summary</td>
<td>29</td>
</tr>
</tbody>
</table>
The use of simulators as a mean of experimentation and test of future possible architectures is very useful. It provides a simple means of test and allows the execution of as many simulations, of different configurations, as needed. However their performance is a bottleneck, limiting both the depth and scope of the investigation work that can be performed.

The performance of the simulators is specially affected when simulating extremely parallel architectures, with thousands of cores, most not even executing in parallel [48], [49], [50], [51], [52], and the ones that do still have to multiplex large amounts of simulated cores in the host cores.

Getting the best possible performance out of a simulator is the key topic of this chapter. The multiple options that can be taken when building a simulator are explained and analysed, culminating with the choice of a simulator to perform the test of multiple many-core architectures.

Ideally, a simulator should satisfy four desirable properties: it should be accurate, fast, execute a wide range of workloads, and be easy to use and modify [53]. This definition of a simulator is very hard to fully comply with, existing different trade-offs that simulators implement in multiple ways to enhance one property in the loss of another.

This chapter is divided in the most relevant aspects of simulators: Type of Simulator, Emulation vs Instrumentation, and Full-system Simulation vs User-level Simulation.

### 3.1 Types of Simulators

There are two main types of simulators: Field-Programmable Gate Array (FPGA) and Parallel Discrete Event Simulation (PDES).

FPGA simulators execute simulations on a circuit board, in which it is possible to define a virtual hardware scheme through the Hardware Description Language (HDL). The FPGA frameworks have as their main advantage providing high performance and precision. However they require a significant amount of effort to develop an test, since this type of simulator implies using expensive complex circuit boards, and require slow FPGA compilation toolchains that complicate the process of architectural exploration. So, this makes it a poor choice. Next only PDES simulators are discussed.

PDES simulators are based on the Discrete Event Simulation (DES) that execute simulations in a sequential way, by manipulating code that is going to be executed on a regular processor. This has as its main advantage being costless and easy to use, using a regular computer, with no need of any special hardware. However, it is usually slow due to the their poor performance. In this work the simulator will have to be a PDES simulator, because it provides the same functionality as on a FPGA simulator at no cost. With this said is important to understand how PDES works.

DES typically uses three data structures: the state variables, describing the system state; an event list containing all the scheduled events that have not been executed yet; and a variable representing the global clock, representing the progress of the simulation. Each event represents a change in the state of the system, such as a memory access, with a timestamp associated with it. The simulator removes the event with the lowest timestamp from the event list and processes it, by executing some simulator code to perform the correct system state change and scheduling any causality events. It
is crucial that the selected event has the lower timestamp ($E_{min}$) of the event list. If any other event is executed it may modify state variables used by $E_{min}$ causing a causality error because a future event affected a past event. For example, consider two events $E_1$ and $E_2$, with timestamp $T_1$ and $T_2$ respectively, being $T_1$ less than $T_2$. If $E_1$ writes on a state variable that is read by $E_2$, so $E_1$ must be executed before $E_2$ to assure that no causality error occurs.

The major parallelization opportunity comes from processing events concurrently in multiple cores or processors. However a direct mapping of this paradigm on a shared memory multicore processor faces some difficulties. Certain constraints must be taken to assure a functionally correct execution. To assure this most PDES simulators use a process oriented methodology that forbids processes to access shared state variables directly (with some exceptions [30], [50], [51]), being the modelled system seen as a set of processes that interact over the simulated time.

The simulator uses a series of logical processes $LP_0$, $LP_1$, ... , one for each physical (simulated) process, with all the interactions between the physical processes modelled by timestamped events sent between the corresponding processes. Each logical process has a portion of the corresponding state of the modelled physical process, namely the local clock that characterises the process progress. All the simulation methods discussed here use this logical process paradigm.

It is possible to guarantee that no causality errors occur if the “Local Causality Constraint” [24] is respected. This states that “A discreet event simulation, consisting of Logical Processes (LPs) that interact exclusively through the exchange of timestamped messages, obeys to the Local Causality Constraint if and only if each LPs process the events in a non-decreasing order of timestamps.” [24]

This constraint causes poor performance on PDES simulators, because to respect this one of two options must be taken, choose an optimistic or a pessimistic solution. The former consists on executing the program freely, and if any causality error occurs the simulator executes a “roll back” of what went wrong maintaining the functional correction. The latter consists on checking if any events will cause a causality error before it executes, harming the scalability because this requires constant requests for synchronisation.

\subsection*{3.2 Emulation vs Instrumentation}

A key aspect of the performance of a simulator is if it simulates the architecture using emulation (virtualization) or instrumentation.

The former consists in decoding one instruction at a time and simulating it, using functional and temporal models on the simulator to do so. This approach has the advantage of allowing the simulation of an architecture, different from the one in which the simulator is being executed, removing limitations of what is the Instruction Set Architecture (ISA) of the simulated architecture. However this approach implies a high and constant overhead throughout the simulation, requiring the execution of a high number of instructions on the host machine to simulate one instruction of the simulated architecture.

The latter consists in instrumenting the executed binary, executing the simulated program directly.
on the present architecture. This allows the simulator to be updated with information about the execution of the binary and to allow the simulator to maintain control of the binary. This approach has the advantage not needing a functional model, which can be hard for complex ISAs, and having a higher potential performance than the previous one, still dependent on the temporal model. As a disadvantage, this approach needs that the binary to simulate is available on the architecture of the host machine in which the simulation is going to be executed, which can be an inconvenient.

### 3.3 Full-system Simulation vs User-level Simulation

An aspect that defines the scope of the type of workloads that are possible to be executed is the type of simulation that the simulator offers, Full-system or User-level simulation [54].

A full-system simulation allows an operative system to be present, providing privileged modes and devices, allowing the execution of more complex multi-threaded or multi-process workloads, and of applications that need network or I/O.

A user-level simulation allows the execution of simpler workloads that only involve single-process execution. This has the advantage of being much simpler to develop and much faster, because the simulators that follow this type of simulation do not need to provide emulation mechanisms for operative systems, disk images, network cards, among other possible computer hardware.

### 3.4 Simulator Comparison

This section provides a summarised description of the most relevant multicore simulators, and their main aspects. The key characteristics that have to be payed attention to are: the simulation Engine; how the parallelization of the simulator is made; if it provides detailed core and uncore (devices tightly coupled with the normal functioning of the core) simulation; if it provides full-system or user-level simulation; if it supports multiprocess applications; if it supports apps run on virtual machines (managed applications, and if it can distribute the simulation across multiple hosts (multi-host).

The following simulators are analysed: gem5, CMPSim, Graphite, Sniper, Hornet, SlackSIM, and ZSim.

#### 3.4.1 gem5

The gem5 [55] is an emulation-based, user-level and full-system, PDES. This simulator was originated by two previous simulators the M5 [56] and GEMS [57]. The former provided an “extremely configurable framework, multiple ISA and multiples CPU models” [55], and the latter complements “this capacities with a detailed and flexible memory model, including support for multiple cache coherency protocols and interconnection models” [55].

This simulator can execute complex workloads, supports multiple ISAs, can load a Linux based operating system in ARM, ALPHA and x86 architectures, and it was developed with the collaboration of multiple companies and universities. However it has a very poor performance.
3.4.2 CMPSim

CMPSim \cite{58} is an instrumentation based, user-level, PDES simulator. This simulator uses “pin to collect the memory addresses of multithreaded and multiprocessor programs to analyse program behaviour. It gives detail about cache line, miss rate and coherence traffic and helps in predicting future performance of an application.” \cite{23}. Its main advantage is having a acceptable performance but it is focused exclusively on cache performance, not being a complete simulator.

3.4.3 Graphite

Graphite \cite{59} is an instrumentation based, user-level, PDES simulator. The purpose of this simulator is to simulate multicores with hundreds or thousands of cores using one or more hosts, one of its strong points. The instrumentation is carried out with Intel’s Pin accordingly to a Dynamic Binary Translation (DBT) scheme, it consist in translating the instructions as they are executed. However the scalability of the simulator is close to linear when performing the synchronisation with slack, that does not compromise the functional correction but adds a small error to the accuracy of the simulation. The accuracy of the simulation is affected by events out of order, that cause the temporal model to calculate the simulated time incorrectly. Another strong point is the support for OpenMP and standard pthread, although limited.

The network models available of the electrical mesh type are: hop counter, analytical, and hop by hop. Hop Counter consists in sending the message directly from one tile to another without any control, which increases the performance but decreases the accuracy; the analytical also sends directly the message from one tile to another but has a global contention model that has a high performance, while providing medium accuracy; hop by hop, consists in a message that is sent form one tile to another stopping at intermediate routers, allowing a per link contention model which reduces the performance but has a very high precision.

All the previous models assume an infinite output buffering, which is not a real situation in a processor, so to overcome this problem queue theory models are used to account for the delay that comes from buffers being full. Yet it has been proven inaccurate \cite{41}.

3.4.4 Sniper

Sniper \cite{60} is an instrumentation based, user-level, PDES This simulator was built upon Graphite but it implements interval simulation. This type of simulation brings advantages to the precision of the simulation reducing the error on a single thread simulation, from 114% to 19.8%, and on a multi-thread simulation from 59.3% to 23.8%. Another import aspect is that Sniper is optimised for mono-host execution, not being fully disclosed which are the possibilities for multi-host execution, and in what way the obtained results about the precision and the performance are reliable.
3.4.5 Hornet

Hornet [61] is an instrumentation based, user-level, PDES. This multicore simulator is directed at simulations with thousands of cores. One of its main characteristics is the ability to offer cycle-accuracy, which guarantees full precision, and is still able to be executed in a more slacked manner through the use of periodic synchronisations that preserve the functional correctness while increasing the simulation performance, but decreasing the precision. It is possible to set the connections between the cores, enabling the implementation of rings, multi-layer meshes or the simpler 2D Mesh. The routing mechanism can also be set to oblivious, static or adaptive routing.

It uses a MIPS (Microprocessor without Interlocked Pipeline Stages) simulator which forces to recompile and adapt programs to execute on this simulator, and can be a disadvantage. On the other hand, this can be useful to perform simulations if the goal is to test possible future processors that use MIPS processors instead of the more common x86 or x86_64 ISAs.

3.4.6 SlackSIM

Slacksim [62] is an instrumentation based, user-level, PDES. This simulator is based on the older SimpleScalar [63] simulator, modified to allow the execution of any core in any number of separate threads and the support to the Intel Netburst [64] micro-architecture. The main characteristic of this simulator is the fact that it uses a slack, i.e the maximum difference between two cores of the point in simulation in which they are, to gain simulation performance. Also, it offers full support to the POSIX thread programming model.

SlackSim simulations are executed based on a Simulation Manager Thread, and multiple Core Threads. The former has a function to command and define the pace of the simulation and to simulate the low level cache hierarchy present on the simulator, the L2 shared cache and its interconnections to the cores. As for the latter it simulates a single core with its L1 private cache.

One conclusion from analysing this simulator architecture is that the Simulation Thread Manager is a major simulation bottleneck, because all other threads depend on its answers to proceed with their execution. However previous work [65] has revealed that in average this thread has less work than the other threads, which reduces wait times for an answer that blocks the execution of a Core Thread. Nevertheless in case this becomes a bottleneck a hierarchical approach can be taken.

3.4.7 ZSim

ZSim [54] is an instrumentation based, user-level, PDES simulator. This simulator is directed at the fast simulation of processors with hundreds and thousands of cores, using three techniques to achieve this goal.

The first technique used is the use of Dynamic Binary Translation performed trough the use of Intel's pin, that translates binary code according to its needs, taking advantage of this process to introduce instrumentation on the executed binary that performs most of the temporal measurements, removing overheads of the call to an external simulator function to process the measurements, as it
is performed in most traditional cycle-driven or event-driven simulators.

The second technique is the utilisation of the Bound-weave algorithm. This algorithm performs synchronisation in two different stages, having as result an improvement on the scalability of the simulations on single host executions while maintain precision. The intuition behind this algorithm is that in small intervals of the hundred thousand cycles the interactions between the cores with shared resources rarely affect the access components. With this in mind the simulation in split in two phases, on the first phase (bound) the interval is executed without contention and with zero-load values for all the memory accesses. Due to all the memory access already being accounted for the synchronisation is adapted allowing a much faster execution than convention PDES simulators techniques.

The third technique consists in having implemented some virtualisation techniques on the simulator that, although being a user-level execution, allows that certain complex workload only supported by full-system simulators to be executed. An example of this are multi-program and managed runtime applications. This has the advantage of increasing the scope of workloads possible to be tested, while still avoiding the problems of operating systems and Instruction Set Architectures that support thousands of cores.

3.5 Simulator Comparison Conclusion

Having the information about the most well-known multicore simulators, they are compared with the goal of having the best performance possible. When the tests are executed the experimental system will be described in detail, but in the scope of the simulators is only important to state that the simulator can be executed in a network of quad-core computers or in one of these machines.

Table 3.1 provides a comparison of the analysed simulators.

On the Engine line it is identified in what way the binary code is executed by the simulator. DBT (Dynamic Binary Translation) usually the best option because it provides higher performance than Emulation. So in this criteria the CMPSIM, Graphite, Sniper and ZSim simulators are preferable.

On the Parallelization line it is provided the name of the technology used to manage the simulation. The most promising approach is the one implemented by ZSim, the Bound-Weave, because it increases the performance without decreasing the precision of the simulation. SlackSIM implements an approximation of PDES approach optimistic(o) and pessimistic(p) according to the necessities. Graphite and Sniper support Limited Skew. Sniper only stands out for also supporting intervals simulation which has a good increase of performance. So in this aspect the simulators that stand out are ZSim, SlackSIM and Sniper.

On the Detailed Core line are identified the simulators that support out of order execution (OOO), a characteristic necessary to simulate more complex modern processors that can execute instructions ahead, while waiting for the end of a memory access. In this aspect the simulators that stand out are Gem5, Sniper and ZSim.

In the Detailed Uncore line are identified the simulators that can include in their simulation process other elements exterior to the core, that can be essential to the core performance, as for example a
Table 3.1: Simulator comparison.

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Gem5</th>
<th>CMPSim</th>
<th>Graphite</th>
<th>Sniper</th>
<th>HORNET</th>
<th>SlackSim</th>
<th>ZSim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engine</td>
<td>Emulation</td>
<td>DBT</td>
<td>DBT</td>
<td>DBT</td>
<td>Emulation</td>
<td>Emulation</td>
<td>DBT</td>
</tr>
<tr>
<td>Parallelization</td>
<td>Sequential</td>
<td>Limited Skew</td>
<td>Limited Skew</td>
<td>Limited Skew</td>
<td>PDES(p)</td>
<td>PDES</td>
<td>Bound-Weave</td>
</tr>
<tr>
<td>Detailed Core</td>
<td>OOO</td>
<td>No</td>
<td>No</td>
<td>Approx. OOO</td>
<td>No</td>
<td>OOO</td>
<td>DBT-based OOO</td>
</tr>
<tr>
<td>Detailed Un-core</td>
<td>Yes</td>
<td>MPKI only</td>
<td>Approx. contention</td>
<td>Approx. contention</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Full-System</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Multiprocess apps</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Trace-driven only</td>
<td>Trace-driven only</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Managed apps</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Multi-host</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

L3 cache or a memory controller. In this aspect the CMPSim only supports Misses per Kilo Instruction (MPKI). Graphite and Sniper can simulate contention in the access to the uncore elements through the use of theoretical queue models. Hornet, SlackSIM and ZSim support all the available uncore elements available nowadays. In this aspect these last are the most adequate because they simulate the real systems in a more correct manner.

The Full-System line indicates which simulators support this type of simulations. This capacity is accessory because this work has the objective of identifying which are the best ways of take advantage of the computational power of the multicores with thousands of cores, only having the simulator to be able to execute programs and extract measurements, not being this capacity limiting in a decisive manner. However this ability can be used to perform complex workloads, as a way of second proof, increasing the confidence of the obtained results. The table indicates that only gem5 supports this, being capable of supporting all the scope of Full-System simulation. ZSim, although it states that is a user-level simulator, supports some aspects of Full-System simulation which can be an advantage. In this aspect the simulators that stand out are gem5 and ZSim.

In the Multiprocess apps line are identified the simulators that support applications that can start, not only more than one simulated thread but more than one simulated process, which can be useful in some tests. In this aspect the simulators that stand out are gem5, CMPSim, Sniper, Hornet and ZSim. In the Managed Apps line are identified the simulators that support applications that are based upon virtual machines, as Java, Scala or Python. This is a characteristic of Full-System simulators. This possibility is very interesting because it gives the possibility of experimenting a great number of applications, not limiting the tests to applications that have to run naively on the machine (C, C++, etc). In this aspect the simulators that stand out are gem5 and ZSim.

Finally, in the Multi-host line are identified the simulators in which it is possible to distribute the simulation effort through a network of machines (grid, cluster, etc). Which is very interesting because the used test environment includes a network like this, reducing the necessary time to perform a simulation and enabling the execution tests with a larger scope. In this aspect the simulator that
stands out is Graphite, the only one that has this feature.

Concluding, the simulator that presents itself has the best option is the ZSim. However, despite multiple attempts, it was not made available by the authors. So, to perform the tests gem5 and Sniper will be used. The former provides more detailed results while the latter can perform simulations with a higher number of cores, but with less inaccurate results. This provides a more accurate study of the architecture, allowing double check of the results.

Furthermore, Graphite makes part of the choice due to its exclusive ability to use multiple hosts to execute a simulation.

### 3.6 Result Validation

The experimental work will consist in developing and testing architectures that have different levels of performance, with the goal of being possible to make a fair comparison between different architectures. The utilisation of a standard measure of performance used by the community will allow the check of the obtained results by others, making it easier to analyse and compare the obtained results.

Having these goals in mind, we present the benchmarking suites that the community uses, with a comparison performed at the end.

#### 3.6.1 SPLASH-2

SPLASH-2 [44] is one of the most used scientific benchmark suites for the study of multicore processors. It possesses a varied set of workloads, having five High-Performance Computing benchmarks (barnes, fft, lu, ocean, and water), three Graphics benchmarks (radiosity, raytrace, volrend), one Signal Processing benchmark (cholesky), and a General benchmark (radix).

One weak point of this benchmark suite is that the included workloads do not represent some emerging problems, and some instances may be considered outdated.

#### 3.6.2 PARSEC

PARSEC [44] (Princeton Application Repository for Shared-Memory Computers) is a multi-threaded benchmark suite developed by Intel and Princeton University, with the goal of providing a more adequate set of workloads to the study of new multicore processors. It possesses a varied set of workloads, with the key goal to provide a more broad and updated set of programs than SPLASH-2.

<table>
<thead>
<tr>
<th>Program</th>
<th>Application Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>Financial Analysis</td>
</tr>
<tr>
<td>swaptions</td>
<td></td>
</tr>
<tr>
<td>bodytrack</td>
<td>Computer Vision</td>
</tr>
<tr>
<td>canneal</td>
<td>Engineering</td>
</tr>
<tr>
<td>dedup</td>
<td>Enterprise Storage</td>
</tr>
<tr>
<td>ferret</td>
<td>Similarity Search</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>Animation</td>
</tr>
<tr>
<td>facesin</td>
<td></td>
</tr>
<tr>
<td>freqmine</td>
<td>Data Mining</td>
</tr>
<tr>
<td>streamcluster</td>
<td></td>
</tr>
<tr>
<td>vips, x264</td>
<td>Media Processing</td>
</tr>
</tbody>
</table>

---

27
Table 3.3: PARSEC’s workloads characteristics.

<table>
<thead>
<tr>
<th>Program</th>
<th>Parallelization</th>
<th>Granularity</th>
<th>Working Set</th>
<th>Data Usage</th>
<th>Maximum Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>data-parallel</td>
<td>coarse</td>
<td>small</td>
<td>low</td>
<td>65536</td>
</tr>
<tr>
<td>bodytrack</td>
<td>data-parallel</td>
<td>medium</td>
<td>medium</td>
<td>high</td>
<td>60</td>
</tr>
<tr>
<td>canneal</td>
<td>unstructured</td>
<td>fine</td>
<td>unbounded</td>
<td>high</td>
<td>2000000</td>
</tr>
<tr>
<td>dedup</td>
<td>pipeline</td>
<td>medium</td>
<td>unbounded</td>
<td>high</td>
<td>94130</td>
</tr>
<tr>
<td>facesim</td>
<td>data-parallel</td>
<td>coarse</td>
<td>large</td>
<td>low</td>
<td>80598</td>
</tr>
<tr>
<td>ferret</td>
<td>pipeline</td>
<td>medium</td>
<td>unbounded</td>
<td>high</td>
<td>256</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>data-parallel</td>
<td>fine</td>
<td>large</td>
<td>low</td>
<td>3000000</td>
</tr>
<tr>
<td>freqmine</td>
<td>data-parallel</td>
<td>medium</td>
<td>unbounded</td>
<td>high</td>
<td>91</td>
</tr>
<tr>
<td>raytrace</td>
<td>data-parallel</td>
<td>medium</td>
<td>unbounded</td>
<td>high</td>
<td>32400</td>
</tr>
<tr>
<td>streamcluster</td>
<td>data-parallel</td>
<td>medium</td>
<td>medium</td>
<td>low</td>
<td>16384</td>
</tr>
<tr>
<td>swaptions</td>
<td>data-parallel</td>
<td>coarse</td>
<td>medium</td>
<td>low</td>
<td>64</td>
</tr>
<tr>
<td>vips</td>
<td>data-parallel</td>
<td>coarse</td>
<td>medium</td>
<td>low</td>
<td>3612</td>
</tr>
<tr>
<td>x264</td>
<td>pipeline</td>
<td>coarse</td>
<td>medium</td>
<td>high</td>
<td>128</td>
</tr>
</tbody>
</table>

PARSEC has five main characteristics:

- Multithreaded applications: all the applications of PARSEC have been parallelized to take advantage of the shared memory multicore processors;

- Emerging workloads: the present workloads characterise more recent problems that require high computational power, being still an open problem how to optimise their execution;

- Diversification: PARSEC includes programs of multiple scopes and different utilisation models with the goal to replicate the way programs are used nowadays;

- Employ cutting edge techniques: the implemented workloads employ the last algorithms and techniques of their field;

- Support search: the benchmark suite supports search by supplying a structure that allows instrumentation and manipulation of the workloads giving a more detailed support to micro-architectural simulations.

The programs of the workloads of SPLASH-2 and PARSEC are fundamentally different, but PARSEC’s is more broad and updated which allows obtaining more reliable results. The benchmark suite PARSEC will be used.

So, with the choice of the benchmark suite made it is important to understand the workloads, this allowing a more detailed analysis of the results obtained latter on.

Table 3.3 specifies the key characteristics of PARSEC’s workloads [66].

From the different aspects presented on the table, the factor that will have the greatest impact on the performance of the applications is the data usage sharing and exchange. This is so because many-core processors have a lot of raw computational power, but have as a major drawback the delays involved in communication.

Next is presented the purpose of each workload:
• blackscholes: calculates the prices of portfolio of options using the Black-Scholes Partial Differential Equations;

• bodytrack: tracks a marker-less human body using as input the feed from four cameras;

• canneal: minimise the routing cost of a chip design with cache-aware simulation annealing;

• dedup: detects and eliminates redundancy in a data stream with a technique called “deduplication”;

• facesim: simulates motions of a human face for visualisation purposes;

• ferret: search engine which finds a set of images similar to a query image by analysing their contents;

• fluidanitemete: simulates the underlying physics of fluid motion for realtime animation purposes with smooth-particle hydrodynamic algorithm;

• freqmine: identifies frequently occurring patterns in a transactional database;

• raytrace: uses physical simulation for visualisation, generating an image of an object with a huge amount of triangles;

• streamcluster: computes an approximation for the optimal clustering of a stream of data points;

• swaptions: prices a portfolio of swaptions with the Heath-Jarrow-Morton framework;

• vips: applies a series of transformations to an image;

• x264: video encoding using the H.264 codec.

3.7 Summary

In this chapter, it was analysed how to perform tests on many-core processors as accurately as possible with the tools available today.

To perform tests with many-core processors a simulation platform has to be used. Multiple were analysed, and as result the gem5, Graphite and Sniper simulator were chosen. These are analysed in detail in the next chapter, also providing a comparison of their results.

Also, two benchmark suites were compared. Here the PARSEC benchmark suite presented itself as the best option, having been built from the ground up to access the performance of architectures with a high number of cores, while improving over the SPLASH-2 benchmark suite.
4

Simulator Evaluation

Contents

4.1 Simulators' Constraints .................................................. 31
4.2 Auto Tester ........................................................................ 34
4.3 Result Comparison .............................................................. 34
4.4 Conclusion ........................................................................ 47
The comparison of multiple systems with thousands of CPUs is made possible by the use of multicore simulators. There is an extensive set of this type of simulators, which was already discussed in Section 3, but the evaluation done does not provide sufficient proof of their results’ correctness.

In this section the previously chosen simulators are analysed, tested and cross examined, with the goal of increasing the confidence in their results. Furthermore, an initial evaluation of the simulators constraints, ease of use and capabilities is provided.

This chapter is divided in: Simulators’ Constraints, containing the description of the key constraints of the simulators; Auto Tester, a section describing the tool developed to streamline the testing procedure; Test Environment, the configurations of the system used to perform the simulations; Result Comparison, containing the comparison of the results of the different simulators; and Conclusion.

### 4.1 Simulators’ Constraints

The effective use of gem5, Sniper, and Graphite presents some limitations in the broadness and scope of the performed tests, fact that enforced a series of constraints. Next are described multiple characteristics that had an impact in the test execution procedure.

#### 4.1.1 Ease of Use

Working with these three simulators allowed the direct contact with the advantages, and disadvantages of the utilisation of either one of them.

In terms of ease of use Sniper stands out for multiple reasons: it is easy to configure, it ships with multiple tools that generate useful graphical outputs of multiple aspects of the simulation, provides sufficient documentation, and a helpful online forum.

Graphite is easy to configure, it provides some documentation, and also has an online forum.

As for gem5 it has an extensive set of documentation, some mailing lists, and multiple video tutorials. Nevertheless, it is quite complicated to work with. The key difficulty in using this simulator lays on the configuration of a system to simulate being spread among multiple configuration files. Also, simulating a specific architecture requires the compilation of gem5, unlike the other simulators. This combined with an extensive set of restrictions imposed by the interconnection network choice and by the cache coherence mechanism, without a clear configuration hierarchy defined, made gem5 difficult to use. Also, this made it hard to understand what architecture was actually created by the simulator.

#### 4.1.2 Architectural Capabilities

The architectures possible to define are not the same in all of the simulators. To compare simulations’ results across the simulators, it is crucial that the simulated architectures are as close as possible. Otherwise the obtained results will be very hard to compare.

In Table 4.1 the key architectural options available in the simulators are displayed.
Table 4.1: Architectural capabilities of gem5, Sniper, and Graphite.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>gem5</th>
<th>Sniper</th>
<th>Graphite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interconnection Networks</td>
<td>bus, 2D mesh, 2D torus, Pl2Pt, Crossbar</td>
<td>bus, 2D mesh, 2D torus</td>
<td>bus, 2D mesh</td>
</tr>
<tr>
<td>Cache Coherence</td>
<td>MI, MESI, MOESI</td>
<td>MSI, MESI, MESIF</td>
<td>MSI, MOSI</td>
</tr>
<tr>
<td>Cache Levels</td>
<td>unlimited</td>
<td>unlimited</td>
<td>2</td>
</tr>
<tr>
<td>Directory Location</td>
<td>LLC, DRAM</td>
<td>LLC(lag-dir), DRAM</td>
<td>DRAM</td>
</tr>
<tr>
<td>Directory Type</td>
<td>fullmap</td>
<td>fullmap, limited</td>
<td>fullmap, limited, ackwise</td>
</tr>
</tbody>
</table>
| CPU Type                | in-order, out-of-order                 | out-of-order                          | In-order Core
Out-of-Order
Memory

The capabilities of Graphite, and Sniper are similar due to the latter consisting of a spin-off of the former. gem5 has also a relatively similar set of capabilities.

This set of capabilities are the ones present on the official version of these simulators. gem5’s modularity allows for a series of extensions that can add capabilities, so this is not a closed set. Also, gem5 has a number of cross cutting constraints that limit the application of different options at the same time. For example, it is not possible to have all the available CPU types with every available interconnection network.

4.1.3 Parsec Support

Sniper and Graphite provide official support for the, previously selected, PARSEC benchmark suite. However this support is relative to two different versions. The former supports officially the 2.1 version of this benchmark, while the latter only supports the last 3.0 version. The difference in the versions is due to having been added support to network benchmarks, user-level parallel Transmission Control Protocol/Internet Protocol (TCP/IP) stack, SPLASH-2, SPLASH-2x enlarged inputs, and a complete redesign of the framework for supporting external suites. From this set of changes it is possible to conclude that the tests themselves have not suffered changes, so the test results expected should be the same in both versions, making the results of Sniper and Graphite comparable.

On Table 4.2 the listing of which of PARSEC’s workloads function correctly on each of the simulators, Sniper’s and Graphite’s support is almost total. As for gem5’s support is quite limited. The cause for the poor gem5 support arises from the fact that the simulator does not provide official support to the execution of this benchmark suite. The simulations where executed using disk images provided by the University of Texas at Austin [67], and the kernel image provided the gem5’s official website. A factor that also influenced negatively the execution of workloads on gem5 was that most of these workloads can only function in their less demanding mode, using the smallest input set, due to the fact that when the workload demands more resources the workload simulation crashes.

In order to execute it on gem5, the simulator must be on full-system simulation mode, this mode implies the presence of an emulated OS, hard drive, and other devices. The execution of the workloads...
Table 4.2: PARSEC Benchmark workloads that function on the simulators.

<table>
<thead>
<tr>
<th>Program</th>
<th>gem5</th>
<th>Sniper</th>
<th>Graphite</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>swaptions</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>bodytrack</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>canneal</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>dedup</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>ferret</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>facesim</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>freqmine</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>raytrace</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>streamcluster</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>vips</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>x264</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

‡ only with one core  
☆ only up to 16 cores  
† only up to 64 cores

and of the pervasive operations alongside of the [OS] alongside may contaminate the results obtained from the execution of the afore mentioned workloads. To minimise the impact on the workload execution results gem5 provides the “/sbin/m5 dumpresetstats” command inside the virtual machine. This, when it is executed, saves the simulation statics so far, allowing them to be analysed later if needed. But more importantly it resets the simulation statistics making the results obtained from this moment onwards almost solely depend on the benchmark workload executed, because now this workload is responsible for the vast majority of computation. So, in the simulations executed on gem5 the simulation statistics are always reset, using this mechanism, right before the benchmark workload starts. Some workloads were not possible to execute due to random crashes, unfortunately why they failed was not made clear by the simulators.

4.1.4 Host Execution Time

Alongside with all the previously mentioned constraints, there is a constraint that was the most troublesome in all the simulations performed: the host execution time. This is the time that the simulator takes to perform a simulation.

As it would be expected, this has a high degree of variability accordingly to the simulated architecture, especially due to the number of cores, and to the executed workload.

To give a sense of how much time each simulator takes, the following Table 4.3 has the execution times for the fluidanimate workload, one of PARSEC’s workloads that takes more time to execute.

So, a simulation can take from 24 minutes to about nine hours to execute. The high host execution times forced the number of simulations to be kept at the possible minimum. This was a major drawback for a proper comparison of the results of the simulators, see Section 4.3, and for an extensive exploration of different architectures, see Chapter 5. Nevertheless, a large number of simulations were still possible to perform.
It is possible to observe in Table 4.3 that the host execution time has different behaviours according to the simulator. On gem5 the host execution time increases as the number of cores grows because it has to simulate a more complex system while using only one core to perform the simulation. As for Graphite and Sniper they are capable of splitting the work by the available cores on the host machine, creating one thread per simulated core. So, as the host machine has four cores, Section 4.2 the host execution time decreases up to this value, but increases afterwards.

4.2 Auto Tester

An application named AutoTest was built to speed-up and streamline the testing procedure. It is composed by an automatic test generator, configurator, executor, and analyser. Written in Java, it supports both Graphite and Sniper test generation fully. As for gem5 it only supports the results analysis.

This is useful for the community that deals with systematic execution of tests in these simulators, so it was made available as an open-source project. It is possible to access it following the link: https://github.com/danieljorge/autotest-manycore.

The tests performed have the same underlying testing platform a Debian 6 GNU/Linux distribution, running on Virtual Box that was executed on top Fedora 19. As for the hardware it was composed by an Intel i7 quad-core CPU with sixteen gigabytes of RAM.

4.3 Result Comparison

Using three different simulators brings forward the issue of their validity. Although each one of them has been proved to have some degree of correction, it is still an open question the similarity of the results obtained with either one.

4.3.1 Compatible Simulator Settings

To assert the result similarity, the following methodology was abode to. Each PARSEC workload that is fully executable by gem5, i.e., the workloads on Table 4.2 and each handcrafted workload, would be also executed by Sniper and Graphite simulators. Also, a configuration that could be simulated in all three simulators would be used.

gem5 is the common denominator in the above mentioned methodology, due to the fact that it presents more limitations in terms of the possible workloads to execute, than the other two simulators.

Table 4.3: fluidanimate host execution time on gem5, Sniper, and Graphite in minutes.

<table>
<thead>
<tr>
<th>Core Number</th>
<th>gem5</th>
<th>Sniper</th>
<th>Graphite</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>186</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>208</td>
<td>22</td>
<td>21</td>
</tr>
<tr>
<td>132</td>
<td>50</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>531</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Nevertheless, it is possible to specify fairly similar configurations in all three simulators, specially in Graphite and Sniper due to the latter constituting an modification of the former (see Section 3).

The gem5 simulator imposes a very different challenge to achieve a fair comparison of the results. It imposes full-system simulation of the mentioned workloads, imposing the presence of a virtual hard drive, an OS, and other devices. Also, the configuration of the architecture is far more complex when compared to the other simulators. All this renders it very difficult to simulate the exact same architecture.

The key factors of the specified configuration for each simulator are depicted in Table 4.4.

Table 4.4: PARSEC benchmark workloads that function on the simulators.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>gem5</th>
<th>Sniper</th>
<th>Graphite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interconnect Network</td>
<td>2D mesh</td>
<td>2D mesh</td>
<td>2D mesh</td>
</tr>
<tr>
<td>Cache Coherence</td>
<td>MOESI</td>
<td>MSI</td>
<td>MSI</td>
</tr>
<tr>
<td>Cache Levels</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Directory Location</td>
<td>LLC</td>
<td>DRAM</td>
<td>DRAM</td>
</tr>
<tr>
<td>Directory Type</td>
<td>fullmap</td>
<td>fullmap</td>
<td>fullmap</td>
</tr>
<tr>
<td>CPU Type</td>
<td>in-order</td>
<td>out-of-order</td>
<td>in-order core out-of-order memory</td>
</tr>
</tbody>
</table>

Unfortunately it was not possible to achieve the exact same configuration in every simulated aspect, this is due to multiple constraints that the simulators impose.

Sniper and Graphite have a very similar configuration, only differing in the type of the CPU. Sniper is a derivative project of Graphite, having as its main changes the inclusion of a different, more accurate, simulation mode built upon intervals (also know as slack), and to be able to use this mode an out-of-order CPU must be modelled. Also, the in-order complex ICOCOM CPU type shipped with Sniper, equal to the one present in Graphite, does not work. So, it is likely to observe faster executions on both the host and simulated time in Sniper.

Sniper’s and Graphite’s configuration differ from gem5’s. This difference is due to gem5’s modularity, and to the built-in strictly fixed configurations. To achieve the obtained configuration first the closest type of cache coherence was chosen and here the issues began, no MSI scheme was available. The closest cache coherence mechanism available was MOESI due to it including a directory, which was not available in no other configuration. But even this directory was different from the ones defined in the other simulators. This one was localised in the last Last Level Cache (LLC) instead of in DRAM. Also, to have a 2D mesh it enforced the use of an in-order CPU, different from the other simulators. So, gem5’s results will be greatly different. On one hand the CPU type will impose a slow down but the cache coherence scheme may have a positive impact, if the simulated workload relies greatly on communication.

Next two results sets are presented, the first from the execution of handcrafted tests, and the second from the execution of the PARSEC benchmark suite.
4.3.2 Handcrafted Workloads

To establish a baseline comparison built upon small and simple workloads, two handcrafted tests were developed for all simulators. These were: pi, and n-body.

Their properties are described in Table 4.5.

<table>
<thead>
<tr>
<th>Program</th>
<th>Parallelization</th>
<th>Data Usage</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Granularity</td>
<td>Sharing</td>
<td>Exchange</td>
</tr>
<tr>
<td>pi</td>
<td>data-parallel</td>
<td>coarse</td>
<td>low</td>
</tr>
<tr>
<td>n-body</td>
<td>data-parallel</td>
<td>fine</td>
<td>high</td>
</tr>
</tbody>
</table>

Pi was developed entirely from the ground up. As for n-body it was modified based on the example shipped as part of Graphite simulator.

To execute these workloads on all three simulators different steps had to be taken.

First of all the benchmarks had to be parallelized. Pi was parallelized by having one core responsible for splitting the work by all the other ones, spawning one thread per core. The core responsible for splitting the work will also contribute to the execution of the work not just assigning it. So, the work is evenly distributed by all cores.

The implementation of the workloads for Sniper and gem5 was straightforward, using the standard pthreads library it was possible to implement all the desired functionality.

The implementation of these workloads for Graphite had a key difference due to it not supporting the pthreads library in its integrity. So, this library was replaced by Graphite’s “Carbon” library that has substitutes for the necessary pthread functions for thread management.

4.3.2.1 Pi

Figure 4.1 depicts the simulated execution time, the time that the program took to execute in the simulated architecture, on all three simulators.

It is possible to observe that the difference of the simulated time is very high between the three simulators, but these results reflect the differences of their configurations.

Sniper’s results are always the smallest, as expected, due to its out-of-order CPU. This workload benefits from this because it performs a repetitive computation on a big a set of data, ideal conditions to take advantage of an out-of-order CPU's advantages, as ILP.

Graphite’s results are higher than Sniper’s, as expected, due to its simpler IOCOOMCPU. But as the number of cores increases the difference of their results decreases. From Graphite's simulated architecture taking almost eight times more time than Sniper’s, with two cores, to taking four times more time, with 132 cores.
While Sniper's and Graphite's results, although very different, are more close to each other than gem5's when compared with either of them. Nevertheless, a higher simulated execution time was expected. The presence of a simpler and slower CPU type, the much more detailed simulation of the router, the presence of an OS and multiple devices, all add up to an impressive difference between the results. Even with such differences in the simulated system these results make clear that some aspects simulated by Sniper and Graphite are under estimated, which leads to results better than the effective architecture. This is visible with the values of the interconnection network contention on Figure 4.1.

As it is made clear by Figure 4.2 the interconnection network contention, which is a critical aspect of the performance of an architecture, is completely different in the three simulators. Graphite's values are a mere fraction of the values of Sniper and gem5. The latter also has much higher values than the former. This abrupt difference of gem5's values is due to two key aspects: the presence of directory's on each tile LLC and of how the simulation of the contention is carried out in them. The first aspect should have a slight impact on the overall contention because this workload does not require extensive communication, as for the second aspect it consists on the detailed way that gem5 performs the simulation of the routers present on the tiles, which reproduces reliably the behaviour of actual routers. In contrary Sniper and Graphite have simpler models built upon queues that are inaccurate and under estimate the communication delays [46].

Another aspect to consider is the access delay to the DRAM, show on Figure 4.3.
Unfortunately gem5 did not provide a way to extract this information from the simulation, so only the data from Sniper and Graphite is presented.

Only with 16 cores the differences between the two simulators are negligible. But with every other core count the values are quite distinct. The number of accesses is almost identical in either of them, the average delay is the key factor for this disparity. For example when the workload is executed with 132 cores, the average DRAM access delay is close to 100 ns in Graphite and around 50 ns in Sniper. As the cache coherence architecture and the interconnection network are identical this is due to the different models implemented in these simulators.

In Figure 4.4 it is possible to compare the host simulation execution time of the three simulators. This set of results falls into the expectations: Sniper is the fastest simulator, followed by Graphite, and finally by gem5. One particular aspect to take note is that as the number of cores increases up to eight the host simulation execution time decreases on Sniper and on Graphite, while on gem5 it increases. This is due to the fact that gem5 can only execute in a single thread, while the other two can split the work across multiple threads, and even amongst multiple hosts, in the case of Graphite.

4.3.2.2 N-body

Figure 4.5 depicts the simulated execution time on the three simulators.

The simulated execution time obtained when executing the n-body workload has a different behaviour than the previous workload. Here Sniper's results are greater than Graphite's. This behaviour was not expected based on the established configuration, nevertheless it can be explained. The ex-
The execution of this workload on Graphite differs from how it was executed on Sniper and gem5. In these the workload was coded using the standard pthreads library. As for Graphite it was coded with its own library that mimics the original one. This is due to Graphite not supporting fully pthreads, so to bypass its unresolved issues in this incomplete implementation, an alternative library, shipped with the simulator, is used.

This was also present in the previous workload and no impact was visible. This is so because the previous workload had very little communication, as opposed by this workload that spends around 90% of the simulated execution time in communication. So, any difference in the implementation of such a crucial element will have an high impact in the final. It is very difficult to provide definitive proof of this, anyway at this moment the Graphite development team is improving the support to standard pthreads.

Figure 4.6: N-body interconnection network contention.

Figure 4.6 depicts the network contention in all three simulators. Analysing this results it is possible to observe that they also display the effects of a workload with high communication. As in the previous workload Graphite’s results are almost negligible when compared with the results of the other simulators. Yet the contention modelled by Sniper is the highest, contrary to what was observed in the previous workload. This is due to the different cache coherence scheme defined in gem5’s configuration which reduces the overhead of communication, causing the value of the contention to decrease below the one observed in Sniper’s simulation.

Another aspect to consider when analysing an architecture is the access delay to the DRAM shown on Figure 4.7.
The DRAM access delay is consistent with the one observed in the previous workload. Also, this results can be considered fairly similar.

Finally in Figure 4.8 is possible to compare the different host simulation execution times. As expected gem5’s times were greatly higher than the other simulators. Also, it was expected that Sniper would take less time than Graphite, but due to the different implementation of pthreads, Sniper has to execute more instructions thus increasing the host simulation execution time.

4.3.3 PARSEC

The handcrafted workloads tested in the previous section increases the confidence on the results of simulator, as its results fell into the expectations. However these tests were indeed very simple, not imposing a great strain on the simulator nor to the host. Also, these tests are not standardised, making it harder for others to reproduce the results obtained.

To bypass this problem, the benchmark suite previously chosen, in Section 3, was used to test the simulators in a broader scale. These results can be easily reproduced due to PARSEC’s availability online. Also, it allows to test all the key aspects to look for when simulating systems with high number of CPUs.

As mentioned in Section 4.1.3 the support of PARSEC is different amongst the simulators. So, to confirm that in fact the different versions of PARSEC do not cause any difference the Graphite simulator was adapted to also be able use the legacy 2.1 version of PARSEC. The adaptation of Sniper to the 3.0 version would also have been possible, but do to the fact that it is an newer version it could include requirements that Sniper does not yet comply. Thus, Graphite was adapted to execute
the 2.1 version of PARSEC.

Graphite documentation \[47\] states that its current versions only supports PARSEC 3.0. Nevertheless, this was circumvented by not using the PARSEC installation scripts shipped with Graphite. To enable PARSEC 2.1 to function the workloads were compiled manually with handcrafted configuration files, to allow their simulation by Graphite. Accompanied by some configuration changes on a Graphite configuration file, it was made possible to execute the workloads as if the 3.0 version was installed.

Therefore, results of the Graphite and Sniper simulators are presented using PARSEC 2.1. This will still provide a form of comparison between the simulators, while also allowing a cross examination with the results of Graphite with PARSEC 2.1.

4.3.3.1 blackscholes

The first tested workload across all three simulators was blackscholes. The following figures ease the effort to analyse its behaviour in the multiple simulators. Figure 4.9 depicts the total simulated execution time.

![Figure 4.9: blackscholes simulated execution time.](image)

It can be observed that the simulated execution time, in Figure 4.9, notwithstanding the different versions of PARSEC, the Graphite measurements are very similar and much higher than those of Sniper. So, it seems to indicate that the version of blackscholes included in PARSEC 3.0 and PARSEC 2.1 have the same performance on Graphite, proving some similarity. Also, its source code was analysed and no major changes were found.

Sniper is the simulator with smaller simulated execution times. This is due to its faster out-of-order CPU that is able to improve the speed of this workload because it relies more on computation than in communication.

As observed before, Section 4.3.2 gem5’s results are the higher by a large margin due to its simpler and slower CPU type. However an interesting effect is present. As the number of cores increases its different cache coherence mechanism allows it to reduce the gap to the other simulators.
To bypass the problem of analysing results from different configurations, these can be put in proportion. In Figure 4.10 it is possible to observe how the simulated execution time varies when the number of cores increases.

![Figure 4.10: blackscholes simulated execution time proportion relative to two cores.](image)

Figure 4.10 allows to draw two different conclusions. First, in Sniper and in Graphite if the same version of *blackscholes* is executed then the same proportion of simulated execution time is observed. This indicates that notwithstanding the differences in the simulator and in the configuration of the architecture at least is possible to assure that the differences in the simulated time are only due to this differences and not to erratic behaviour of the simulator. Meaning that the results are consistent.

Second, although gem5 is executing the same 2.1 version of PARSEC’s *blackscholes* that Graphite and Sniper are executing, its results are extremely close to the 3.0 version. So, either this similarity is due to the inherent differences in the architecture and in the simulation mode of gem5 or this indicates that version 3.0 of PARSEC has indeed some differences.

Figure 4.11 depicts the host execution time of the simulators.

![Figure 4.11: blackscholes host execution time.](image)

From the data on the graph above it is possible to conclude that Sniper takes more time than Graphite with either versions of PARSEC on this workload.
Also, gem5 is by far the slowest simulator, increasing this difference as the number of cores increases due to not having a parallel implementation.

![Figure 4.12: blackscholes host execution time proportion relative to two cores.](image)

In the graph above the proportion between the simulator execution times is displayed. This is relevant because the simulators may have different execution times for the same workload but how the execution time varies between different instances, with different inputs, of the same workload should remain fairly similar.

Here it is possible to observe that the execution time proportions of Sniper and Graphite with PARSEC 2.1 are extremely similar. On the other hand, Graphite with PARSEC 3.0 has a completely different behaviour. It shows a significantly higher proportion in all the tests performed.

Therefore, it can be concluded that in fact the blackscholes workload in PARSEC 2.1 and PARSEC 3.0 must have indeed some differences. This contradicts the information made available on the changes of the benchmark [68].

4.3.3.2 fluidanimate

The following figures depict the behaviour of the simulators when executing fluidanimate PARSEC benchmark.

Figure 4.13 depicts the simulated execution time.

In line with the results obtained in the previous workloads, Sniper was the simulator with the smallest values and gem5 with the highest results. Nevertheless, it was observed across all simulators that as the number cores was increased up to 64 cores the simulated execution time decreases. As for when the number of cores reaches 132 it is observed a slight increase on the execution time on Sniper and on Graphite with both versions of PARSEC.

Unlike in the previous workloads, Graphite was not able to perform this workload of PARSEC 2.1 with two, four and eight cores, although it was possible to execute the same workload but of the 3.0 version of PARSEC. So, to analyse the obtained data two figures were created. Figure 4.14 depicts the behaviour up to eight cores, as for Figure 4.15 it allows to observe the behaviour from 16 cores up to 132.
Simulated Execution Time (ns) vs. Number of Cores

Figure 4.13: fluidanimate simulated execution time.

Simulated Execution Time Proportion vs. Number of Cores

Figure 4.14: fluidanimate simulated execution time proportion relative to 2 cores.

Figure 4.15: fluidanimate simulated execution time proportion relative to 8 cores.

The former figure only allows to observe that Sniper and Graphite with different versions of fluidanimate have the same proportion. In the latter figure Sniper and Graphite with PARSEC 2.1 have very similar results, but with 132 cores they diverge more. On the other hand Graphite with PARSEC 3.0 has a much higher proportion than either Sniper or Graphite with the 2.1 version.

Host Execution Time (s) vs. Number of Cores

Figure 4.16: fluidanimate host execution time.
The host execution time keeps the trend of the workload previously tested. In Figure 4.16 it is possible to observe that gem5 takes always more time than the other simulators. Also, Sniper always presents the lower host execution times as expected.

As for the host simulated time proportion, in Figures 4.17 and 4.18, its results are in line with the ones of the simulated execution time proportion. Here the values for Sniper and Graphite with PARSEC 2.1 are fairly similar, as expected, and Graphite with PARSEC 3.0 are higher.

**4.3.3.3 streamcluster**

The Figure 4.19 depicts the behaviour of the simulators when executing streamcluster PARSEC benchmark.

Before starting the analysis of the obtained results for this result it is important to call the attention for the fact that it was not possible to execute it with 132 cores with any of the simulators due to the fact that it does not finish with more than 64 cores. Also, it was not possible to execute it with Graphite using the version included in PARSEC 2.1. This was caused by the workload crashing with every count.

It is possible to see in Figure 4.19 a behaviour similar to what was observed on all the previous
workloads. Sniper’s simulated execution time was inferior in all core counts, except with 64 cores, due to its faster out-of-order CPU. Graphite, even though with a different version of PARSEC follows relatively closely behind. Finally, gem5 takes more time than both of them.

With the help of Figure 4.20 it is possible to conclude that the host execution time of gem5 is in line with the previous workloads. As for Sniper it did not provide the lowest results.

4.3.3.4 swaptions

Another workload that was possible to execute in all the simulators was swaptions. Next are multiple figures that depict the behaviour of the simulators when executing this workload. Image depicts the simulated execution time observed on all three simulators.

Unlike the afore observed workloads, in this the simulated execution time of Sniper and of Graphite with PARSEC 2.1 do not line up. The results of Graphite are very close with exception of the simulation with 64 cores. On the other hand, gem5 has a much higher execution time. Nevertheless, it decreases over time. This set of results seems to indicate that the swaptions workload included in both 2.1 and 3.0 versions of PARSEC are the same or very similar. This is so because notwithstanding the different
versions of the simulator, the simulated execution time differences resemble those observed in the handcrafted workloads, see Section 4.3.2.

The simulated time proportion matches perfectly between all three simulators, a signal of consistency of the workload across them.

Both the host execution time, Figure 4.24, and the host execution time proportion, Figure 4.25, show a similar behaviour to the one observed in the previous simulations.

The only result that steps aside from this, is the result for Graphite with PARSEC 2.1 with 64 cores, which can be attributed to experimental error of this particular instance.

4.4 Conclusion

In this section multiple aspects of the chosen simulators were analysed under multiple lights, from the first initial contact with them to details of the simulated architectures.

On the constraints imposed by the simulators, they have multiple key differences. They simulate different types of CPU, interconnection networks, cache coherence mechanisms and other elements.
Even when the same element is simulated its simulation is never carried out in the same way. All this aspects made it unlikely to obtain the same results when simulating the same system.

The ease of use was good with the Sniper and Graphite simulator, due to the former consisting on a derivative of the latter. As for gem5 it proved itself harder to use.

But the key issue in the use of simulators is the consistency and reproducibility of results, for the same system and workload, across the simulators. The lack of capacity to specify the exact same configuration in all the simulators made the process of comparison difficult and less trustworthy. Nevertheless, the majority of the results, specially the ones from the handcrafted workloads, complied with the expectation for each individual architecture. So, it is possible to assure that the chosen simulators have at least a medium degree of consistency, thus increasing the confidence of the results.

Another question is if this results obtained with the simulators would actually map to the results of an actual system with the simulated architecture. This is difficult to confirm. For instance, gem5, a simulator built to be very accurate, as been found to have discrepancies ranging between 5% [69] and 50% [70]. Also, Sniper has been validated against the Intel Core 2 microarchitecture [71], but differs greatly from the results obtained with gem5.
Concluding, the available simulators present many issues, and still face many issues. However, they can already provide some insights for the thousand-core era.

The exploratory architectural interconnection network and cache coherence tests, described in the rest of this document, are performed with Sniper. This is due to its higher simulation speed and correction of the obtained results. However, the fact that it can only simulate an out-of-order CPU is a drawback that limits the exploration of differences in performance at this level in architectures with a high number of cores.
5

Architectural Exploration

Contents

5.1 Interconnection Network Simulations .......................... 52
5.2 Cache Coherence Simulations ............................... 71
The adequate exploration of possible many-core architectures is the topic at hand in this chapter. In it, the results of multiple tests are discussed. The characteristics of the testing platforms discussed in the previous chapter, see Chapter 4, are also mentioned to justify the results observed on these tests.

The results presented on this chapter are obtained using the Sniper simulator due to the conclusions drawn on the Chapter 4. These are divided in two key sections: interconnection network simulations, and cache coherence simulations.

The first encompasses all the tests performed where the interconnection network was modified. Here, there are small interconnection network simulations, and large interconnection network simulations. The former contain tests up to 132 cores while the latter has tests from 256 cores up to 1024 cores. Also, results for the execution of the spidergon-donut interconnection network, described in Section 2.1, are also presented.

The second contains all the tests performed with the goal of improving the cache coherence performance, keeping the same interconnection network. Here the simulations presented are just relative to a high number of cores due to its higher relevance to the topic at hand, and due to time constraints caused by the high execution time of Sniper with the workloads of the PARSEC benchmark suite.

With these simulations performed, a comprehension of the effect of different options on many-core processors, will become visible. These results are analysed and a set of best choices for both interconnection network, and cache coherence are brought forward.

### 5.1 Interconnection Network Simulations

The study of interconnection networks for many-core processors begins with the exhaustive test of their performance. This is performed by executing the PARSEC benchmark suite on them. The results obtained have to be analysed and compared, with the goal of picking the best interconnection networks.

The first set of tests include the most promising interconnection networks, accordingly to the background work, that are possible to simulate on the Sniper simulator. Based on these restrictions five different interconnection mechanisms were possible to analyse: 2D Mesh, 2D Mesh Cluster, 2D Torus, 2D Torus Cluster, and Global Bus. These are described as interconnection mechanisms and not interconnection networks because the Global Bus is not qualified as a network, due to not having routers in its tiles.

In Table 5.1 are depicted the values of the specifications of the simulated architectures, with the exception of the Global Bus, that have the greatest impact on their performance. Following it its a detailed explanation of the rationale of the options presented.

- **hop_latency**: specifies the number of cycles the simulator is going to assume per each hop, i.e., the communication between adjacent routers;

- **link_bandwidth**: defines the maximum number of bits that can be conveyed through the inter-tile links per each cycle;
Table 5.1: 2D Mesh interconnection network architecture specifications, used in the simulation.

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>hop_latency</td>
<td>2</td>
</tr>
<tr>
<td>link_bandwidth</td>
<td>64</td>
</tr>
<tr>
<td>concentration</td>
<td>1</td>
</tr>
<tr>
<td>cache_levels</td>
<td>3</td>
</tr>
<tr>
<td>directory_type</td>
<td>Fullmap</td>
</tr>
<tr>
<td>directory_implementation</td>
<td>DRAM</td>
</tr>
</tbody>
</table>

- concentration: specifies the number of cores in each network interface, i.e., included in each tile;
- cache_levels: specifies the number of levels in the cache hierarchy;
- directory_type: defines the directory logic to handle its entries;
- directory_implementation: specifies in what physical device the directory is implemented, in order to include the appropriate delays.

This configuration maps exactly to the 2D Mesh configuration, the other configurations are slight modifications of this one. The 2D Mesh Cluster is as the previous configuration with the difference that the concentration value is four, meaning that each tile has not one but four cores. The 2D Torus and 2D Torus Cluster are the corresponding modifications of the two previous networks with the difference that the network forms a torus instead of a mesh.

As for the Global Bus, it consists of the Intel Xeon Gainestown architecture. The adequate configuration of the simulation parameters for this architecture is shipped with Sniper. So, simulations with it should assure a higher degree of confidence in the results. Another aspect that favours the utilisation of this architecture is that the values that it uses, for delays and other aspects, were obtained from analysis of the real processor, also providing a high confidence of the simulation results. Furthermore, it uses the desired global bus to serve as baseline comparison with nowadays processors.

Figure 5.1, on the left side, is part of the image of the 2D Mesh and 2D Torus interconnection networks, that is possible to generate with the Sniper simulator. As for Figure 5.1, on the right side
shows the configuration of the tiles for the 2D Mesh Cluster, the 2D Torus Cluster, and for the Global Bus configuration. Also, it is possible to observe the difference between having a shared L3 cache, and DRAM controller, with and without clustered cores. On the figures is possible to observe the cores, the caches, the “tag-dir” that contains the tags used to control the cache blocks on the L1 and L2 caches, and the “dram-cntlr” that stands for DRAM controller which allows the access of the main memory.

The other specifications of the simulated architecture are consistent with the recent Intel Xeon Gainestown architecture, used in the Global Bus, partially depicted on Table 5.1.

As mentioned before, this choice was made to assure a more realistic set of results. This configuration is based on a real architecture, fact that allows to overlook a set of important aspects of the specifications, for the moment, and focus solely on the architecture. Also, it uses a cache coherence mechanism based on a directory, something that is seldom found in nowadays processors, making it an almost unique combination.

To reduce the analysis complexity the conducted tests are divided in two sets. The first set concerns tests with architectures up to 132 cores, while the second set comprehends architectures with a greater number of cores. The analysis complexity changes due to the fact that the interconnection network characteristics impact in different ways when with they have a different number of cores. Also, the configuration of the PARSEC benchmark suite is distinct in the two sets. This benchmark suite allows to adapt the size of problem that is going to be processed by the program, providing four input sets with increasing sizes: small, medium, large, and native. Due to the large input set imposing high simulation times, from hours to days, this was only used for the architectures with over 256 cores. As for the other architectures, these were tested with the small input set. Despite its name, it is still complex enough to elucidate the architectures advantages and disadvantages.

### 5.1.1 Small Interconnection Network Simulations

In this section the results of the simulations of 2D Mesh, 2D Mesh Cluster, 2D Torus, 2D Torus Cluster, and of the Global Bus interconnection mechanisms presented before are discussed. This set of results comprehend simulations up to 132 cores, executing PARSEC’s workloads with its small input set.

The goal was to present results for all the workloads on the PARSEC’s benchmark suite but, as mentioned in Section 4.1.3 and on Table 4.2 some of the workloads had difficulties to execute. From all the workloads the following were not executed: bodytrack, facesim, freqmine, and raytrace. The facesim workload was only fully executable with one core, when it is executed with more cores the simulation crashes. The freqmine workload did in fact execute with all core counts but its results were deemed irrelevant because it did not distribute the execution effort. It only used one core to process the information, having always the same performance for any number of cores. The raytrace workload was only executable up to 16 cores, from that point on the simulations crashed. As the set of results was very small, these were not included.

Also, due to the 2D Mesh Cluster and the 2D Torus Cluster interconnection networks use of a tile
containing four cores it was not possible to simulate these with just two and 35 cores, therefore these results are not presented.

Next the results for the workloads that were indeed fully executed are presented.

### 5.1.1.1 blackscholes

In Figure 5.2, the results of the blackscholes workload are presented. It is possible to observe that the results obtained, with either interconnection mechanism, are very similar. This lack of distinction of very different interconnection mechanism is due to the characteristics of this workload. The blackscholes workload is computation intensive and does not rely much on communication. So, as the cores themselves are equal no great difference is perceivable. Nevertheless, it is remarkable how the execution time decreases with the increase of the number of cores, and how the Global Bus has a good performance even though it has a single contention point.

### 5.1.1.2 canneal

In Figure 5.3 the results of the canneal workload are presented.

The execution of the canneal workload contrasts with the one of the blackscholes. In this the results do not line up on every core count, existing a constant better performance of all the interconnection networks when compared with the Global Bus.

This workload is the one with the most demanding memory behaviour, so a fast access to main memory is crucial. The Global Bus implies a higher contention due to its single choke point which penalises its performance.

The interconnection networks have a slight advantage when executing this workload. Even though they share one DRAM controller for every four tiles, which causes that every time the memory has to be accessed it has, in the worst case, to make four “hops”, two to reach the closest DRAM controller and other two back. It does not have a single contention point, so multiple DRAM controllers stand
at the ready to access the main memory as soon as possible, reducing the overall memory access delay.

5.1.1.3 dedup

In Figure 5.4, the results of the dedup workload are presented. The results obtained from the dedup benchmark challenge the preconception that the Global Bus presents the worst results when the number of cores increases. The results of the interconnection networks are quite similar in all the tests, as for the Global Bus it is equal to the other interconnection mechanisms when the architecture has four cores, but afterwards its execution times are always smaller by a large margin. These can only be explained by this workload having a pipeline parallelization model, which combined with a high need for communication gives advantage to Global Bus. As for the other interconnection mechanisms, these pay the price of the router and interconnection delays and do not harvest the foreseeable advantages.
This workload allows to draw the conclusion that accordingly with the parallelization model that is going to be put in place, an interconnection network is not always the best solution when dealing with high number of cores.

5.1.1.4 ferret

In Figure 5.5, the results of the ferret workload are presented. Unfortunately the 2D Torus Cluster interconnection network failed to execute this workload, crashing with all core counts.

![Figure 5.5: ferret simulated execution time comparison.](image)

Notwithstanding, the results obtained with the other workloads fall precisely within the expectations. Before analysing the graph, a remark to the fact that the first set of data belongs to an architecture with eight cores instead of the four cores used in the previous tests, this is due to this workload failing to execute with a number of cores bellow eight.

When the number of cores is eight the 2D Mesh, 2D Mesh Cluster, and 2D Torus, take more than double of the time to execute than the bus.

5.1.1.5 fluidanimate

In Figure 5.6 the results of the fluidanimate workload are presented.

The results obtained with the fluidanimate workload mimic the ones obtained with blackscholes. Most of the workloads present a very similar simulated execution time across all interconnection networks, and the Global Bus.

As this workload has a low communication and a data-parallel parallelization model, a possible explanation for these different interconnection mechanisms presenting the same results is that the workload himself does not demand enough to make the differences amongst them clear. Also, something that is also visible is its small simulated execution time, always inferior to one second.

5.1.1.6 streamcluster

In figure 5.7 the results of the streamcluster workload are presented. Before proceeding to the analysis of the results for this workload, it is important to call attention to the fact that this, unlike the
other workloads, does not have the results for the interconnection mechanisms with 132 cores. When streamcluster was executed with any core number higher than 64 it failed to end. No error was given but it executed for more than one week without ending.

The results are fairly similar up to 16 cores. With the available data at 35 cores the 2D Mesh presents the lowest simulated execution time.

Unlike the previous workloads, here the simulated execution time does not decrease every time the core count increases. This happens because when Sniper executes it, the working set is increased to use all the available cores. Otherwise some of them would be idle. Therefore, in this workload speedups can not be measured.

5.1.1.7 swaptions

In Figure 5.8 the results of the swaptions workload are presented.
The swaptions workload shows a relatively similar behaviour of all the five interconnection mechanisms. One note that has to be taken when analysing it is that with this workload it does not make sense to simulate more than 64 cores due to it only having 64 work units to distribute. So, the 64 core simulations have the peak performance that is achievable with this workload.

Concluding, with this workload the differences of the interconnection mechanisms do not express themselves. Having all the interconnection mechanisms almost the same performance, only diverging by a small margin.

5.1.1.8 vips

In Figure 5.9 the results of the vips workload are presented.

The figure compares the simulated execution time on the five interconnection mechanisms in core counts from four to 132 cores.
One first global conclusion that can be draw from the obtained results, is that the results are very similar. These have a maximum difference amongst themselves of 17 ms, about 0.017 seconds. Nevertheless, the differences observed represent trends that should be observed as the number of cores, and the complexity of problems, increase.

The 2D Mesh interconnection network presents the worst results for four and eight cores, however this tendency is inverted when the number reaches 16.

The 2D Torus interconnection network presents the best results with eight cores, for the rest of the tests its results stand between the results of the 2D Mesh and of the Global Bus interconnection networks.

The Global Bus presented the best results for four cores, as it would be expected, and remains close to the other results throughout the tests. It was expected that the bus would perform better in architectures with a small core count, due to not having to pay the penalties associated with a interconnection network, router delays, amongst other aspects. Yet it remains fairly close in higher core counts, and has in fact the best simulated execution time with 64 cores, something not expected.

5.1.1.9 x264

In Figure 5.10 the results of the x264 workload are presented.

This workload presents the oddest results of the entire benchmark suite. It is observable that the execution time decreases from four to eight cores, as expected, but when the number of cores reaches 16 the execution time becomes extremely small, and remains in values similar to this on the rest of tests. This behaviour can not be backtracked to the interconnection network, due to it behaving in the same manner with all three interconnection mechanisms. This odd behaviour can only derive from the work distribution mechanism implemented by the workload itself. Despite it stating that it can split the work up to 128 cores, see Table 3.3, it must allocate more than one work unit per core to have this behaviour. Consequently, this workload must not be considered a good measure of performance of the interconnection mechanism.
5.1.1.10 Conclusion

The results described above start shedding some light over what to expect of architectures with a number of cores from 4 up to 132. In these results there are three main conclusions to draw: the Global Bus presents results much better than expected, the 2D Torus Cluster presents more frequently the best results of the interconnection networks, and overall the differences of performance between different interconnection mechanisms are rather small.

Despite the evaluation of the Sniper simulator performed on Chapter 4, the results obtained here raise some concerns over the correctness of the simulator, especially the Global Bus results. Taking into account all the different characteristics of each workload the performance can have variations. Still the Global Bus had a suspiciously good behaviour with 64 and 132 cores, where the contention should already be extremely high. So, either the simulator is providing exceedingly optimistic results for the Global Bus, or it is a good option up to 132 cores.

5.1.2 Large Interconnection Network Simulations

This section presents the results of tests comprehending a core count superior to 132 cores. In these, the initial expectation is that the bus contention, using the Global Bus interconnection mechanism, will increase vastly. This is expected to be its major bottleneck, proving how inadequate this interconnection mechanism is for high core counts.

Before advancing to the analysis of the results, one observation that has to be made about this set of tests is that not all the PARSEC workloads, used to perform the simulations with a small core count, are part of this result set. This is due to multiple problems in their executions with this high number of cores. For instance, the streamcluster benchmark does not finish with any of the core counts here tested.

So the workloads that were able to execute with this number of cores were: blackscholes, canneal, dedup, and fluidanimate.

5.1.2.1 blackscholes

The comparison of the blackscholes workload execution time between the five interconnection mechanisms is presented in Figure 5.11.

![Figure 5.11: blackscholes simulated execution time comparison.](image-url)
As it is possible to observe, the differences in the execution time are small in every core count. For 256 cores the 2D Torus and the Global Bus take the least time. This result contradicts the initial expectation for this test set, however it can be explained by the characteristics of this workload.

The blackscholes workload has small working sets and communication, see Table 3.3 which alongside with the shared L3 cache can reduce vastly the utilisation of the bus. So, it will not suffer from the high contention that harms its scalability on high core counts.

But there is also the issue of the performance of the other two interconnection mechanisms. These never suffer from contention of a single bus do to their inherent network characteristics, but they do pay the price of having a interconnection network through the delay caused by the routers that have to forward packets calculating the packet routing.

Nevertheless, with 528 cores the Global Bus presents the worst results.

5.1.2.2 canneal

The comparison of the canneal workload execution time between the five interconnection mechanisms is presented in Figure 5.12.

This set of results presents a challenge to explain do to its odd behaviour. Analysing this result set by interconnection mechanism, it is possible to conclude that the 2D Mesh has a close to linear grow of the execution time, a result that can be explained by the workload itself becoming bigger as the number of cores increases. The 2D Torus presents the worse results for 256 cores, but it has better results than the 2D Mesh. This differences are too expressive to be only dependent on the interconnection network, that are quite similar.

The oddest aspect of this test is the results for the Global Bus. For 256 cores it has better results than the 2D Torus and the 2D Mesh Cluster, as for 528 cores it presents the best result by a short margin.

canneal works with huge working sets and uses a fine grain parallelism, built upon atomic instructions to synchronise, and adapted to the cache size of the architecture. This causes the Global Bus to have a key advantage. The four interconnection networks presented here mimic the cache behaviour of the Intel's gainestown architecture, with one L3 cache per each four cores. So, each time a core needs a cache block of the L3, which happens constantly with synchronisation built upon locks, the
Global Bus (gainestown architecture) will have an access delay extremely short due to its L3 cache composing its cache hierarchy. On the other hand the 2D Mesh and 2D Torus interconnection networks may impose a huge delay due to the cache block request having to hop through six routers to get the cache block.

Programs that need synchronisation or, in a more general way, rely on the use of shared caches are commonly found, and as the number of parallelized programs and the CPUs that execute them get more parallel power this problem will only increase. Therefore, one conclusion can be drawn from this benchmark is that L3 shared caches should either be avoided in 2D Mesh and 2D Torus, or should rely on a different connection mechanism than the interconnection network itself. On the other hand this problem does not occur with the 2D Mesh Cluster and the 2D Torus Cluster interconnection network.

5.1.2.3 fluidanimate

The comparison of the fluidanimate workload execution time between the three interconnect mechanisms is presented in Figure 5.13.

![Figure 5.13: fluidanimate simulated execution time comparison.](image)

The results of this workload corroborate the initial expectation. The Global Bus presents the worst results throughout the tests. As for the interconnection networks, the 2D Torus Cluster presents the best results for 256 and 1024 cores. As for 528 the better interconnection network is the 2D Torus.

The fluidanimate workload is characterised for having large working sets with some communication. So, this workloads mimics the behaviour of the dedup benchmark.

5.1.2.4 swaptions

The comparison of the swaptions workload execution time between the five interconnection mechanisms is presented in Figure 5.14.

![Figure 5.14: swaptions simulated execution time comparison.](image)

The Global Bus performance complies with the initial expectation, by having a much higher simulated execution time than the interconnection networks. The interconnection networks present a fraction of the Global Bus time, showcasing their validity for many-core processors. Globally the 2D Torus Cluster presents itself as the best choice, having the lowest simulated execution time.
5.1.2.5 Conclusion

In this section it was possible to get a good overview of the most promising interconnection mechanisms many-core processors with 256, 528, and 1024 cores.

As expected the Global Bus presented execution times higher than the interconnection networks in almost every test, usually by a large margin. This complied with the expectations, that state that as the number of cores using a bus increases the contention becomes exceedingly high which hurts performance.

As for the interconnections networks these usually had a performance relatively close to each other. Albeit this, the 2D Torus Cluster interconnection network had the best performance more frequently.

In this set of tests the results of the Global Bus have started to depict the effect of high contention. Nevertheless, these still seem too optimistic. For instance the fluidanimate workload with 256 all the interconnection mechanisms have a similar performance. This appears improbable as the Global Bus will have an bus connected to 256 cores, that even with low communication should impose a high level of contention, and subsequent memory access delay, that should increase the simulated execution time further.

5.1.3 Spidergon-Donut Interconnection Network Simulations

Accordingly to the research carried out previously, see Section 2.1, a interconnection network that provides the best performance is the Spidergon-Donut. However it is currently not implemented in any simulator.

The solution to this was implementing it on the Sniper simulator. To achieve this the implementation of the 2D Torus was modified to add the additional connection that set it apart from the Spidergon-Donut. This process is explained in detail in Appendix A.

The simulations presented next were performed with a Spidergon-Donut with multiple spidergons with four tiles. This configuration was chosen due to the fact that each DRAM controller is every four cores, so in each spidergon every core has a direct connection to the DRAM controller. This key difference should provide a relevant performance increase because this accesses are very frequent, especially applications with heavy communication, so the delay per packet should decrease slightly.

Due to execution timing constraints only simulations up to 132 cores were conducted with this interconnection network. The results are presented next.
5.1.3.1 blackscholes

In Figure 5.15, the results of the blackscholes workload are presented.

![Figure 5.15: blackscholes simulated execution time comparison.](image)

In this workload, the reduced level of communication and the high percentage of time spent on actual computations makes it less prone to improvements by this interconnection network. Nevertheless, the results of the Spidergon-Donut are always as low as the 2D Mesh or even lower.

5.1.3.2 canneal

In Figure 5.16, the results of the canneal workload are presented.

![Figure 5.16: canneal simulated execution time comparison.](image)

The canneal workload is communication intensive, so here the advantages of this interconnection network should become clear. Despite this the results showed a performance very similar to the 2D Mesh. Only having a slight decrease of execution time with eight cores, and a small increase in the execution time with 64 cores.

These results may arise form the Spidergon-Donut only having four cores per spidergon. Despite it reducing the time to the DRAM controller, it may have a negative impact in core to core communication.

5.1.3.3 dedup

In Figure 5.17, the results of the dedup workload are presented.
The *dedup* workload has a high level of communication and a pipeline parallelization model, so an interconnection network that reduces the delay of communication should improve the performance vastly. Despite this, the *Spidergon-Donut* interconnection network does not decrease the simulated execution time. In fact it is almost equal to the the 2D *Mesh* in every core count.

### 5.1.3.4 ferret

In Figure 5.18, the results of the *ferret* workload are presented. Here the results only comprehend eight and 16 cores because when it was simulated with higher core counts the simulation crashed.

This workload has the same characteristics as the *dedup* workload: high communication, and a pipeline parallelization model. So, the results obtained are in line with what had already been observed. The results of the *Spidergon-Donut* interconnection network are almost equal to the 2D *Mesh*.

### 5.1.3.5 fluidanimate

In Figure 5.19 the results of the *fluidanimate* workload are presented.
The *fluidanimate* workload is computation intensive and has negligible communication. So, the effects of the *Spidergon-Donut* interconnection network should not be visible. The results confirm this hypothesis, having the same performance all the interconnection mechanism in all core counts. Nevertheless, there is a slight advantage of the *Global Bus* with 132 cores.

### 5.1.3.6 *swaptions*

In Figure [5.20](#) the results of the *swaptions* workload are presented.

![swaptions simulated execution time comparison](image)

This workload is characterised by having a low communication and a coarse parallelization granularity. So, the effects of the *Spidergon-Donut* should not be visible. Despite this, this interconnection network improved upon the *2D Mesh* results in all core counts, except with four cores.

### 5.1.3.7 *vips*

The *vips* workload, see Figure [5.21](#) is computation intensive and has a coarse parallelization granularity. The effect of this interconnection network should not have a great impact. Nevertheless, it
is visible that the Spidergon-Donut keeps up with the 2D Mesh, and improves upon it with eight and 64 cores.

![Graph showing simulated total execution time comparison](image)

**Figure 5.21**: vips simulated execution time comparison.

### 5.1.3.8 Conclusion

The Spidergon-Donut interconnection network was pointed out by previous research (see Section 2.1), as the best choice using conventional technology. Nevertheless, the results point just to a slight decrease of the simulated execution time when compared to the 2D Mesh. So, it falls short to the previous expectations.

### 5.1.4 ATAC Interconnection Network Simulations

The ATAC interconnection network (see Section 2.1.3), differs from the previously seen interconnection networks by taking advantage of recent developments in the integration of optical components inside a chip.

So, it is interesting to put this solution to the test and compare its results to the ones obtained with the ones of typical networks.

To test it was used the same testing methodology that was applied before. This interconnection network is tested with the PARSEC benchmark suite. However unlike the previously seen networks, in this case it can only be simulated using Graphite because it is only implemented on this simulator.

Therefore, the version 2.1 of PARSEC will be used to provide a more fare comparison.

Before advancing to the comparison of the results it is important to point out that only three simulations were able to be performed with it. The rest of the simulations failed to end, due to the workload crashing when it was simulated.

The causes for this behaviour are due to implementation of ATAC, because this simulator with these workload had already been used before without incidents. So, as the change of interconnection network was the only factor modified, it must be the cause of the constant crashes.

Thus, only three workload were possible to execute: blackscholes, fluidanimate, and swaptions. All these were executed solely with 256 cores.
Also, this interconnection network has multiple possible configurations. In this case it uses four cores per cluster, mimicking the cluster size seen previously, and it uses a star intra-cluster interconnection network.

The results are presented next.

### 5.1.4.1 blackscholes

The comparison of the blackscholes workload execution time between the interconnection networks is presented in Figure 5.22.

![Figure 5.22: blackscholes simulated execution time comparison.](image)

In this workload the ATAC interconnection network provides the worst result by far. Even tough ATAC brings improvements by reducing the communication time between clusters, the few memory accesses that blackscholes needs to perform may have a higher cost that with the other interconnection mechanisms. So, the advantages of this interconnection network may only become visible on other communication heavy workloads.

### 5.1.4.2 fluidanimate

The comparison of the fluidanimate workload execution time between the interconnection networks is presented in Figure 5.23.

![Figure 5.23: fluidanimate simulated execution time comparison.](image)

As in the previous workload the ATAC workload has the slowest results by a large margin.

### 5.1.4.3 swaptions

The comparison of the swaptions workload execution time between the interconnection networks is presented in Figure 5.24.
Despite the swaptions workload not splitting its work for more than 64 cores its results are still showcased here because they differed substantially, and because the number of workloads that were possible to executed with ATAC was very reduced. The results for this workload inverted what was observed in the two previous set of results, here the interconnection network with the worst results was the 2D Mesh.

5.1.4.4 Conclusion

Unfortunately, the available implementation of the ATAC interconnection network did not suffice to perform all the desired tests.

The results showcase the results only for 256 cores. Nevertheless, it is visible that the ATAC interconnection network does not perform better in all cases, on the contrary on two of the three workloads it underperformed by a large margin. The only workload where it was better than the 2D Mesh was with swaptions.

Despite the Graphite simulator stating that it supported the execution of programs on the ATAC interconnection network fully it fell short.

5.1.5 Interconnection Network Simulations Conclusion

The testing of a series of different architectures encompassing multiple interconnection mechanisms enables the discovery of their key characteristics and performance.

On the small interconnection network simulations, Section 5.1.1 the blackscholes, fluidanimate, streamcluster, swaptions, vips, and x264 workloads presented the same performance with all the different interconnection networks. With the canneal workload the Global Bus presented the worst results, while with dedup and ferret it presented the best results.

From these results the first conclusion that can be drawn is that up to 132 cores the Global Bus seems to provide the best performance. This contradicts the expectation that the contention would hurt the performance vastly as the number of cores increase. This good performance may be due to the simulator underestimating the contention and due to these workloads not causing enough strain on the architecture to observe this penalty.

On the large interconnection network simulations, Section 5.1.2 the blackscholes presented the same performance on all of the analysed interconnection networks. For the rest of the workloads the
Global Bus usually presented the worst results, while the 2D Torus Cluster presented the best results more often.

As for the Spidergon-Donut and the ATAC interconnection networks they were pointed out by research as the best options when using current technology and with on-chip photonics, respectively. Nevertheless, the obtained results did not confirm the expectations raised by the research.

Overall, the main problem in the execution of these tests was the uncertainty generated by odd results that did not comply with the initial expectations, as the case of the Global Bus.

5.2 Cache Coherence Simulations

With the simulations comprehending the part of the interconnection network concluded it is time to test multiple schemes of cache coherence.

As discussed in Section 2.2, there are multiple issues that influence the choice of an appropriate cache coherence scheme. For instance the directory can be distributed amongst the LLC of all tiles, or off-chip on the DRAM. Also, it can be tested the NUCA mechanism.

In this section the interconnection mechanism chosen was the 2D Mesh. This is so because this interconnection network is the most discussed and in actual use, so the results obtained here will have more impact, will be more useful, and will be easier to reproduce.

So, in this section four different architectures are analysed: 2D Mesh, 2D Mesh private L3, 2D Mesh in cache directory, and 2D Mesh NUCA.

The architecture named 2D Mesh is the same that was used in the previous section, to perform the analysis of the interconnection network with the same name. From the point of view of cache coherence concerns it has an off-chip fullmap DRAM directory, with 8 MB L3 caches shared among every four tiles.

The 2D Mesh private L3 tackles the issue of high L3 cache access latency faced in the previous architecture. Here there is no LLC shared amongst a sub set of cores, every tile has its own L1, L2, and L3 caches. To make the comparison as fair as possible the total L3 cache size in the architecture was maintained, so every L3 cache has 2 MB.

As for the 2D Mesh in cache directory tries to face the problem of high L3 access delay observed in the 2D Mesh architecture while reducing the directory access delay. Here the L3 cache in shared among every two tiles thus reducing the worst case scenario to access the closest DRAM controller, observed in the 2D Mesh architecture, from four “hops”, to only two. Also, the directory is distributed amongst all tiles in a special memory block denominated “tag-dir”, see Figure 5.1. This memory block acts as the LLC for simulation proposes. In the literature about distributed directories these are always placed in the LLC, but due to the Sniper simulator constraints this was not possible to achieve. Nevertheless, this “tag-dir” memory block should provide a close enough recreation of an actual distributed directory that uses the LLCs.

Finally, the 2D Mesh NUCA seeks to explore the performance gains, or losses, of using a non-uniform cache access mechanism. Here 1 MB of the L3 cache would be accessible in this form. This
could avoid unnecessary duplication of the same data, but can also introduce excessive delays if the access cache is too many “hops” away.

5.2.1 Large Cache Coherence Simulations

In this section are the simulations with a core number higher than 132 cores. The same workloads that were executable in section 5.1.2 were executed in here. But due to the different architectures simulated some results were not possible to obtain.

5.2.1.1 blackscholes

The comparison of the blackscholes workload simulated execution time between the four interconnection networks is presented in Figure 5.25.

The results of the blackscholes workload allow to enlighten the differences between the three architectures. With 256 cores the 2D Mesh with private caches has a fraction of the execution of any other architecture. As for higher core counts, in these the 2D Mesh has the lowest simulated execution time. It is followed closely the the 2D Mesh private L3.

5.2.1.2 canneal

The comparison of the canneal workload simulated execution time between the four interconnection networks is presented in Figure 5.26.
In this workload there is not clearly defined pattern as in the most simulations done before. Here for 256 cores the 2D Mesh is the best one, for 528 the 2D Mesh in cache directory and for 1024 cores 2D Mesh private L3.

5.2.1.3 ferret

The comparison of the ferret workload simulated execution time between the four interconnection networks is presented in Figure 5.27.

![Figure 5.27: ferret simulated execution time comparison.](image)

The ferret workload has one clear choice as the best one: the 2D Mesh private L3. In every core count this architecture is incomparably small when facing the results of the other architectures.

5.2.1.4 fluidanimate

The comparison of the fluidanimate workload simulated execution time between the four interconnection networks is presented in Figure 5.28.

![Figure 5.28: fluidanimate simulated execution time comparison.](image)

This workload does not have a clear best choice for the architecture. For 256 cores the 2D Mesh private L3 is the best option, taking almost one third of the time taken by the other architectures. Then for higher core counts the 2D Mesh presents the best results.
5.2.1.5 swaptions

The comparison of the swaptions workload simulated execution time between the four interconnection networks is presented in Figure 5.29.

From the available data, for 256 cores the 2D Mesh private L3 is the best option.

5.2.2 Cache Coherence Simulations Conclusion

With the multiple workloads executed in all five architectures with different cache coherence mechanisms, some conclusions can be drawn:

- With 256 cores the architecture that prevailed in all the workloads was the 2D Mesh private L3;
- With 528 cores the blackscholes, fluidanimate, and swaptions workloads presented the best result with the 2D Mesh. On the other hand, the canneal workload takes less time the 2D Mesh in cache directory. Also, the ferret workload takes less time with the 2D Mesh private L3;
- With 1024 the results were also not consistent, on the blackscholes and fluidanimate workloads the 2D Mesh presented the best results. While on the canneal and ferret workloads the 2D Mesh private L3 presented the best results.

From this set of results it is possible to conclude when architectures with 256 cores are built with a mesh, they should use a private L3 cache instead of a shared one. This mimics the behaviour of the 2D Mesh private L3.

If an architecture with 528 is to be built and has to execute workloads with low communication, it should follow the 2D Mesh defined architecture. If it has heavy communication it should use the 2D Mesh in cache directory architecture. Otherwise, it should use the 2D Mesh private L3 architecture.

Finally, to build an architecture with 1024 cores it should use the 2D Mesh if low communication is foreseen. Otherwise it should use the 2D Mesh private L3.
6

Execution Enhancement

Contents

6.1 Interconnection Network Aware Thread Mapping .......................... 76
6.2 Parallel Thread Creation .............................................................. 79
6.3 OpenMP Constraints ................................................................. 80
The problem of using a many-core CMP to its full potential has two distinct elements: the hardware, and the software.

A correct option for the interconnection network and the cache coherence mechanism compasses the key issues on the hardware front, discussed in Chapter 5, but it can only take the solution to this problem so far.

As for the issues in the software front, they comprehend the programming model and the parallelization strategy, discussed in Section 2.4.

With the possibility of using simulators to execute benchmark suites on top of many-core architectures it became possible to create and test multiple different implementations of the same handcrafted workloads. These executions allowed to extract some insights and conclusions about which are the best choices in the parallelization strategies for these architectures.

In this chapter are analysed: the influence of thread placement on the 2D Mesh and the 2D Mesh Cluster interconnection networks, parallel thread creation, and OpenMP constraints.

6.1 Interconnection Network Aware Thread Mapping

Architectures with a high number of cores provide a challenge that is often overlooked in nowadays parallelization strategies: adequate thread placement. Due to the low core counts in place today on general purpose processors, an inefficient thread placement does not cause large penalties. However with the increase of cores per processors this may be one of the first problems to arise.

The issue of thread placement already receives some attention in the program parallelization process due to its inclusion on the Foster’s Design Methodology, constituting its last step.

So, the novel interconnection networks analysed in Section 5.1 may perform very differently accordingly to a bad or a good thread placement. One interconnection network in which this issue has already been tackled is the ATAC. Its authors state that threads sharing some memory should be placed in the same cluster, and threads that share memory constantly should be placed in adjacent cores if possible. These are two different situations, because if two cores have to read and write the same data sporadically their location inside the cluster does not have an impact. On the other hand when they share data constantly the cache coherence scheme in place can improve their performance by standing close to each other.

Analysing the interconnection networks mentioned in Section 5.1 there are two key aspects to look out when mapping threads to cores: the distance between two threads sharing data, and the presence of clusters. The former is understood intuitively, in the interconnection networks that were analysed before. Closer cores can usually share data faster, so threads that share data should be as close as possible. Although assigning more than one thread per core is common and useful, here it is only considered a maximum of a thread per core due to the simulators not supporting more than one per core. The latter, as discussed in the ATAC interconnection network, is also easy to grasp. When clusters exist on a processor they usually share a LLC (see Figure 5.1). If two threads share data and are on the same cluster, the sharing of a cache will speed up their execution by a large margin.
Table 6.1: 1D Heat workload description.

<table>
<thead>
<tr>
<th>Program</th>
<th>Parallelization</th>
<th>Data Usage</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D Heat</td>
<td>data-parallel</td>
<td>fine</td>
<td>Calculates the temperature in ( n ) points along a conductive pole</td>
</tr>
<tr>
<td></td>
<td>Granularity</td>
<td>Sharing</td>
<td></td>
</tr>
</tbody>
</table>

Data being read will only cause a cache read miss instead of two, and data being written will require only one invalidation instead of two and a lot less communication delays which also lowers the overall contention.

To test the afore mentioned suppositions, an experiment was executed where an workload that required communication and needed information processed by adjacent threads was developed. A workload that has this behaviour is the 1D Heat, described on Table 6.1.

With the workload defined the testing methodology must be defined. Each interconnection network used will execute two instances of 1D Heat, one with a good thread mapping and a second with a poor thread mapping. For this workload the good thread mapping chosen was to have them assigned consecutively, each thread being responsible for adjacent points of the pole. On the other hand the bad mapping was having them mapped so that threads that are responsible for non-adjacent points are next to each other.

In the Figure 6.1 is an example of a good and a bad thread placement for the 2D Mesh Cluster, as it is for the 2D Mesh.

This section is divided in: 2D Mesh, and 2D Mesh Cluster. In each section the interconnection network that names it will be tested with the good and bad thread mapping.

6.1.1 2D Mesh

The testing methodology discussed before is applied to the 2D Mesh interconnection network already used on the architectural exploration chapter, in Section 5.1.

Next are the results for the average L3 cache write and read misses behaviour for good and bad thread mappings.

The Figure 6.2 allows to observe the difference between a good and bad mapping of threads, and its impact on the cache hit rate.
Here both Write Miss Rates are quite similar even though there are two very different thread mappings in play. This happens because each thread is responsible for writing on a specific position, to calculate the temperature of a position of the pole. So, no invalidation will occur and the local cached copy will always be up to date.

As for the Read Miss Rates these differed greatly. As each thread reads the positions that are controlled by the threads next to it the good mapping has a much lower miss rate than with a bad mapping. This difference does not encompass just a higher statistic of the behaviour of the cache, it has real impact on the overall performance of the processor. This workload is relatively small so other factors impose a great hold on the obtained execution time of these multiple experiments. Nevertheless, for 16 the execution time is around 33% greater with the bad mapping and 100% greater with 64 cores. As for 8 cores their execution times are equal. This analysis of the results also gives another insight, as the number of cores increases this problem only gets worse. So, an adequate handling of this issue has to start now before serious impacts are felt.

6.1.2 2D Mesh Cluster

The testing methodology discussed before is applied to the 2D Mesh Cluster interconnection network.

Next are the results for the average L3 cache write and read misses behaviour for good and bad thread mappings.

The behaviour observable on Figure 6.3 mimics the one observed on Figure 6.2. This happens because the differences in the interconnection network do not have an impact on the cache access behaviour of this workload. Nevertheless, the execution of the same workload on a different interconnection network allows to showcase two different aspects. The first is that multiple executions may have slightly different cache behaviours due to the existence of a high number of threads that causes a non deterministic execution. The second is that, even tough the same cache behaviour is present...
the overall execution time is different. Here the execution time increases less than 1% when good and bad mappings are compared, due to the fact that this interconnection network reduces the delay to the L3 cache.

With the tests performed in this section it is possible to conclude that an adequate mapping of threads, considering the problem at hand, and the interconnection network is a crucial step to optimise the execution of workloads in architectures with a high number of cores. Furthermore, if this problem is not tackled it can still be minimised be the use of better interconnection networks as shown by the 2D Mesh Cluster results.

### 6.2 Parallel Thread Creation

The thread creation process takes a negligible amount of time when analysing the overall execution time of a program. This statement is true when discussing nowadays processors that require a low number of threads. But when discussing architectures with a higher number of cores this delay cannot be overlooked.

Consider a processor with 1024 cores. If the first core is responsible for creating every thread it will have to perform a thread creation process 1024 times and as usually the threads do not exist just to be idle, the threads that are created will start working immediately. Thus increases the contention, causing an increasingly higher delay as more threads are created.

However this issue does not only present itself with such high core numbers. The Intel Many Integrated Core (MIC) architecture [72] is a new solution that consists of a PCI card containing a high number of x86 cores to serve as a co-processor for an Intel Xeon processor. The current prototype Intel Knights Ferry (KNIF) has up to 32 cores, and here this problem already presents itself.

According to research [72] the thread creation process can be sped up almost by 7 times, on the aforementioned architecture. Despite the differences between this architecture and a many-core CPU, it still showcases the margin of improvement that can be done in this area.
The thread creation process can be sped up on Linux by parallelling the `clone` system call that is responsible for the spawning of threads. This parallelization takes advantage of idle cores so that it does not increase the execution time.

The optimisation of the thread creation process is one of the new challenges for the use of many-core processors. Nevertheless, the work already developed [72] provides solutions for the Intel MIC. Also, the research group that developed this solution defined as future work the adaptation of this parallel `clone` to a Linux kernel to be used with many-core processors.

### 6.3 OpenMP Constraints

The use of OpenMP with many-core processors has a series of challenges that do not arise with conventional multicore.

A good example of its inadequacy is when using it to split the work of a cycle. Here the usual automatic split of “for” may not provide the best performance. This happens because splitting all the steps of the “for” by a huge number of cores may impose a slowdown when compared with splitting by a smaller number of cores with a higher number of steps. This is due to the reuse of the information inside the cache block that is in cache. Also, the creation, mapping and destruction of threads, discussed before, will also have an impact.

To showcase this behaviour an experiment was conceived. A simple program would be executed on the 2D Mesh architecture with a high numbers of cores. This program first would use the default OpenMP way of splitting the work, by all the cores. After this the same architectures will execute the program but it would use a sub set of the cores. This program performs a simple addition to a global value of all the number for zero up to one million.

![Figure 6.4: OpenMP Simulated Execution Time Comparison.](image)

It is possible to observe on Figure 6.4 that the Standard solution does not provide the best results, as expected. The best results for this program are obtained when the work is split by 64 cores, on all three architectures.

To improve the behaviour of parallel “fors” there is no single solution. On one hand the user can define manually the appropriate value of the “chunks” that each thread, or core considering a thread...
per core in this case. On the other hand if no option is selected the “auto” mode should be modified to embrace the different constraints of many-core processors. Despite this option existing already configured for the usual nowadays multicore processors, the implementation of a good “auto” option should be quite difficult. At compile time or at run time the “chunk” size should be ideally set to the best option accordingly to the time that each step of the for takes. To perform this perfectly it would imply to execute the program and then set the “chunk”, something difficult to perform because the execution time may be a few milliseconds as days, information that is not provided. An easier way to perform this adaptation of the “chunk” size is using an heuristic. The choice of a good heuristic is the key issue here. It can be based on the number of instructions, and their type, among other options. It should be an important extension to OpenMP adding an adequate heuristic, so that pre-existent programs using the “auto” option would be automatically ready for many-core processors, at least in this aspect.

The adaption of the OpenMP for many-core processors is a crucial step to ease the transition to many-core processors. Embracing the challenges and opportunities of these processors by extending certain options of OpenMP will have a great impact on the development process, avoiding inefficiencies caused by bad programming from developers that ignore, or do not understand, the inherent characteristics of this architecture.
Conclusions and Future Work

Contents

7.1 Conclusion ........................................................................................................... 84
7.2 Future Work ......................................................................................................... 84
7.1 Conclusion

The topic of systems with thousands of CPUs constitutes a paradigm shift on the way the general purpose computation is carried out. With the available simulators not only it is possible to prepare today’s programs for this new paradigm, but they also provide a way of testing what are the most adequate cache coherence schemes and interconnection networks, from the traditional electrical networks to the on-chip optics networks.

The conducted research made it possible to give some confidence over the results of the Graphite, Sniper and gem5 simulators. Nevertheless, despite all the effort made to access the correctness and accuracy of the simulators’ results there were still unexpected outcomes. Using the Sniper simulator it was possible to analyse multiple interconnection networks and cache coherence mechanisms. No single correct answer exists as the best option for all cases, there are always trade-offs. Nevertheless, as a global conclusion the interconnection network that stood out in most cases was the 2D Torus Cluster and the cache coherence mechanism that stood out was the one on 2D Mesh private L3 for core counts lower or equal to 256 cores and 2D Mesh for higher core counts.

Other cutting edge interconnection networks were also tested but their results fell short from what was expected.

The enhancement of the execution by following some criteria when programming also leads to some interesting results. One effect to stand out in this area is the effect of good and bad thread mapping, consisting a critical step of the parallelization process.

Many-core processors are predicted by the end of this decade, with the research that already exits and the one that was developed here it is possible to state that we are now better prepared to face their many advantages and their drawbacks, enabling a smoother transition from the current paradigm to this new one.

7.2 Future Work

The research developed on interconnection networks and on cache coherence mechanisms, on Section 5.1 and on Section 5.2 comprehends a reduced subset of the entire possible experiments that can be carried out. Despite the fact that these presented themselves as the most promising ones, there is a lot of room for further experimentation and creation of more adequate solutions. The existence of highly configurable simulators that allow the addition of new interconnection networks and cache coherence mechanisms makes this job easier, as depicted on Appendix A.

Notwithstanding, the previously mentioned room for new experiments is smaller than the one there is for the development of adequate programming strategies for this new type of processors. The search of papers on this topic returned few results, even though there are a lot of unresolved challenges on this area. The topics presented on Chapter 6 try to solve some of these issues. These were found out by the contact with direct experimentation with many-core processors in the simulators, however much more should exist.
Bibliography


Adding Interconnection Networks to the Sniper simulator
In order to test the Spidergon-Donut interconnection network, in Section 5.1.3 the Sniper simulator was extended to accommodate this change.

Due to this extension process of the Sniper simulator not being made clear in the documentation, it is important to specify how an addition of a new interconnection network is performed.

Sniper has the folder common/network with all the code referent to the available interconnection networks, with four relevant files: network_model_bus.cc, network_model_emesh_hop_by_hop.cc, network_emesh_hop_counter.cc, and network_model_magic.cc. In these there is work already performed to simulate different types of interconnection mechanisms. The first comprehends all the interconnections where there is a need for a bus, while the second and the third should be used when a interconnection network is going to be built. Finally, when the interconnection mechanism that is going to be built has a fixed cost in all cases, typically only for testing purposes, the last one should be used.

The Spidergon-Donut is a interconnection network so there are two options the network_model_emesh_hop_by_hop.cc, and the network_emesh_hop_counter.cc. The difference between the two is that the latter does not provide contention modelling. As for the former it provides a higher accuracy but a lower performance. As the interest in this case is results with high accuracy the former was chosen.

In this file, and in its respective header file, are all the options needed to add a new interconnection network. Here, the following functions play an essential role:

- **createQueueModels**: here are defined all the queues that exist on this interconnection network, i.e., all the connections that a router, on a tile, has access to;

- **routePacket**: this function has the task of figuring out what path should a packet take to go from a router to another in the interconnection network;

- **computeDistance**: it is possible to have different distances due to the existence of multiple routing solutions, so here the distance that is between two routers is dictated. Be aware that this does not interfere with the routing of the packet itself, it only gives a numerical value. Also, if this distance is not consistent with the routing solution used on the routePacket function, the simulator will have an undefined behaviour;

- **getNextDest**: here as in the routePacket function the packet is routed from a router to another. But here only the position of the next hop of the packet is returned. As on computeDistance, if this is not consistent with routePacket the simulator will have an undefined behaviour;

- **computeMeshDimensions**: despite its name this function does not only calculate which are the dimensions of the interconnection network at hand. Here the values set on the configuration file passed to the simulator are read, existing a different chain of behaviour for different configurations and no only the size of the mesh;

Taking the example of the implementation of the Spidergon-Donut interconnection network, all the previous functions were modified to accommodate its changes. On the first one a new queue was added to handle the connections that the router in each tile has across the spidergon. To make this
possible an array with the directions that the packet can take was also altered in both files. This is visible in the Figure A.1.

Figure A.1: Section of the 2D Torus next to a Spidergon.

So, to implement this new interconnection network the existent implementation of the 2D Torus was taken advantage of.

As for the routePacket, the computeDistance, and the getNextDest functions these had to suffer major changes because the new connections that cross each spidergon, or the section of the 2D Torus, cause changes to the packet routing. Therefore, the computation of the distance had also to suffer from these changes.

Finally, it must be made possible for the user to pick this interconnection network in the configuration file. To made this possible the computeMeshDimensions function was altered so that when the network/emesh_hop_by_hop/dimensions value is set to three the simulator will use the Spidergon-Donut interconnection network.

Furthermore, two fields were added to the configuration file the: network/spidergon_donut/tiles_per_spidergon, and the network/spidergon_donut/number_of_spidergons. The first allows the user do define how many tiles should be allocated in each spidergon, while the second how many spidergon structures should be placed on the Spidergon-Donut. In the code it is possible to acces this configuration simply by executing Sim()->getCfg()->getInt(configuration name).
Poster Presented at the 51st Design Automation Conference
Background

Final year student of MSic in Information Systems and Computer Engineering at Técnico, Universidade de Lisboa
- Primary Field: Distributed Systems
  - Network and Computer Security
  - Virtual Execution Environments
  - Mobile Computing
- MSic Thesis: "Extreme Multicores: Programming of Systems with thousands of CPUs"
- Taken EDA related courses:
  - Parallel and Distributed Computing
  - Introduction to Computer Architectures
  - Operation System Design

Project Description

- Develop a tool to perform automatic tests on the gem5 [1], Sniper [2], and Graphite [3] simulators;
- Evaluate and compare the gem5, Sniper, and Graphite simulators;
- Access the performance of multiple interconnection networks, as: 2D Mesh, and 2D Torus. Compare it to the usual Global Bus solution in place today;
- Test the recent ATAC interconnection network;
- Access the performance of multiple cache coherence solutions for many-core processors, such as: directory on DRAM, directory on the LLC;
- Measure the impact on the overall performance of different cache hierarchies as: shared LLC, private LLC, NUCA;
- Extend the Sniper simulator by adding the spidergon-dout interconnection network;

Simulator Evaluation & Architectural Exploration

- The closest architecture possible to recreate in all the simulators was defined;
- The PARSEC benchmark suite, and a set of handcraft tests were executed on all three simulators;
- A set of interconnection networks were chosen, and tested using the Sniper simulator;
- A set of cache coherence solutions with matching cache hierarchies were chosen and tested using the Sniper simulator;

Results

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>gem5</th>
<th>Sniper</th>
<th>Graphite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interconnection</td>
<td>2D Mesh</td>
<td>2D Mesh</td>
<td>2D Mesh</td>
</tr>
<tr>
<td>Cache Coherence</td>
<td>MOESI</td>
<td>MSI</td>
<td>MSI</td>
</tr>
<tr>
<td>Cache Levels</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Directory Location</td>
<td>LLC</td>
<td>DRAM</td>
<td>DRAM</td>
</tr>
<tr>
<td>Directory Type</td>
<td>fullmap</td>
<td>fullmap</td>
<td>fullmap</td>
</tr>
<tr>
<td>CPU Type</td>
<td>in-order</td>
<td>out-of-order</td>
<td>in-order core out-of-order memory</td>
</tr>
</tbody>
</table>

Test Environment:
- CPU: Intel(R) Core(TM) i7 3770 @ 3.4GHz
- RAM: 16 Gb
- Executed on top of the Oracle VM VirtualBox
- OS: Debian Squeeze

Legend:
A) Blackscholes Simulated Execution Time on gem5, Sniper, Graphite
B) Blackscholes Simulated Execution Time Proportion on gem5, Sniper, Graphite
C) Blackscholes Simulated Execution Time on multiple interconnection networks
D) Blackscholes Simulated Execution Time with multiple cache coherence mechanisms

Figure B.1: Poster presented at the Design Automation Conference.