HAL
open science

Benchmark-driven Approaches to Performance Modeling of Multi-Core Architectures

Bertrand Putigny

To cite this version:

Bertrand Putigny. Benchmark-driven Approaches to Performance Modeling of Multi-Core Architectures. Distributed, Parallel, and Cluster Computing [cs.DC]. Université Sciences et Technologies - Bordeaux I, 2014. English. NNT : . tel-00984791v2

HAL Id: tel-00984791

https://theses.hal.science/tel-00984791v2

Submitted on 29 Apr 2014 (v2), last revised 24 Nov 2015 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

en vue d'optenir le grade de

Docteur de l'Université de Bordeaux

Spécialité : Informatique

au titre de l'école École Doctorale de Mathématiques et d'Informatique

présentée et soutenue publiquement le 27 mars 2014

par

Bertrand Putigny

Benchmark-driven Approaches to Performance Modeling of Multi-Core Architectures

---

Après avis de :

William	JALBY	Professeur, UVSQ	Rapporteur
André	SEZNEC	Directeur de recherche, Inria	Rapporteur

Devant la commission d'examen formée de :

Denis	BARTHOU	Professeur, IPB	Directeur de thèse
Olivier	COULAUD	Directeur de recherche, Inria	Examinateur
Brice	GOGLIN	Chargé de recherche, Inria	Encadrant de thèse
William	JALBY	Professeur, UVSQ	Rapporteur
André	SEZNEC	Directeur de recherche, Inria	Rapporteur
Josef	WEIDENDORFER	Chargé de recherche, Université de Munich	Examinateur

Remerciements

Il est coutume de remercier les personnes qui ont été importantes durant les années passées en thèse. Aussi, afin de ne pas déroger à cette tradition, je vais ici citer les personne qui ont compté durant ces 3 années et demi, et sans qui l'environnement de ces recherches n'aurait pas été aussi agréable.

Tout d'abord je tiens à remercier la région Aquitaine ainsi qu'Inria d'avoir financé mes travaux de thèse. Je remercie MM. Barthou et Goglin d'avoir accepté de m'encadrer durant cette thèse. Les membres de mon jury, tout d'abord MM. Jalby et Seznec pour avoir accepté de rapporter mon manuscrit et pour leurs retours pertinents, Olivier Coulaud pour avoir accepté de présider mon jury, ainsi que Josef Weidendorfer qui a fait un long déplacement dans des conditions difficiles pour assister à ma soutenance.

Au cours de cette thèse j'ai travaillé dans un environnement agréable en partie grâce aux membres de l'équipe Runtime que je tiens à remercier chaleureusement. En particulier Emmanuel, chez qui nous avons passé des soirées jeux très captivantes. Brice, auprès de qui j'ai beaucoup appris, et sans qui je n'aurais probablement pas terminé cette thèse, qui malgré les difficultés, s'est avérée bonifiante. Guillaume et ses connaissances poussées en nanars et en muay-thaï et surtout sa production d'œufs. Sans oublier Nathalie, Samuel, Pierre-André et Raymond. Les membres de "l'open-space": Cyril B. pour ses jeux de mots ; Cyril R. animateur de discussions enflammées ; François militant, engagé pour la neutralité des Internets ; Sylvain, résistant avec ferveur aux dogmatismes dans plusieurs domaines ; Paul-Antoine, notre expert en grammaire française et anglaise ; mais aussi James, Lilia ainsi que, plus récemment, Christopher qui a su trouver sa place dans l'open-space.

Je tiens également à remercier d'autres collègues chez Inria : comme Pascal ; Mathieu ; Emmanuel A. et Abdou adepte du triathlon à condition de remplacer la partie cycliste par une autre discipline que je laisse imaginer.

De manière générale je tiens à remercier les services d'Inria qui permettent aux chercheurs de travailler dans de bonnes conditions. Et je souhaite particulièrement remercier certaines personnes comme : Ludovic qui a passé du temps à nos côtés dans l'équipe Runtime ; François R. pour son humour et son engagement pour le bien-être des collègues et Marc F. défenseur de la langue française.

D'un point de vue personnel, je veux remercier les membres du club de triathlon de Bègles, source inépuisable de motivation. Je pense en particulier à la franchise de Julie (qui a toujours porté grand intérêt au confort de mes pieds), au dynamisme de Jojo, à l'authenticité de Maudinette, la jovialité de Mimi, la bonne humeur de Damien L., l'enjouement de Shrek, la bienveillance de Vincent, et pour finir l'altruisme de Pascalou.

Pour terminer, mes derniers remerciements vont à ma famille, qui toujours su trouver les mots pour me soutenir au cours de cette thèse.

Contents

Table of Contents iii

List of Figures vii

List of Tables x

Résumé en français 1

Introduction 3

1 Hardware Architecture 7
- 1.1 Core Architecture . . . 7
  - 1.1.1 Pipeline . . . 8
  - 1.1.2 Superscalar processor . . . 10
  - 1.1.3 Out-of-Order Execution . . . 10
  - 1.1.4 Vector Instructions . . . 13
  - 1.1.5 Low level Code Optimization . . . 14
- 1.2 Towards Parallel Architectures . . . 16
  - 1.2.1 The Energy Wall . . . 16
  - 1.2.2 Multi-Processor . . . 17
  - 1.2.3 Simultaneous Multithreading . . . 18
  - 1.2.4 Accelerators . . . 18
  - 1.2.5 Clusters . . . 19
- 1.3 Memory Architecture . . . 20
  - 1.3.1 Virtual Memory and Translation Lookaside Buffer . . . 21
  - 1.3.2 NUMA Architectures . . . 22
  - 1.3.3 Caches . . . 23
  - 1.3.4 Non-Coherent Caches . . . 31
- 1.4 Summary . . . 32

2 Performance Modeling 35
- 2.1 Propostion . . . 36
- 2.2 On-core Modeling: Computational Model . . . 36
  - 2.2.1 Related Work . . . 37
  - 2.2.2 A methodology to measure Latency, Throughput, and to detect Execution Port assignations . . . 38

2.2.3 Detecting Instruction Parallelism . . . 40
2.3 Case Study: Power Aware Performance Prediction on the SCC . . . 46
2.3.1 Related Work . . . 47
2.3.2 The SCC Architecture . . . 47
2.3.3 Performance Model . . . 48
2.3.4 Model evaluation . . . 51
2.3.5 Power efficiency optimization . . . 54
2.3.6 Summary . . . 56
2.4 Summary about On-core Modeling . . . 56
2.5 Un-Core Model: Memory . . . 57
2.5.1 Memory Hierarchy Parameters Needed to build a Memory Model . . . 57
2.5.2 Cache Coherence Impact on Memory Performance . . . 59
2.5.3 Bringing Coherence into a Memory Model . . . 60
2.6 Conclusion . . . 62

3 Designing Benchmarks for Memory Hierarchies 65
3.1 Problem Formulation . . . 66
3.1.1 Requirements of Benchmarks due to Cache Coherence . . . 66
3.1.2 Building Reliable Benchmarks . . . 66
3.2 Framework and Technical Choices . . . 67
3.2.1 Related Work . . . 67
3.2.2 Framework Overview . . . 69
3.2.3 Achieving Peak Memory Performance . . . 70
3.3 A Language to ease Benchmark writing . . . 71
3.3.1 Language Description . . . 71
3.3.2 Benchmark Compilation Framework . . . 73
3.4 Benchmarking Memory Hierarchy . . . 74
3.4.1 Motivating Example . . . 74
3.4.2 Automatic Generation of Coherence Protocol Benchmarks . . . 76
3.4.3 Comparing Cache Architectures and Coherence Protocols . . . 78
3.4.4 Guidelines for Improving Coherence Behavior . . . 87
3.5 Conclusion . . . 89

4 Benchmark based Performance Model 91
4.1 Scope and Model Overview . . . 92
4.2 Program and Memory Models . . . 93
4.2.1 Program Representation . . . 93
4.2.2 Memory Model . . . 95
4.2.3 Time Prediction . . . 97
4.3 Experiments . . . 99
4.3.1 MKL dotproduct . . . 100
4.3.2 MKL DAXPY . . . 102
4.3.3 FFT Communication Pattern . . . 103
4.3.4 Conjugate Gradient . . . 104
4.4 Application to Shared Memory Communications . . . 105
4.4.1 Intra-node Communication Memory Model . . . 106

4.4.2 Evaluation 108
4.4.3 Impact of Application Buffer Reuse 111
4.5 Conclusion 117
4.5.1 Discussion 117
4.5.2 Related Work 118
4.5.3 Summary 119

Conclusion **121**

Bibliography **125**

List of Publications **135**

List of Figures

1.1 A classical five stage Pipeline. 8
1.2 A Stall in a simple Pipeline without forwarding. 9
1.3 An Instruction Queue Example. 11
1.4 A vectorial Addition. 13
1.5 A SMP Multi-Processor System: 4 processors connected to their shared memory. . . 17
1.6 A Chip Multi-Processors. 18
1.7 Virtual and Physical memory mapping. 22
1.8 A NUMA Memory Architecture. 23
1.9 A cache illustration. 24
1.10 A parallel cache hierarchy. 27
1.11 The `MSI` protocol. 28

2.1 Overview of the SCC chip Architecture. 47
2.2 The Memory organization of the SCC. 48
2.3 Vector dotproduct model. 52
2.4 Matrix-vector multiplication model. 53
2.5 Matrix-matrix multiplication model. 55
2.6 Illustration of a benchmark to measure cache associativity. 58
2.7 Write Bandwidth measured on a Xeon X5650 Processor. 60
2.8 Read miss Bandwidth measured on a Xeon X5650 Processor. 61
2.9 Cost of an `RFO` message depending on the state of the cache line involved. 62
2.10 Cost of a write back message. 63

3.1 Benchmark compilation framework. 73
3.2 Load Hit Exclusive Benchmark results on two different Micro-Architectures. 75
3.3 Load Miss Exclusive Benchmark results on two different Micro-Architectures. 78
3.4 Micro-Architectures Compared. 80
3.5 Load Hit Benchmark Comparison. 81
3.6 Load Miss Benchmark Comparison. 81
3.7 Store Hit Benchmark Comparison. 82
3.8 Store Miss Benchmark Comparison. 84
3.9 A Intel Dunnington micro-architecture Socket. 84
3.10 Output of the Store Hit Benchmark on the Dunnington Micro-Architecture. 85
3.11 Parallel bandwidth ratio for several benchmarks on a Sandy Bridge processor 87
3.12 Full Benchmark Set results on Sandy-Bridge Micro-Architecture. 88

4.1 Illustration of the interaction of the different components of our Memory Model. . . 93
4.2 Illustration of Memory hierarchy Viewed by the Model 96
4.3 Automaton used to Track Memory State. 98
4.4 Dotproduct pattern performance prediction compared to MKL dotproduct 101
4.5 Dotproduct kernel speedup with 1 MB data set on Intel Sandy Bridge processor. . . 102
4.6 Dotproduct strong scalability on two 32 MB vectors. 102
4.7 DAXPY-pattern strong scalability prediction compared to MKL. 103
4.8 FFT pattern performance prediction on a Sandy Bridge Processor. 104
4.9 Speedup of the Conjugate Gradient and Predicted Speedup. 105
4.10 Cache state transition in a data transfer. 107
4.11 One socket of each kind of node in the evaluation platform. 109
4.12 Comparison of the benchmark based prediction and the actual transfer. 110
4.13 Benchmark-based prediction of both side memory copies. 111
4.14 Impact of buffer reuse on IMB Pingpong throughput with Open MPI 1.7.3. IMB was modified to support buffer reuse on one side without the other. Intel platform. . . . 111
4.15 Anatomy of a *Load Miss Modified* (step 3) in the MESI protocol. 112
4.16 Impact of a flush of modified data on the performance of reading from another core, on the Intel platform. 112
4.17 Impact of non-temporal stores and manually flushing on the performance of the sender write step 2, on the Intel platform. 113
4.18 Impact of non-temporal stores in the sender write step 2 on the performance of IMB pingpong between 2 cores on different sockets, on the Intel platform, with a modified Open MPI 1.7.3. 114
4.19 Anatomy of a *Store Hit Shared* (step 2) in the MESI protocol. 114
4.20 Impact of remote flushing on the performance of a local *Store Hit Shared* on the Intel platform. 115
4.21 Store Hit performance depending on Shared, Owned and Modified state, inside a shared L3 cache, on AMD platform. 116
4.22 Performance of shared-memory data transfer depending on buffer reuse direction, inside a shared L3 cache, on AMD platform. 117

List of Tables

1.1 Example of a Program executed by an Out-Of-Order Engine. 11
1.2 Anti-dependence avoided by register renaming. 12
1.3 Output-dependency avoided thanks to register renaming. 12
1.4 Loop Unrolling example. 15
1.5 Vectorization example. 16

2.1 Influence of the Loop Unroll factor on Loop Performance. 40
2.2 Instruction Latencies measured Comparison. 43
2.3 Comparison of several code versions of the `ADDPD` benchmark. 45
2.4 Comparison of execution time of two code versions. 45
2.5 Relation between voltage and frequency in the SCC chip. 50

4.1 Benchmark used for memory access time computation. 99
4.2 Memory access parallelism during a pipelined transfer when the message is divided into 3 chunks and the processor can execute one load and one store in parallel. . . . 107
4.3 Transitions involved in our model for each transfer step. 108

Résumé en français

Ce manuscrit s'inscrit dans le domaine du calcul intensif (HPC) où le besoin croissant de performance pousse les fabricants de processeurs à y intégrer des mécanismes de plus en plus sophistiqués. Cette complexité grandissante rend l'utilisation des architectures compliquée. La modélisation des performances des architectures multi-cœurs permet de remonter des informations aux utilisateurs, c'est à dire les programmeurs, afin de mieux exploiter le matériel. Cependant, du fait du manque de documentation et de la complexité des processeurs modernes, cette modélisation est souvent difficile. L'objectif de ce manuscrit est d'utiliser des mesures de performances de petits fragments de codes afin de palier le manque d'information sur le matériel. Ces expériences, appelées micro-benchmarks, permettent de comprendre les performances des architectures modernes sans dépendre de la disponibilité des documentations techniques.

Le premier chapitre présente l'architecture matérielle des processeurs modernes et, en particulier, les caractéristiques rendant la modélisation des performances complexe.

Le deuxième chapitre présente une méthodologie automatique pour mesurer les performances des instructions arithmétiques. Les informations trouvées par cette méthode sont la base pour des modèles de calculs permettant de prédire le temps de calcul de fragments de codes arithmétique. Ce chapitre présent également comment de tels modèles peuvent être utilisés pour optimiser l'efficacité énergétique, en prenant pour exemple le processeur SCC. La dernière partie de ce chapitre motive le fait de réaliser un modèle mémoire prenant en compte la cohérence de cache pour prédire le temps d'accès au données.

Le troisième chapitre présente l'environnement de développement de micro-benchmark utilisé pour caractériser les hiérarchies mémoires dotées de cohérence de cache. Ce chapitre fait également une étude comparative des performances mémoire de différentes architectures et l'impact sur les performances du choix du protocole de cohérence.

Enfin, le quatrième chapitre présente un modèle mémoire permettant la prédiction du temps d'accès aux données pour des applications régulières de type OpenMP. Le modèle s'appuie sur l'état des données dans le protocole de cohérence. Cet état évolue au fil de l'exécution du programme en fonction des accès à la mémoire. Pour chaque transition, une fonction de coût est associée. Cette fonction est directement dérivée des résultats des expériences faites dans le troisième chapitre, et permet de prédire le temps d'accès à la mémoire. Une preuve de concept de la fiabilité de ce modèle est faite, d'une part sur les applications d'algèbre et d'analyse numérique, d'autre part en utilisant ce modèle pour modéliser les performance des communications MPI en mémoire partagée.

Introduction

Need for Speed

The need for intensive computation is growing fast as more and more scientific fields rely on numerical simulation. Simulation is used in many domains in order to reduce production costs. For instance in the car industry it is cheaper to run crash simulations instead of crashing a real car. It is used in numerous areas such as aerospace or car industry, meteorology or geology *etc.*

Simulation has many advantages, among other the price, over real experiments. As, it is ran by a computer, it allows to record every information needed by scientists. One can easily change parameters of the experiments and run it again. We can also easily run simulations in conditions where the experiment could not be done. For instance because these conditions cannot be reproduced easily for a real experiment, or for security.

But all these strengths have also a drawback: simulation needs a lot of computational power. This means that simulation results can be very long to obtain. To overcome this problem, we need to build fast computers to shorten computation time. This explains why computer speed is crucial to science.

Computer Architecture

In order to fulfill this need for computation, hardware has to evolve as fast as the need for fast computation grows. For a period of time, speeding-up processor clock rate allowed to increase computer computational power. However, heat and power consumption of processors grow with the square of the processor frequency. Thus we have reached the limits of processors frequency with thermal resistance of processors. Central Processing Unit (CPU) designers had to find other ways to increase processors computational power. More and more architectural features have been added to computers in order to make them more powerful. Allowing, for instance, CPU to issue more than one instruction per cycle, this is called *instruction parallelism*. Another commonly used mean to increase processor computational power is to allow it to perform the same operation on several data at the same time, this is *data parallelism*. But the urge for computational power grows faster than architecture improvement. In order to keep up with growing needs for computational power, processor vendors had to go parallel. The area of single processor is now over and even general purpose computers, workstations and now even cell phones embed multi-core CPUs.

Hardware models and Software

The ever growing complexity of processors leads to numerous research topics for software optimization. Indeed, software has to be well adapted to the underlying architecture in order to benefit from all hardware features. Moreover we need to find a way to exploit all the parallelism available on the hardware. Expressing or finding parallelism in applications can be one tough research theme. New programming paradigms have been released in order to be able to express as much parallelism as algorithms have. However one has to be careful when writing software for High Performance Computing (HPC) since keeping all functional units busy in order to achieve good performance can be tricky due to dependencies. In order to be able to attain good efficiency on a machine, one has to know the architecture deep details. This can be a long task as computer architecture are becoming more and more complex. Moreover as processor vendors release new architectures very often, learning new architecture capabilities can become a big overhead for programmers.

In order to reduce this overhead, people build hardware models. These models are an abstraction of the architecture that helps understanding computer behavior. This also permits better adaptation of software to the machine. Building architecture models can still be burdensome. Even if one does not have to rebuild it from scratch for each new coming architecture, understanding how to use efficiently every new feature can be quite time consuming.

Goals and Contributions

In the race for better performance, computer architectures are becoming more and more complex. Therefore the need for hardware models is crucial to *i)* tune software to the underling architecture, *ii)* build tools to better exploit hardware or *iii)* choose an architecture according to the needs of a given application.

In this dissertation, we aim at describing how to build a hardware model that targets all critical parts of modern computer architecture. That is the processing unit itself, memory and even power consumption. We believe that a large part of hardware modeling can be done automatically. This would relieve people from the tiresome task of doing it by hand.

Our first contribution is a set of performance models for the on-core part of several different CPUs. This part of an architecture model is called the computational model. The computational model targeting the Intel SCC chip also includes a power model allowing for power aware performance optimization. Our other main contribution is an auto-tuned memory hierarchy model for general purpose CPUs able to *i)* predict performance of memory bound computations, *ii)* provide programmer with programming guidelines to improve software memory behavior.

Dissertation Organization

This dissertation is organized in 4 chapters. The first chapter is dedicated to a state of the art while the three others present our contributions.

Chapter 1 describes some existing computer architectures, hardware concepts and features. This is the basis for understanding both the motivations of research in the HPC field and motivations for our contribution.

Chapter 2 presents our contribution to help building computational models by automatically measuring instructions latencies and detecting instruction parallelism. It also presents a power aware performance model built for the Intel SCC.

Chapter 3 presents how to build benchmarks to model a memory architecture. Especially how to control the environment for representative and reliable benchmarks. We will also present a language we developed to ease the process of writing memory hierarchy benchmarks.

Last, Chapter 4 presents how to use benchmarks in order to build a performance model. And what choices have to be made to model memory. This model is evaluated by predicting the run-time of real codes running on the real hardware and is also applied to MPI communications.

Contents

1.1 Core Architecture **7**
1.1.1 Pipeline 8
1.1.2 Superscalar processor 10
1.1.3 Out-of-Order Execution 10
1.1.4 Vector Instructions 13
1.1.5 Low level Code Optimization 14
1.2 Towards Parallel Architectures **16**
1.2.1 The Energy Wall 16
1.2.2 Multi-Processor 17
1.2.3 Simultaneous Multithreading 18
1.2.4 Accelerators 18
1.2.5 Clusters 19
1.3 Memory Architecture **20**
1.3.1 Virtual Memory and Translation Lookaside Buffer 21
1.3.2 NUMA Architectures 22
1.3.3 Caches 23
1.3.4 Non-Coherent Caches 31
1.4 Summary **32**

Chapter 1

Hardware Architecture

"Welcome to the machine"
— Pink Floyd

As the need for intensive computation grows fast with the need for simulation, processor vendors had to find alternatives to increase CPU computational power.

In this chapter we will describe some important hardware features that increase the processor performance. The chapter is divided into three sections, one focuses on the core architecture itself: it explains how single processor architectures can be upgraded to deliver better performance. This section also explains how to optimize code in order to benefit from the hardware features presented. The second section presents why and how computer architectures are becoming parallel. It also describes several parallelism paradigms available in general purpose computers or clusters dedicated to high performance computing. The third section focuses on memory performance. We will present some features used to increase memory bandwidth as well as how to tune software to make better use of the memory.

1.1 Core Architecture

The core part of the processor is the one responsible for computation. It is a critical part of CPU design since it is responsible for executing all the instructions of a program running on the

machine. In order to execute an instruction, the processor needs to read, decode, execute the instruction, and eventually write the result back.

1.1.1 Pipeline

The Instruction Pipeline

One way to increase instruction throughput consists in devising the instruction execution into several stages. This allows a better usage of all functional units of the CPU. Since for a n stage pipeline, n instructions can be executed in the pipeline at the same time, each instruction being in a different stage. A pipeline does not reduce time to execute one instruction, but it allows to issue instructions while still executing others, which increases the instruction throughput. A common image to illustrate pipeline is to compare it to an assembly line. The classical pipeline is decomposed into the five following stages. A graphical representation of this pipeline is shown in Figure 1.1:

Instruction fetch: The stage is responsible of reading the instruction from memory and bringing it to the processor. In the stage, the instruction fetched for execution is pointed out by the *Program Counter* (PC). This stage is therefore also responsible for updating the PC to the next instruction to be executed.

Instruction decode: This stage is responsible for decoding the instruction, *i.e.* reading the instruction and its operands. The instruction is decomposed into the opcode, the operation to be executed, and its operands ,*e.g.,* registers or memory.

Execute: In this stage instructions are executed: for instance arithmetic instructions are dispatched to the *Arithmetic and Logic Unit* (ALU).

Memory: In the memory stage access to the main memory are performed.

Write back: The write back stage is responsible for writing the instruction results to the registers.

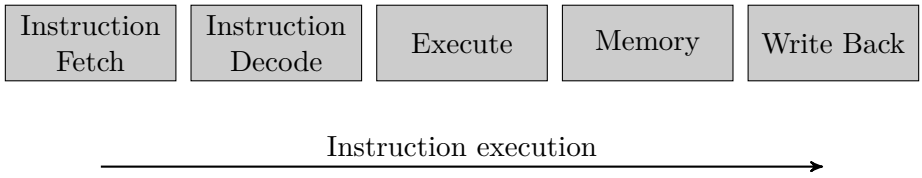


Figure 1.1: A classical five stage Pipeline.

Real world processors are composed of many more stages, Intel Core2 pipeline counts 14 stages while Nehalem has 16 stages. But the trend is toward shorter pipelines: the latest *NetBurst* micro architecture called *Prescott* has up to 31 stages.

While this description of a pipeline is simplified, it shows the basic operation of a pipelined processor. But even this small example allows us to illustrate several performance issues that can happen in pipelines. For instance the instruction pipeline is only able to increase hardware

performance if every stage of the pipeline is kept busy during the computation. This means being able to issue[1] one instruction at every cycle. For instance if several consecutive instructions have data dependences, meaning that some instructions need the result of others to be issued, the pipeline cannot be fed with one instruction at every cycle. Figure 1.2 shows a pipeline stall, consequence of data dependency between two instructions in the code. As we can see, if there is a data dependency between two consecutive instructions, the second instruction cannot be executed before the first writes its result into registers. On an n stages pipeline, this stalls the pipeline for $n-1$ cycles.

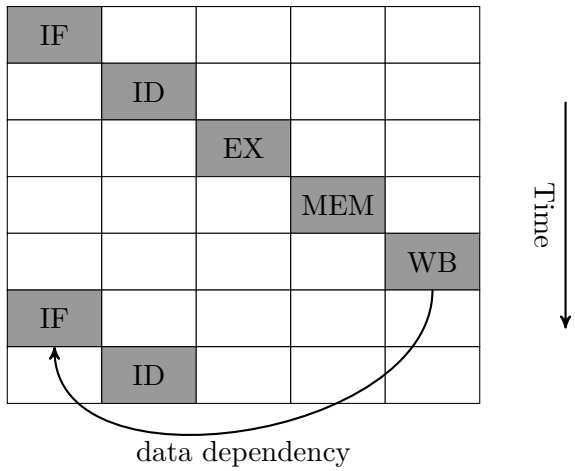


Figure 1.2: A Stall in a simple Pipeline without forwarding: the second instruction cannot be executed before the first one is retired, stalling the pipeline for 4 cycles.

Programs contain instructions controlling its execution flow, *i.e.* jump or conditional branching instructions. This can also lead to poor performance by stalling the pipeline.

Branch Prediction

In order to overcome stalls, branch prediction and speculative execution were added into processor pipeline [48, 72, 74]. When a conditional branch instruction is executed, the instruction pipeline cannot issue another instruction before this instruction is retired[2]. Indeed, the next instruction to be executed depends on the result of a condition. And the result of the conditional will only be available after the end of the execution of the condition.

As conditional jumps are used to implement loops, they are critical to achieve good performance. The 90/10 law says that programs spend 90% of their time in only 10% of the code. This portion of the code is therefore critical: this is usually loops. Branch prediction avoid stalling the instruction pipeline by deciding which way of the branch will be taken before the condition is retired. The next instruction can therefore be issued without stalling the pipeline. If the branch prediction was wrong, instructions that were issued when they should not have to be discarded. This is

[1] Issuing an instruction consists in starting its execution by feeding it to the first stage of the pipeline.
[2] An instruction is said to be retired when its execution completely over.

called speculative execution. Mispredictions present the same problems as pipeline stalls since the instructions executed after the branch will be discarded. Yet if branch prediction is correct this greatly improves branching performance. The longer the pipeline, the higher misprediction penalty and pipeline stalls.

Instruction Loop Buffer

As previously said, loop performance is critical. To improve loop efficiency some processors feature an instruction loop buffer. When the processor enters a loop, decoded instructions goto this buffer. This allows the bypass of the first stages of the pipeline: within a loop, instructions are decoded once and for all during the first iteration.

Register Forwarding

In order to reduce the penalty of pipeline stalls due to instruction dependences, a register forwarding mechanism can be added to the pipeline. This mechanism allows stages of the pipeline to provide a previous stage with data that has just been computed. This reduces the penalty of pipeline stalls by allowing the execution of instructions carrying dependence with an instruction already in the pipeline right after the execution stage instead of waiting for the result to be written back.

1.1.2 Superscalar processor

To further improve instruction throughput, processors were enhanced with superscalar capability. This mechanism is another form of instruction parallelism. It allows processors to issue more than one instruction per cycle. This processor optimization can lead to great computational power enhancement if one is able to bring several independent instructions to the processor per cycle. Indeed a superscalar processor with two pipelines will be able to issue two Instructions Per Cycle (IPC) leading to a twice higher theoretical peak performance.

To benefit from this feature, software have to present enough instruction parallelism and independent instructions. Otherwise the multiple execution ports will not be used. Modern processor such as the Sandy Bridge micro-architecture have six specialized execution ports. Specialized execution ports can only execute a subset of all available instructions. On the Sandy Bridge micro-architecture, three ports are dedicated to arithmetic and logical operations, two for memory reads and one for memory write. Leading to a maximum of three computations and three memory access during one clock cycle.

1.1.3 Out-of-Order Execution

We saw that control hazards due to branching instructions can be overcome by an efficient branch predictor. However pipeline stalls due to data hazards such as instruction dependences have not been tackled yet. This is the task dedicated to the out-of-order engine. The out-of-order engine allows the execution of other instructions when an instruction has to wait for its operands

to be ready. Instructions are therefore not executed in the initial order given by the program. Out-of-order pipelines also reduce the cache miss penalty by avoiding stalling the pipeline when miss occurs [73]. We will discuss in more details cache in Section 1.3.3

In order to implement an out-of-order engine, processor pipelines are extended with an instruction queue, a retire stage[3], and a register renaming mechanism to avoid unnecessary dependence.

Instruction Queue

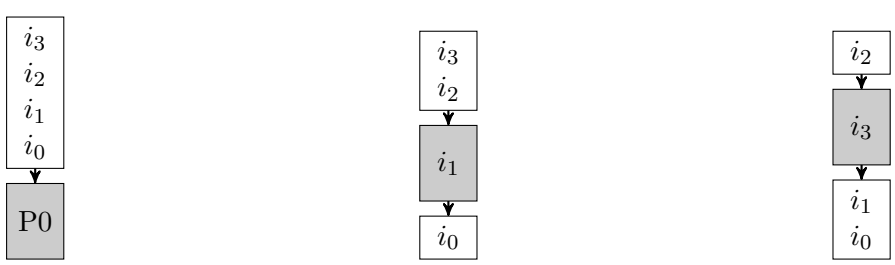
After decoding an instruction is dispatched to one of the execution ports. The instruction will stay in the queue until all its operand are ready. Therefore instructions will be executed as soon as its operands are available, even if older instructions are still waiting in the queue for their operands to be available. Instruction are said to be executed in data order instead of program order.

Figure 1.3 illustrates the execution of the program represented in Table 1.1 on an out-of-order pipeline. Figure 1.3a represent the pipeline with all four instructions waiting to be executed.

Table 1.1: Example of a Program executed by an Out-Of-Order Engine.

Program:	
i_0:	$r_0 \leftarrow 1$
i_1:	$r_1 \leftarrow 2$
i_2:	$r_2 \leftarrow r_0 \times r_1$
i_3:	$r_3 \leftarrow 3$

Figure 1.3b shows the state of the pipeline after instruction i_0 is retired and instruction instruction i_1 is in the pipeline. In Figure 1.3c we see that instruction i_3 is issued before instruction i_2 because instruction i_2 depends on instructions i_0 and i_1. Instruction i_3 is thus issued right after i_1 is issued. But i_2 has to wait for its operands r_0 and r_1 to be ready after i_0 and i_1 retire.



(a) Instructions is dispatched to the queue in program order.

(b) Instruction i_0 has been issued and is now retired. i_1 has been issued and is being executed: i_2 cannot be issued.

(c) The processor issues i_3. i_1 had time to finish: i_2 will be issued the next cycle.

Figure 1.3: An Instruction Queue Example.

[3]Also known as ROB: Re-Order Buffer.

Register renaming

The register renaming mechanism is used to avoid false data dependences. False data dependences are due to the name of the registers used (instead of being caused by real data dependences). If we take a look and the code shown in Table 1.2, instruction i_2 and i_3 cannot be executed at the same time since the variable B is both an operand of instruction i_1 and the output of instruction i_2. However we can rename B into B_0 and B_1, the code still computes the same thing but the

Table 1.2: Anti-dependence avoided by register renaming. Anti-dependence is also called Write after Read (WAR).

Before renaming			After renaming		
i_1:	B $\leftarrow$	1	B_0	$\leftarrow$	1
i_2:	A $\leftarrow$	B + 2	A	$\leftarrow$	B_0 + 2
i_3:	B $\leftarrow$	2	B_1	$\leftarrow$	2

instructions i_2 and i_3 can be executed at the same cycle since there is no dependence anymore between the Anti-dependences are called name dependences, if we can rename variables, or in the case of computer architecture, registers, we can avoid such dependences.

Another kind of name dependence can be avoided through register renaming: it is the output dependence. An output dependence happens when the same register is used as the result of several instructions, *e.g.,* in the code shown in Table 1.3 we cannot change the instruction order nor can

Table 1.3: Output-dependency avoided thanks to register renaming. It is also known as Write after Write (WAW) dependency.

Before renaming			After renaming		
i_1:	B $\leftarrow$	1	B_0	$\leftarrow$	1
i_2:	A $\leftarrow$	B + 2	A	$\leftarrow$	B_0 + 2
i_3:	B $\leftarrow$	X + 1	B_1	$\leftarrow$	X + 1

we execute any instruction in parallel since B is an operand of instruction i_2 and the output of instruction i_3. However if we rename B into B_0 and B_1, as shown in the right hand side of the table, we still compute the same thing, but we can now reorder instruction i_3 before instruction i_2 or perform both of them at the same time.

The goal of the register renaming mechanism is to avoid these name dependencies. Only a subset of all the physical registers of the processor are exposed to the programmer. When an instruction is executed, the register renaming mechanism chooses one physical register to use among the physical registers corresponding to the logical register provided by the instruction. Having several physical registers available for each logical register allows the processor to rename registers in order to avoid name dependencies.

Reorder Buffer

The register renaming mechanism used together with out-of-order execution engine avoids unnecessary stall in the pipeline by keeping the pipeline fed with instructions with satisfied dependencies. However as we saw in Section 1.1.1 dedicated to the instruction pipeline, because of the speculative execution and branch miss-prediction, the processor might have to discard some instructions. If instructions are not executed in program order, discarding the instructions speculatively executed after a branching instruction can become messy. In order to reorder instructions after they retired, a stage is added to the pipeline. This stage is called ROB for Re Order Buffer.

The ROB is a queue, as soon as an instruction enters the renaming stage, before dispatched to an instruction queue, an entry is reserved for this instruction in the ROB. Thus entries in the ROB are in program order. Instructions can only leave the ROB when they are retired and are at the head of the ROB. Hence instructions leave the ROB in program order, and the CPU is able to easily decide which instruction to discard when a branch miss-prediction occurs.

1.1.4 Vector Instructions

We saw several mechanisms used to leverage processor performance by increasing instruction throughput *via* instruction parallelism. But processors can even do better: they can use data parallelism to increase their computational power. Indeed, compute intensive code often expose data parallelism, *i.e.* the same operation is applied to several independent data. Multimedia applications and linear algebra codes are good examples of compute intensive software. For instance when computing the sum of two vectors, multiple sums of corresponding elements of the vectors can be performed at the same time.

For this reason, processors now feature vector registers. A vector register can hold several values. Instructions operating on it perform the same operation at the same time on every element. Figure 1.4 illustrates a vector instruction.

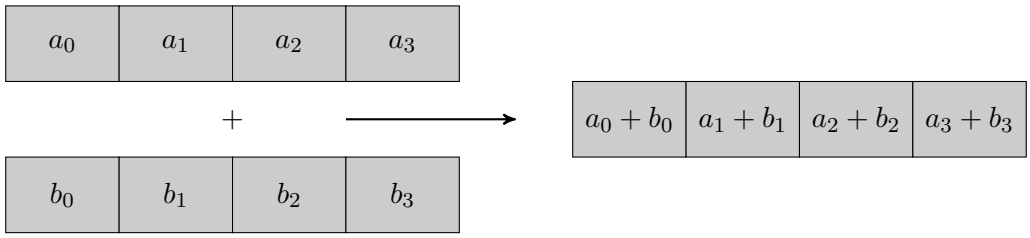


Figure 1.4: A vector instruction performing 4 additions at the same time on 4 elements of two distinct vectors.

The `MMX`, `SSE`, and `AVX` extensions are actually vector instructions added to the `x86` instruction set. PowerPC architectures feature AltiVec instruction that are vector instruction too. `MMX` instructions operate on 64 bits wide registers, `SSE` on 128 bits and `AVX` on 256 bits registers. Depending on the size of one element, one single instruction can perform up to 32 arithmetic operations at the same time (*e.g.,* an `AVX` addition will perform 8 operations on 32 bits wide elements, but only 4 if the elements of the vector are 64 bits wide).

1.1.5 Low level Code Optimization

In order to exploit the hardware computational power, software have to be well adapted to the architecture. Code optimization allows to perform the same computation faster by better tuning the software to the underlying hardware. The compiler is responsible for producing efficient machine code from its input in a higher language. For the compiler to produce fast code, a large range of optimizations are available. These optimizations can be combined to achieve mode efficient code. Combining optimization methods is also used in other domains where performance matters [58]. But finding the right set of optimization to apply to a particular program is non-trivial, and numerous research have be led on this particular topic [1, 18, 33, 82]. The instruction scheduling phase of compilation is responsible for scheduling machine instructions after the instruction selection phase. Instruction scheduling assigns an order to instructions in such a way that *i)* dependencies between instructions are not broken, *ii)* optimization constraints. These optimization constraints can be to provide faster code, lower register pressure, *etc.* In order to provide faster code, instruction latencies have to be overlapped. Therefore instruction latencies is a key information to provide to compilers. It is known that, with enough hardware information the optimal instruction scheduling can be achieved [10, 53, 84].

Therefore, instruction performance is an important information for compiler to produce efficient code. Information about hardware feature of the CPU can be found in hardware documentation [44]. However execution ports used by instruction, their latencies, throughput and execution port are harder to find out. Agner Fog provides a large amount of information about instruction performance [31]. He discovers this information by running experiments: benchmarks for each instruction to provide information about instruction performance to the community. Framework dedicated to benchmark writing can also help retrieving such information [80]. Until now no fully automatic method is available to get these information. The goal of one of our contributions is to provide insight to automatically obtain critical information to build hardware model. This will be discussed in Section 2.2.

Instruction scheduling is usually made at the basic block level of the program. A basic block is a piece of program where only the first instruction can be the target of a branching instruction: there is no other entry point into a basic block than the first instruction. A basic block does not contain any jump or conditional branch expects for the last instruction. In the execution flow of a program, a basic block is thus always either entirely executed, or not executed at all. Hence instruction scheduling is limited by the scope of a basic block and they are only a few alternatives for shifting instructions in small basic blocks. Small loops (with only a few instructions) usually leads to small basic blocks. In order to provide more search space for the compiler to select a better instruction scheduling, loop unrolling can be used to transform loops with a small body.

Loop Unrolling

Unrolling a loop consists in executing several loop iterations as a single one with a bigger body. Table 1.4 shows an example of loop unrolling. We can see on the right hand side of the table that one single execution of the loop computes the sum of four elements of the array t. The loop is unrolled by a factor of 4. The machine code corresponding to the unrolled loop will therefore contain 4 times more instructions than the initial code. This will give more freedom to move instructions around to avoid stalls due to dependences.

Table 1.4: Loop Unrolling example. For brevity we omitted the tail code when N is not a multiple of 4.

Before Loop Unrolling	After Loop Unrolling

`s = 0;`
`for(i=0; i<N; i=i+1) {`
`  s = s + t[i];`
`}`
	`s = 0;`
	`for(i=0; i<N; i=i+4) {`
	`  s = s + t[i];`
	`  s = s + t[i+1];`
	`  s = s + t[i+2];`
	`  s = s + t[i+3];`
	`}`

An other reason why loop unrolling improves software performance is that for each loop iteration a condition has to be checked. Instruction used to perform this versification are just an overhead: they are only needed to control the program flow but not for real output computation. As an n-unrolled loop will perform n times less iterations than the unrolled version, it will reduce the overhead due to conditionals by a factor of n as well as the number of branch taken.

Loop unrolling is very well handled by compilers since it is a really simple code transformation. However the hard part of automatic loop unrolling consists in finding the optimal unroll factor of a loop. Indeed several factor affect loops performance: if the loop is not unrolled enough, the compiler might not find the best instruction scheduling due to the lack of instruction in the loop body. But unrolling too much a loop can lead to too big loop body preventing the processor to use its instruction loop buffer. Automatic methods exist to overcome this issue: for instance auto-tuning based optimization will solve this problem by generating several loops with different unroll factors and compare all of their execution. However one has to be careful when using auto-tuning to select the unroll factor of loops. Since nested loops can be unrolled and jam the combinatorics of auto-tuned nest loop optimization can become very high.

Loop unrolling helps the compiler to better schedule instructions, but it can also help the compiler optimize even further the code: since an unrolled loop will present more arithmetic operations, the compiler can even try to use vector instructions to perform all of them at once. This is called code vectorization.

Code Vectorization

Code vectorization is a compiler optimization that tries to replace scalar operations with vector operations. In the code example shown in Table 1.4, since we perform 4 additions at the same iteration we can try to vectorize this 4 operations. Yet the 4 *add* instructions are not independent since they are all reduced to a single scalar. In order to perform all these *add*s at once we have to make these instructions independent. To do this we can split the sum of the array t into 4 partial sums. This is the code exposed is Table 1.5 After splitting the sum into 4 partial sums, the 4 *add*s do not carry dependencies between them anymore. Thus we can use a single vector instruction to perform all the operation at the same time. As all the 4 arithmetic instructions can be performed

Table 1.5: Vectorization example.

Before vectorization	After vectorization

`s = 0;`
`for(i=0; i<N; i=i+4) {`
`  s = s + t[i];`
`  s = s + t[i+1];`
`  s = s + t[i+2];`
`  s = s + t[i+3];`
`}`
	`s0 = 0; s1 = 0;`
	`s2 = 0; s3 = 0;`
	`for(i=0; i<N; i=i+4) {`
	`  s0 = s0 + t[i];`
	`  s1 = s1 + t[i+1];`
	`  s2 = s2 + t[i+2];`
	`  s3 = s3 + t[i+3];`
	`}`
	`s = s0 + s1 + s2 + s3;`

with one single instruction, we can further unroll the loop to let more space for the compiler to schedule instructions.

This section ends the description of single core architecture. We saw several hardware features allowing great performance gains. But single processor machines do not provide enough computational power to sustain compute intensive numerical simulation, architectures have switched from single core processors to multi-core processors to increase even further their performance.

1.2 Towards Parallel Architectures

This section briefly describes parallel computer architectures. In a first section we describe the main motivations for increasing hardware parallelism to achieve better performance. The next sections present several different levels of parallelism within computer architectures. We present parallel designs by growing granularity. Starting from parallelism embed on the CPU itself, with multi-processor and simultaneous multithreading. Then we present parallelism available outside of the processor itself with accelerators. Eventually we present coarse grain parallelism with architectures dedicated to HPC such as clusters.

1.2.1 The Energy Wall

When aiming at increasing processor computational power one has two alternatives: either increasing the processor speed (*i.e.* frequency) or increasing the number of instructions that it can execute in one clock cycle. Both these methods have a drawback: they increase CPU power consumption.

To add new features to the hardware processor vendors have to increase the number of transistors on the die. Since each new transistor has to be powered, it increases the chip electrical needs.

In the same manner, boosting the processor frequency raises its consumption. But worse with heightening the frequency: it increases heat dissipation. The heat produced by a processor is

proportional to its frequency: increasing its frequency by a factor two leads to doubling power dissipation. And worse: when processor frequency increases, the voltage has to be increased too. This avoids hardware errors by augmenting the electric signal strength. And the power dissipation is proportional to the square of the voltage. The power consumption P of a CPU is approximately: $P = c \times f \times V^2$ where c is a constant, f the frequency and V the voltage of the CPU.

In order to keep the processor cool, we have to set up cooling systems. These systems need a lot of space since heat transfer depends on the surface. Up to half of the space of many machines is therefore actually dedicated to cooling.

For these reasons, single processor performance could not be further enhanced. A new way to improve performance is build computer with several processing units. The following sections will describe some parallel architectures featuring multiple CPUs.

1.2.2 Multi-Processor

Multi-Processors systems are computers equipped with several identical processors. Processors are connected by the mean of a bus on the motherboard to the same shared main memory. Each of these processors can be dedicated to different tasks. This is called SMP for *Symmetric Multi Processor*.

Another kind of Multi-Processors systems are CMP for *Chip Multi-Processors*. On this kind of hardware systems, processors sharing the same chip also share some resources such as a level of cache. This is a more complex hardware hierarchy than SMP systems since it can lead to contention when cores are trying to access the same shared resource. However this can also increase communication efficiency between cores sharing a level of cache. This can spare some resources and space on the chip, allowing processors to have more cores.

Figures 1.5 and 1.6 illustrate the concept of SMP and CMP. We can see that CMP systems are more hierarchical than SMP ones. One has to be careful when writing programs targeting CMP architecture since communications costs is not the same between two cores located on the same chip and two *distant* cores. We should emphasis that things have moved to some private resources and some shared. For instance on most modern processors some caches are privates and others are shared.

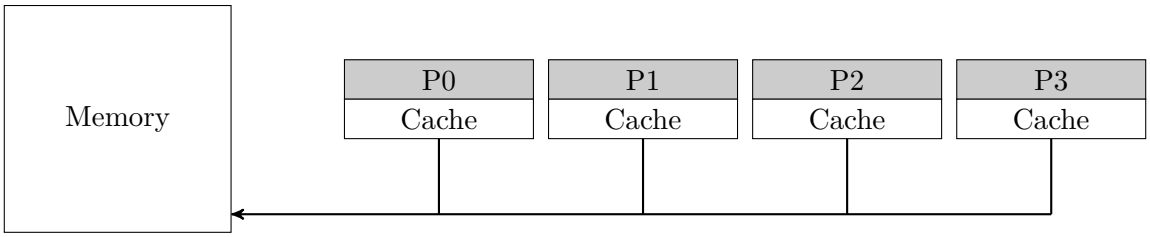


Figure 1.5: A SMP Multi-Processor System: 4 processors connected to their shared memory.

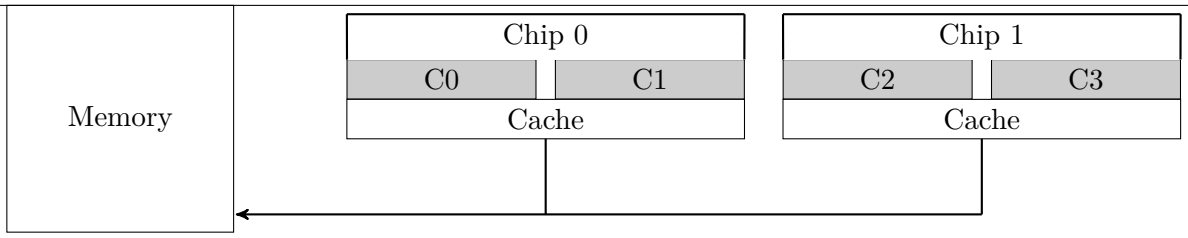


Figure 1.6: A CMP: 2 chips, each chip made of 2 cores sharing hardware resources such as a cache level.

1.2.3 Simultaneous Multithreading

To even further spare hardware resources one can extend feature sharing to lower levels (*i.e.* closer to processor). This is the called *Simultaneous MultiThreading* (SMT): the hardware threads are processing units located on the same core. But all functional units are not duplicated. Only registers (both general purpose and special registers) are duplicated. But the pipeline, the *Arithmetic and Logic Unit* (ALU), *etc* are shared.

Hardware threads can execute independent instruction flows, or programs. Therefore it does not improve hardware peak performance since different threads sharing the same functional units cannot perform arithmetic operation at the same time. Still it can improve the pipeline utilization by filling the bubbles inserted into the pipeline by one of the thread with instruction from the other thread. Simultaneous Multithreading only increases hardware sustainable performance.

Schedulers can be aware of the hardware threads (and in particular that they shared some functional units) and can therefore better balance the load across the system [14].

1.2.4 Accelerators

In the last section we described several ways to improve general purpose processor performance. In order to perform more specific tasks, specialized processors can be used. Specialized processors are called accelerators.

Chips dedicated to one single kind of task can skip all features not compulsory to carry out their job. This makes room on the die for more functional units dedicated to the task of the accelerator. Since accelerators are becoming more and more present in the HPC field, which is the area of the dissertation, we decided to describe some on them that are often seen in published work. However accelerators architectures are beyond the scope of the contribution of this dissertation since we do not try to model them.

Graphics Processing Units

Graphics Processing Units (GPU) were initially designed for graphics rendering. Yet GPUs are very efficient for SIMD computation. Since HPC heavily rely on this kind of computation, GPUs

are used in many super-computers dedicated to simulation. GPUs are composed of many simple processors dedicated to arithmetics. GPUs are therefore very efficient for embarrassingly parallel computation: every processor executes the same instruction, but each of them on different piece of data. They can embed their own memory (*e.g.,* discrete graphic cards), data can be transferred to and from the GPU memory through a *Peripheral Component Interconnect* (PCI) bus. Compute intensive tasks can be offloaded to the GPU, freeing the CPU from this task, and letting it executing some other tasks.

Cell

The Cell processor was initially released by IBM, Sony, and Toshiba for the The PlayStation 3 game console (PS3) [21]. It features a general purpose CPU: a PowerPC processor called the Power Processor Element (PPE). This PPE is surrounded with between six and eight accelerators, *Synergistic Processing Elements* (SPE). The PS3 feature six SPEs while Cell processors released for HPC platforms feature eight. SPEs feature vector instructions for fast arithmetic processing. They have a private fast memory and are connected to other SPEs by a ring bus.

Single-chip Cloud Computer

The Single-chip Cloud Computer (SCC) released by Intel is a many core architecture. It features 48 cores embed on the same die. These cores are organized on 24 tiles connected through a 2 dimensional mesh. Cache coherency of the SCC is handled by software. Intel provides an Application Programming Interface (API) to program the SCC that handles cache coherence automatically. This is an interesting approach since, as we will see later in this dissertation, hardware managed caches can present some difficulties for performance modeling as well as scalability issues. The SCC is not designed to be an accelerator, a Linux kernel runs on each of the cores[4]. It is more of a distributed platform embed on a chip. Common distributed platforms will be presented later and focus on large scale. We choose to present the SCC in the section dedicated accelerator because of the scale of its architecture, that is closer to accelerators than to clusters.

Xeon Phi

Intel recently released a new kind of accelerator: the Xeon Phi. The Xeon Phi processor family that was released in 2012. This processor implements the idea of integrating many core on the same chip. This board can be connected to the motherboard *via* a PCI bus. The Xeon Phi is a massively parallel chip embedding up to 61 processors with large vector registers and instructions (512 bits). As for GPUs, compute intensive tasks can be offloaded to the Phi processor.

1.2.5 Clusters

Until now we presented features raising performance of a single computer, either by leveraging core throughput or by increasing parallelism. In order to run large computations, one single machine

[4]Yet, a bare-metal mode can be used run software on the cores without a running operating system.

is usually not enough. To deliver more computational power, computers can be bound together by networks and perform massively parallel computation. Computers linked together to perform scientific computation are called clusters.

Most of the top500 [79] machines are actually clusters. Because general purpose machines are cheap, one can interconnect many of them together to increase the computational capability of a system. For these system to work properly, each computer has to be able to communicate with the others. The more nodes[5] are present in the cluster, the more communications have to be efficient. Since network is much slower than memory, waiting for data to be transferred between node can dramatically decrease performance of parallel applications. In order to improve communication efficiency high performance networks such as InfiniBand [39] were developed. Efficient communication strategies as well as placement are often investigated to improve communications efficacy [30].

1.3 Memory Architecture

We saw several methods to increase computational power of modern architecture. But for the computation to carry out, it needs data to operate on. As CPUs do not have enough registers to store all the accessed data inside the processor, they are connected to the memory where data can be stored when no instructions is using it. Memory is much slower than the processor. Therefore, accessing it is critical to keep CPUs fed with data to operate on. Since arithmetic instructions usually have a relatively small latency, with an efficient instruction scheduling, either by the compiler or thanks to the out-of-order engine, compute intensive programs are usually able to utilize the pipeline very efficiently. But memory accesses will stall the pipeline even with an efficient instruction scheduling. A load instruction that brings a piece of data from memory to the processor can be up to 200 times longer than an arithmetic instruction[59].

This section is dedicated to the memory organization of computer architecture. We will describe which features were added to existing hardware in order to speed up memory access or to hide memory latencies. We will first present how software and the operating system access physical memory in Section 1.3.1. Section 1.3.2 focuses on modern processor memory organization. Section 1.3.3 presents caches, a hardware feature designed to hide memory latency. Finally, Section 1.3.4 presents a few caches architecture without hardware managed coherency.

Memory hierarchy is a critical part of computer architecture, especially in the context of HPC. Indeed improving processor performance is useless if memory performance is not increased at the same time: how fast a processor can compute does not matter if it constantly has to wait for memory. The memory wall is a concept explaining why memory performance is so critical to computer performance.

The Memory Wall This concept was formalized in 1995 [86]. It explains why memory performance is becoming such critical parameter for performance. Considering a cache hierarchy with a perfect cache with a t_c cycle latency and a RAM memory module with a latency of t_m, the average access time to memory is: $t_{avg} = p \times t_c + (1 - p) \times t_m$, with p the probability of a cache hit. Also

[5] In the context of clusters, nodes refer to computers.

since the cache is often on core t_c is close to 1 (1 clock cycle). Since memory performance grows slower than CPU performance, t_c and t_m diverge. This means that t_{avg} grows at the same time. No matter how fast caches and processors are, the average access time to main memory will grow.

As long as memory performance cannot match the processor performance, accessing memory will be remain critical to performance. Moreover with the appearance of vector instructions and the increasing number of core, an increasing pressure is put on the memory. Before going through some hardware features designed to increase memory performance, we have to explain how programs and the Operating System (OS) access memory.

1.3.1 Virtual Memory and Translation Lookaside Buffer

When software needs to access memory, instructions have to provide the memory address they want to access. In modern operating systems, physical memory is virtually divided into separate address spaces. Each program – or process – running on the machine has an address space dedicated by the operating system for storing its data. This allows several interesting features such as memory protection: a process can only access its own address space separating it from the other programs. Also virtual memory can virtually extend memory available on the machine: if the machine runs out of memory (physical memory) the operating system can choose to write physical pages to the hard drive to free them and allow other processes to use newly available memory pages. Of course reading and writing memory pages to the disk is slow and should be avoided, yet it allows computers to work on larger data sets than the physical memory.

When accessing memory, the software provides the CPU with the virtual addresses they want to access. The processor and the system are then responsible for the translation of the virtual addresses to the corresponding physical addresses. Figure 1.7 illustrates virtual to physical memory mapping. The system keeps a page table for each process where it stores the mapping between the process virtual memory pages and physical memory frames. Since this table is stored in memory, translating virtual memory would be very inefficient if no hardware would speed this translation. In order to speed up this translation process, the *Translation Lookaside Buffer* (TLB) is a very fast memory location where address mappings are kept after each translation. Since this is a limited memory, the operating system – or whatever piece of hardware – has to choose what mapping to store in the TLB. When an address translation is needed and the translation is already in the TLB the mapping stays in the TLB. If no translation can be found in the TLB the system or the hardware *Memory Management Unit* (MMU) has to do the translation by reading the page table. The translation that has just been performed is stored in the TLB. If the TLB is full, a translation has to be evicted out of the TLB to make room for the new entry, the *Least Recently Used* (LRU) entry is usually selected for eviction.

We briefly saw how the operating system and the hardware collaborate to access memory, we can now go back to our main concern: optimization, focused on memory performance. We will describe features that are important for understanding the contribution of this dissertation, but more details about memory hierarchy can be found in literature [28].

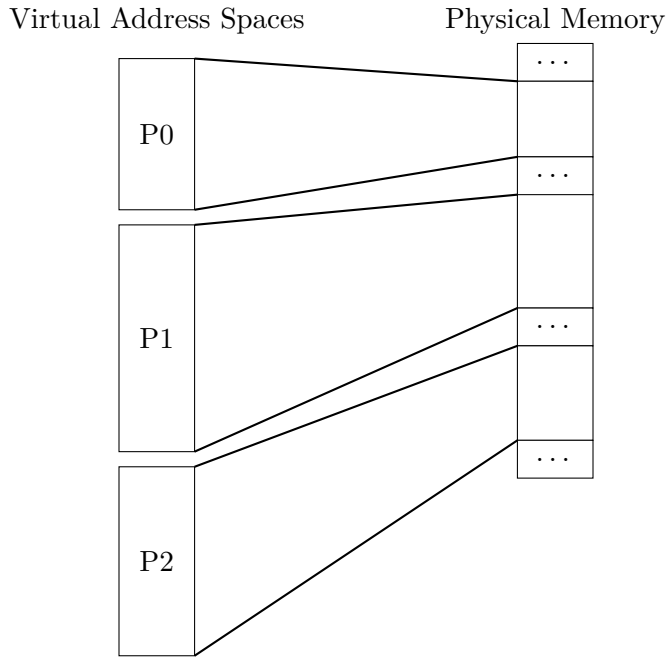


Figure 1.7: Virtual and Physical memory mapping.

1.3.2 NUMA Architectures

In order to build parallel architectures one has to be able to sustain high memory bandwidth to avoid stalling processors by waiting for data. The main problem in accessing memory is contention on the shared bus when several processors or core are reading or writing to memory. Figures 1.5 and 1.6 illustrating multi processor systems show the problem: each processor needing memory access has to use the same bus as the others. This leads to contention and each processor has only access to a fraction on the full memory bandwidth of the architecture.

NUMA architectures address this issue by partitioning memory in several chunks called memory banks. Each bank is directly linked to a subset of processors. A memory bank and its connected processors is called a NUMA node. NUMA nodes are interconnected through an efficient interconnection bus. Since processors access memory on their own NUMA node faster than memory on external NUMA nodes, the access to memory is said to be not *uniform*: memory latency depends on the memory bank that has to be accessed to fulfill the memory request. This is why these memory architecture are called NUMA for *Non Uniform Memory Access*. Figure 1.8 illustrates a NUMA memory architecture.

When processors access memory on their node, no traffic has to go through the interconnect, this can reduce the traffic on the interconnect. However poor data placement among memory banks can lead to contention on the interconnection bus. One has to carefully allocate data on local memory banks to minimize the traffic outside of the socket.

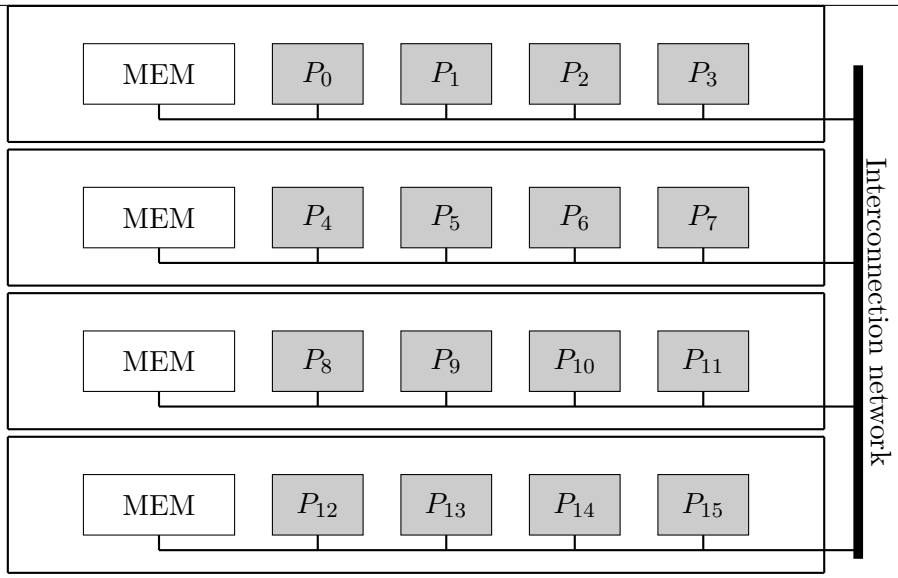


Figure 1.8: A NUMA Memory Architecture.

1.3.3 Caches

If one is careful with data allocation, high memory bandwidth can be achieved thanks to NUMA architecture. Yet this is not enough to reduce instruction latencies due to memory accesses. Fast memories were added into memory architectures to speed up data access. One fast memory used in almost all general purpose CPUs are caches. Since the main contribution of this dissertation focuses on modeling memory hierarchies, the next section will present cache architectures in details.

Caches are very fast memories that can be embedded onto the CPU die. However to keep caches fast and to limit the cost of CPUs, these memory have to be small, much smaller than the computers main memory. Therefore the entire data set of software cannot fit in cache, and one has to wisely choose what to put into the cache to achieve better performance. Also most caches are completely implemented in hardware, and software has no control over it. Choices made in cache design are therefore critical: it has to be efficient – or at least avoid degrading program performance – for all kinds of code. The next section will describe caches architecture and hierarchy.

Cache Architecture

Caches can be seen as a large array. Each line of this array is called a cache line. Cache lines are usually relatively small (64 bytes on most of `x86` architectures). A cache line contains a copy of a piece of data from main memory, a tag containing information about the address of the data stored in the cache line, and some flags. When the processor needs to access memory, it first asks the cache if the address to be read or written is already in the cache. If it is in the cache, then there is no need to go to memory. The cache provides the CPU with the data it requested, this is called a *Cache hit*. But is the cache does not hold the data requested – this is a *Cache miss* – main memory has to be accessed to retrieve the piece of data requested.

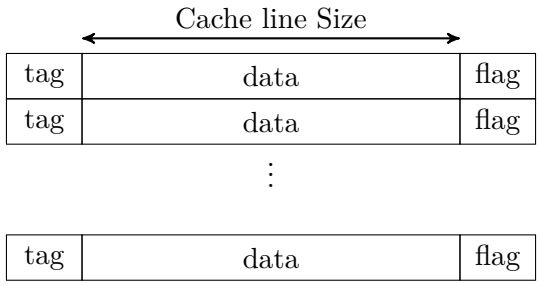


Figure 1.9: A cache illustration.

Spacial and Temporal Locality When a program accesses a data it will usually access it again soon. It is called *temporal locality*. When a cache miss occurs (*i.e.* an address not present in the cache is requested by the processor) the cache chooses a cache line where to store the data, fetches is from main memory and stores it in the selected cache line. Hence the next time the same data will be requested it will already be in the cache: this is a cache hit. If the cache is already full it has to free a line. Usually the LRU cache line is flushed out of the cache. When data is read from memory it goes into a cache line. Since a cache line is larger than one single element, some other elements next to the requested one are loaded in the cache as well (the granularity of all transfers to and from the cache are a cache line). This helps taking advantage of what is called *spacial locality*. This concept says that when software access a piece of data it will also access data located close to it. Therefore when a full cache line is loaded because of the access to a single element of the cache line, we can expect to soon access the other elements of the cache line. Hence we avoid cache miss by loading a full cache line instead of a single element.

Cache Associativity In order to keep accesses to the cache fast, we need an efficient way to check if an address is or is not present in the cache. If every address can go into every line of the cache, the cache will have to check for every line if the address stored in the line is the one requested. These caches are said to be *fully-associative*. But checking whether an address is present in a full-associative cache is expensive.

Cache designers usually build a hash function based on the address giving the exact cache line number where the data should be – or go if not yet in the cache. Hence there is only one single location to check to know if the address requested is in the cache or not. Cache where each address can go to a single cache line are called direct mapped. But it might happen that a program accesses many addresses that all go to the same cache line, in this case the cache will not be able to use all its cache lines and a lot of space would be wasted. This kind of cache misses are called conflict miss: every address is competing to get into the same cache line flushing the former one that will have to be fetch from memory again after. In order to avoid this problem one as to choose a good hash function that will dispatch addresses into all cache line avoiding conflict miss. However such a hash function cannot be found for all possible programs. A trade-off is to build associative (but not fully-associative) caches. A n-way associative cache is a cache where every address can go to n different cache lines. When a checking if an address is in an n-associative cache, there is only n locations to check. It is slower than for a direct mapped cache but it reduces conflict misses.

Cache Hierarchy Since smaller caches are fast, small caches are a great way to speed up memory access in case of many cache hits. But smaller caches mean more cache misses since less data fit in it. If we choose bigger caches, we can achieve better hit/miss ratio because more data can fit in the cache, but the cache will be slower. We can get the best of both worlds by building cache hierarchy. A smaller (thus faster) cache can be connected to a larger one, itself connected to another bigger one. When reading memory, the processor can check if the address is in the first cache (called L1 cache), if it is, then the access will be fast. For instance the L1 cache of Sandy-Bridge processor are 32 kB wide and the time needed to access it is 1 to 2 CPU cycles. If the address is not in L1, it will check if the address is in L2 (the second cache, which is larger) *etc* until the address is found in a cache level or that all cache levels have been checked. The L2 cache on Sandy-Bridge processors are 256 kB wide and the latency is 12 cycles. And the L3 is shared among all processor of the same socket, it is 20 MB wide and its latency is 26 to 31 CPU cycles[6]. In hierarchical caches when a cache line is loaded from memory for the first time it goes to the first level of cache. Depending on the cache design it can also be written in higher cache level or not. A cache level is said to be inclusive if all data in lower caches are also in this cache. It is said to be exclusive if a data in a cache level is not present in lower cache levels. And it is said to be non-inclusive otherwise. Cache hierarchies can be complex: for instance Intel's Nehalem and Sandy Bridge cache architecture feature a inclusive L3 cache (it includes all data in L2 and L1 caches) and the L1 and L2 caches are non-inclusive: data in L1 may or may not be in L2. In these two micro-architectures, the L2 is a victim cache of the L1. This means that a cache line only goes to L2 when it is flushed out of the L1.

Cache Flags The flags of a cache line contain information about the state of the cache line. We will see more details about it in a later section dedicated to cache coherence. For now, we will only keep in mind that these flags tell whether the cache line is clean *i.e.* the copy in main memory is the same as the one in the cache, or if the line is dirty: the cache line holds a version of the data that is different from the one in main memory. This happens when the processor writes data: it is written into the cache but not in main memory to save memory bandwidth. But the cache line will have to be written to memory as soon as the cache line is flushed out of the cache. Yet this still saves memory bandwidth since data can be written several times into the cache before having to be written back to main memory. This strategy is called *write-back*, since data are only written back to memory when needed. In contrast *write-through* caches write data into the cache and into main memory as soon as the write occurs.

Section 1.3.1 has presented virtual and physical memory addresses. Also we saw that for caches to work we need to keep in every cache line the address of the data stored in it. Cache design has to choose either to put the physical or the virtual address in the cache line information. The great advantage of tagging cache lines with virtual addresses is that it does not require to wait for address translation to know if a cache access is a hit or a miss. However when the OS switches the process being executed on the processor, it has to flush the entire virtually tagged cache. In order to keep the cache requests fast and avoid flushing the cache after context switch, cache can be virtually indexed and physically tagged. This means that the cache line (or set) where a virtual address should go is determined by its virtual address, but the tag in the cache line is the physical address the virtual address maps to. In this cache design, looking for an address in the cache can be

[6]The shared L3 cache of the Sandy-Bridge processor is organized slices connected through a ring. Depending on the location of cache line accessed the latency and the core requesting it, the latency can vary.

done in parallel with the address translation. But the cache will only decide if it is a hit or a miss after the virtual to physical address translation. Since the tag holds the physical address, there is no need to flush the cache after a context switch: if the two process have the same virtual address mapping to different physical pages, the cache will make the difference between them thanks to the tag that will be different.

However in such caches if several virtual addresses map to the same physical address, the same data can be stored in several location of the cache. This is called cache aliasing. In order to keep memory consistency, explicit cache flushing has to be done when such cases happen. This cache aliasing problem is avoided in Linux kernel by carefully choosing the virtual address of shared pages: all the aliased addresses are given to the user so that they all will go to the same set. Therefore the cache is tagged with the physical address which is the same, even aliases of the same physical memory will go the same cache line.

Instruction Cache Since programs are stored in memory, reading the instructions can be slow and lead to poor performance if the processor has to wait for memory to decode the instruction. In order to avoid memory latencies not only for data access but also when reading instructions, program code can be in caches. Cache design for instructions can be simpler than for data: we do not need tags to know if a cache line is dirty or not: the instruction of a program are not supposed to change after being loaded to memory. Therefore in an instruction cache all caches line are always clean: we can save the tags bits. In most of the general purpose processors, they are two L1: L1d for level one cache for data and the L1i the first cache level for instruction. And the other cache levels are unified: they contain both data and instructions.

Cache Coherence

We saw in Section 1.2.2 that modern architectures feature several cores or processors. On a parallel processor several processes or threads can access the same data set, these data sets are said to be shared. Since each processor has a private cache it is important to keep memory consistent when several execution threads access shared memory. For instance if a data cell is updated by a thread and then read by another is it important that the latter access provides the CPU with the correct value for the variable. In order to maintain memory consistency, cache coherence protocols were added in to cache.

These protocols are a set of rules to be applied when read or write to the cache occur. The next section will describe some cache coherence protocols. Figure 1.10 illustrates a cache hierarchy with several cores sharing some caches and with other caches that are private. Each core has its own level 1 cache, the level 2 caches are shared by pair of cores and the last level of cache is shared by all 4 cores on the chip.

Cache coherence protocols

In order to maintain coherence, the common solution is to add some bits to the cache line flags. These bits are used to represent the state of the cache line, *i.e.* if it is clean or dirty and information about. A protocol defines the actions to be taken when cache events occur. We will now present some coherence protocols to illustrate the idea.

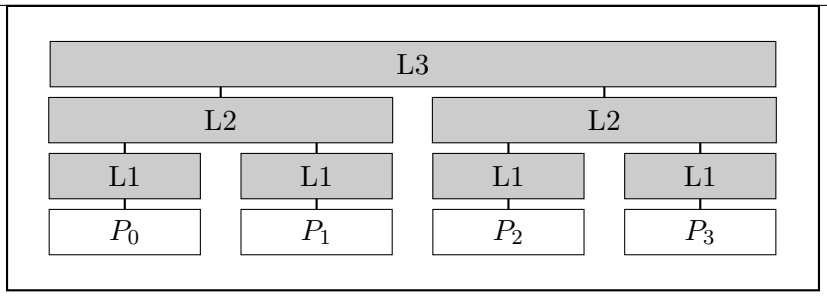


Figure 1.10: A parallel cache hierarchy.

MSI The simplest coherence protocol is the `MSI` protocol. In this protocol cache lines are tagged with one of the M, S or I tags. The meaning of this tags are:

M for Modified: this means the cache line has a newer version of the data than the memory. Only one cache can hold this address in the cache.

S for Shared: the line is clean. Several caches can hold the same address, all of them will have the corresponding cache line S state.

I Invalid: no valid data is stored in the cache line.

The protocol defines actions to be performed on cache events. Cache events can be local read or write and requests posted on the bus connecting caches. The `MSI` protocol can be implemented as a snooping protocol. This means that caches have to monitor traffic on the bus. Figure 1.11 is a graphical representation of the `MSI` protocol that can be defined with the following actions:

- When a cache hit happens the cache can satisfy the request itself. If the request is a write to a shared cache line, an invalidation request is broadcasted on the bus. All other caches holding the same address will discard their copies, and the cache writing the cache line will set the cache line state to modified.
- When a read miss occurs th processor checks if the line requested is present in another cache.
  - If another cache holds the requested data in a shared state, it will send the cache line over the bus.
  - If another cache holds the data in modified state, the remote cache writes its line back to memory and either sets the line in shared or invalid state (this depends on the design). The cache that issued the bus request gets the cache line either from the bus or from the memory, the state of the cache line is shared.
  - If no other cache has the data, it has to be brought into the cache from the main memory. The cache line will be set in the shared state.
- When a write miss occurs, the cache has to issue a *Request For Ownership* (`RFO`) for this address on the bus. Caches snooping an `RFO` on the bus will have to invalidate their cache lines that hold the address. If a cache holds the address in the M state it has to write it back to memory before invalidating its cache line.

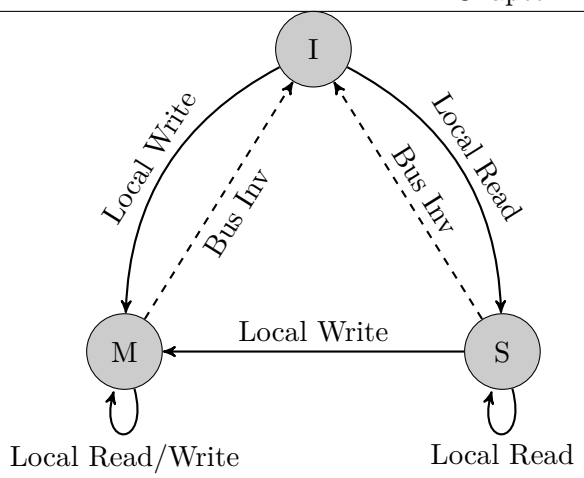


Figure 1.11: The MSI protocol.

MESI One of the weaknesses of the MSI Protocol is that when a single cache holds a cache line in the S state, it still has to broadcast an invalidation request to write to this cache line. An optimization would be to avoid this broadcast by knowing that the cache is the only one holding the given address. The MESI protocol brings this optimization to the MSI Protocol. It was developed at the University of Illinois [64]. It adds a state called *Exclusive* (E) which means that the cache line is clean and is the only copy in the cache hierarchy. It is like the S state of the MSI Protocol excepts that the cache holding a cache line in the E stats does not have to broadcast an invalidation before writing to this cache line. When an address is brought to the cache from memory (and not from another cache) the cache line is set to state E. This is a very useful optimization because writing to exclusive data happens very often in software. The most common usage is for instance incrementing a variable.

MESIF Another weakness of the MSI and MESI protocols is that when several caches hold the same address (therefore in S state) all of them will respond to a bus request asking for this cache line. This leads to redundant traffic on the bus. The goal of the MESIF Protocol is to avoid this unnecessary traffic bus by adding the *Forward* state (F). A cache line in the F state will behave almost like in the S state. The only difference is that only the cache with the line tagged F will respond to the requests, not the ones with the S state. This is the protocol used in many Intel processors such as the Nehalem and Sandy Bridge processors.

MOESI In coherence protocols such as MSI, MESI, and MESIF protocols, a lot of time can be lost when reading remotely modified cache lines since, in these protocols, modified cache lines have to write to memory before being read by another cache. Some cache coherence protocols allow sharing of a dirty data. In the MOESI Protocol the states M, E and I have the same semantics as in the MESI protocol but the shared state can be dirty. If every cache holding the same cache line are in the S state, then the data is clean. But if one cache has this line in the O state (which is specialization of the S state meaning *Owned*), then the data is dirty. The cache holding the copy tagged O is

responsible for responding to bus read request in this cache line. Modification to the `MESI` protocol to benefits from the O state are the following:

- A cache is responsible for responding to bus read request on cache lines in M and O states. After a response to such a bus request the state of the cache line is set to O state (unchanged if it already was in O state).
- Only a cache line tagged O can respond to bus read request, this is one drawback of this protocol: clean shared cache lines cannot be shared through the bus but have to be fetch from memory.
- In order to write to a Shared line the cache has to broadcast an invalidation for this cache line to all other caches. After writing to the cache line it will be in the M state (since all other copies were invalidated).
- Cache lines in O state are responsible for writing their content back to memory when they are flushed.

The `MOESI` Protocol is used in some AMD processors such as Bulldozer architecture.

Firefly Until now we only saw cache coherence protocols with a write-back policy, meaning update to the memory is only performed when necessary. But some coherence protocols use a write-through policy meaning that when a data is written to a cache, it also goes to memory. On the one hand write-back policy allows for easier coherence protocol design since memory is alway updated with the last write. On the other hand each write on such coherence protocols have to go to memory which increases memory usage. The Firefly protocol use both write-through and write-back policy to avoid too much overhead due to using write-through policy on every write. In the Firefly protocol each cache line can be in one of the following states:

- Valid-Exclusive: this is same meaning as Exclusive in the `MESI` protocol. The line is clean and only the cache holding this cache line has this address in its cache.
- Shared: the cache line is clean and several caches may hold the address contained in the cache line. This is the same state as S in the `MESI` protocol.
- Dirty: the cache lines is dirty and is the only copy of the data (same as state M in the `MESI` protocol).

The goal of the Firefly model is to avoid as much of cache line invalidation as possible.

- On a load hit, the cache provides the requested data.
- When a load miss occurs, the request for this address is sent over the bus.
  - If another cache can respond, it will provide the cache line.

    * The cache providing the cache line has to write the cache line back to memory if it was dirty (by writing the data to memory it makes it clean).

  - If no cache has the requested cache line, it has to be fetched from memory, the cache line will be set to the Valid-Exclusive state.

- On a store hit:

  - if the cache line is in Valid-Exclusive state, it is updated and the state goes to Dirty.
  - if the cache line is in Dirty state it can be updated and the state of the cache line does not have to change.
  - if the cache line is shared, the cache updates the cache line and also writes the data to memory (write-through). It also has to broadcast the new value of the cache line on the bus for others cache to update it. Since the cache made a write-through the memory is also updated and the data is not dirty.

Dragon The Dragon protocol is similar to the Firefly protocol except that it allows sharing of dirty cache lines. This avoids the write-through when a store to a shared cache line happens. In order to achieve this, a state is added to the three states of the Firefly protocol: the shared-dirty state. Cache lines are set to this state after they are updated by a broadcast on the bus due to a store hit on a shared cache line (either shared-dirty or shared-clean). This avoids the compulsory write-through of the Firefly protocol.

Summary We saw that many coherence protocols can be used to maintain memory coherence, these mechanisms are implemented into the hardware. Most of them involve non scalable communication such as broadcasts. This means that maintaining cache coherence becomes harder and harder as the number of processors and cores in the system increases. In order to reduce the number of unnecessary coherence messages, directories can be added into cache hierarchies to keep track of which caches have a given line [3, 20]. Some say that cache coherence does not present such a big overhead – especially thanks to directory based coherence mechanisms [54]. More room for optimization regarding memory access can be achieved by delegating this task to the programmer. Moreover, since this hardware coherence is all done automatically by the chip, programmers – even those who are aware of cache coherence performance problems – have a restricted control over it. Some instructions allow controlling caches. For instance non-temporal instruction can be used to bypass some cache level, or explicit flush allow programmers to evict a particular cache line. Architectures without hardware cache coherence were released to provide the programmers with more control. The Intel's SCC and the Blue Gene/L chips are examples of such non-coherent cache architectures [22, 35].

Since legacy codes cannot be easily modified, software developed with the assumption that the hardware features a hardware coherent cache cannot be ported to architecture featuring software managed cache. Hardware cache coherence is therefore compulsory for legacy code.

1.3.4 Non-Coherent Caches

The main problems of hardware maintained cache coherence are that it cannot be adapted to the application. For instance one might want a coherence protocol for an application and another one for a different one: this can only be achieved by software coherence. To give an example, a parallel application with heavy communication would probably benefit from a cache implementing a broadcasting strategy when writing to shared cache line: the communication would be performed through the bus connecting caches instead of through main memory. However this same protocol would lead to high overhead with an application where data produced by a thread are not read by others.

Also on most applications just a few amount of data are actually shared: only these data need to be carefully maintained coherent between computing threads. For this purpose hardware coherence is too costly. Getting rid of hardware cache coherence can help reducing hardware cost. As well as allowing for larger number of cores on chip.

Maintaining Memory Consistent by Software

In order to maintain cache coherence on caches without hardware coherence the programmer has to add some instructions into the code to keep memory consistent. These instructions are responsible for invalidating stale data, or more generally handle the coherence traffic. It would be a mistake to think that it is deporting a piece of hardware into the software. Since the programmer knows what to keep coherent and what are the data sets used by every computing threads, it can reduce coherence traffic to the minimum actually required.

Software cache coherence can be almost transparent to the programmer – and thus enhance its efficiency – it is handled by a library or inside a compiler. For instance, Intel released the Single-chip Cloud Computer (SCC) in 2010 [22]. This architecture feature non coherent caches where the software coherence is ensured by a message passing library, avoiding programmers to get into too low level details [56].

Scratchpad

Another kind of fast memories can be used to reduce memory latencies: *scratchpads*. Scratchpad memories are fast memory modules where processes can store frequently used or critical data. It can achieve high bandwidth like cache memories, but software has control on what data to put into the scratchpad, contrary to traditional caches where all accessed data go automatically. Scratchpads can therefore avoid problems such as cache pollution. Cache pollution means storing into the cache data that will never be reused, they use cache space but the program never benefits from it. Another strength of scratchpads is that Direct Memory Access (DMA) engine are usually used to transfer data between main memory and the local scratchpad, which frees the CPU from this task. This is comparable to prefetching excepts that once a DMA transfer is initialized the CPU does not have to execute any extra instruction. While with prefetching special instructions are executed by the CPU to perform the memory transfer. Also interesting optimizations can be done thanks to DMA, one can overlap memory transfers with computations or realize prefetching in a very efficient manner. The Cell processor developed by Sony, Toshiba, and IBM [21] is an example of computer

architecture using scratchpads called *local store*. Each SPU has its own private 256kB wide local store and data can be moved to and from these local stores thanks to DMA engines.

The Cyclops64 project developed by the United States Department of Energy, the U.S. Department of Defense, IBM and the University of Delaware is another architecture using scratchpads to speed up memory access [40].

1.4 Summary

We looked over some architectural features of modern processors allowing for fast and parallel computations. But challenges await programmers with high performance needs since the more complex hardware is, the more difficult its efficient use.

Computer architectures in the HPC field trend to more and more parallelism as well as a growing heterogeneity: several different architectures can have to work together to carry out a computation. The appearance of GPU in clusters dedicated to intensive scientific computation is an example among other illustrating heterogeneity. Challenges for programmers are therefore, being able to produce efficient sequential code, express parallelism to utilize all computing cores available or even different machines connect *via* a network.

The contribution of this dissertation is help the understanding of modern hardware architecture through benchmarking. Modeling the core architecture requires knowing hardware features available on the processor pipeline (availability of a register forwarding mechanism, of an out of order execution engine *etc*), the number of execution ports and the latency of instructions. Hardware information are usually available in processor documentation, but instruction latency and throughput are harder to find. We will try to address this issue in Chapter 2.

Chapter 2 will present an automatic method to retrieve critical information to build hardware models. These hardware models can help automatic code optimization or code quality analysis.

Modeling memory hierarchy is even harder since because of some undocumented features – especially regarding cache coherence – that are included in the memory architectures. Cache coherence involves lots of automatic message exchanges that are hard to predict. Moreover the timings – or overhead – of these coherence messages are not documented and are hard to measure since these mechanisms are transparent to the programmer. Numerous automatic mechanisms embed in modern memory hierarchies are very efficient for general purpose usage. But less for fine tuned HPC applications, also taking this mechanisms into account when model hardware performance is difficult.

Chapter 3 will focus on bringing knowledge about memory architecture to programmers by mean of micro-benchmarking.

However we will see that even with a large amount of information about memory architecture, it is too complex to build a theoretical model that matches the reality precisely. We found an alternative in order to build memory models, it is to bring benchmark data into the model and build the model upon the output of benchmarks.

Chapter 4 aims at building a memory model for cache coherent architectures that is based on benchmarks instead of building a theoretical model based on hardware parameters.

Contents

2.1 Propostion **36**
2.2 On-core Modeling: Computational Model **36**
2.2.1 Related Work 37
2.2.2 A methodology to measure Latency, Throughput, and to detect Execution Port assignations 38
2.2.3 Detecting Instruction Parallelism 40
2.3 Case Study: Power Aware Performance Prediction on the SCC **46**
2.3.1 Related Work 47
2.3.2 The SCC Architecture 47
2.3.3 Performance Model 48
2.3.4 Model evaluation 51
2.3.5 Power efficiency optimization 54
2.3.6 Summary 56
2.4 Summary about On-core Modeling **56**
2.5 Un-Core Model: Memory **57**
2.5.1 Memory Hierarchy Parameters Needed to build a Memory Model 57
2.5.2 Cache Coherence Impact on Memory Performance 59
2.5.3 Bringing Coherence into a Memory Model 60
2.6 Conclusion **62**

Chapter 2

Performance Modeling

"The purpose of science is not to analyze or describe but to make useful models of the world. A model is useful if it allows us to get use out of it."
— Edward de Bono

Through the two last chapters we presented the state of the art of HPC field and identified several research challenges we chose to focus on. The next three chapters will now focus on the contribution of this dissertation.

The last chapter presented how HPC applications are developed and optimized. We saw that hardware modeling gives insight to the programmer about the hardware and therefore helps matching software to the underlying hardware. Also, precise hardware models allow automatic code optimization if they can be brought to tools such as compilers, runtime systems, or libraries.

Since new computer architectures are released at a high frequency to fulfill the growing need for computational power, hardware models have to be update very frequently too. The issues encountered when modeling hardware are mostly: *i)* getting enough information about the architecture

and *ii)* understanding how each hardware component interacts with each other. Since hardware is becoming more and more complex, hardware modeling is becoming a real challenge.

The contributions of this chapter are: a methodology to automatically measure instruction performance: latency, throughput and execution ports, this is described in details in Section 2.2. A performance model of the SCC architecture allowing power performance optimization, described in Section 2.3. And a study of the important parameters to be taken into account when trying to model cache coherent memory hierarchies, in Section 2.5.

2.1 Propostion

The difficulty to deeply understand modern hardware leads to building performance models to abstract the complexity of computer architectures to better utilize it. The contribution presented in this chapter aims at providing tools and methods to get information about hardware in order to ease building hardware models.

We choose to divide hardware models into two different parts: *on-core* and *un-core* model. We choose this classification because software are often also divided into 2 categories: compute-bound or memory-bound. Compute-bound software execution time depends on the speed on computation while memory performance is only marginal. On the contrary, memory performance is critical for memory-bound software. The on-core model section is related to features located on the core itself: the ALU, the instruction pipeline, *etc.* These models are important for understanding and optimizing performance of compute-bound software. The un-core part is related to features outside of the core: mainly memory and caches. Although level 1 and 2 caches are often physically on the core, we choose to include the modeling of the full memory hierarchy (*i.e.* all levels of cache and main memory) in the un-core model. Un-core models are used to predict or optimize memory-bound software.

We try to respond to the lack of architecture knowledge by presenting automatic methods to retrieve important data about hardware. This chapter is divided into on-core and un-core hardware modeling methodologies.

The on-core method aims at presenting opportunities to automatically retrieve instruction latencies and execution ports to build computational models. Section 2.5 presents the first steps towards memory modeling: it shows that several undocumented information about hardware can be discovered through experimentation. We also presents the essential parameters needed to build memory hierarchy models and what factors are influencing memory performance.

2.2 On-core Modeling: Computational Model

We saw that it is important to know the latency of instructions as well as the execution port each instruction can be executed on. This enables scheduling instructions for increasing the pipeline utilization or to give feedback to programmer about which optimizations to use to speedup software performance. By knowing the latency of each instruction of the instruction set of a given architecture, we are able to know the time elapsed between when instruction issue and it is retired.

This allows predicting the pipeline utilization and time needed for a given block of instructions to be executed.

But instruction latencies are not documented on many general purpose processors. Since the trend is to have architectures with more and more instructions, bringing all instruction latencies to a model can be a tedious task. To move towards automatic hardware modeling, we are going to see how the information can be found automatically.

2.2.1 Related Work

In order to perform the instruction performance measurement presented in this section, we used the benchmarking framework that will be presented more thoroughly in Section 3.3.2. The basic idea of this framework is to allow users to write their own benchmark function. User defined benchmark function can then be called from the framework that handles the time measurement, and repeats the experience several time to achieve best performance. The only user input needed is, the code to benchmark and the number of instruction in the code[1]. In this chapter we will therefore mainly focus on generating the correct code to measure a particular performance metric.

Other existing tools are designed to perform low level hardware benchmarks. MicroTools is a framework that fits exactly our needs because because it allows user to write their own benchmarks [11]. It handled register renaming at source code level as well as loop unrolling in order to select the best code version. However at the time we did this work it was not yet released as an open source software. Therefore we could not use it to carry out our work. LIKWID is a performance-oriented toolbox [80]. One of the tools embed in this project is called likwid-bench that eases micro-benchmark writing. It allows prototyping benchmarks by passing several options to the likwid-bench tool. Several benchmarks are provided out of the box. User can also extend the framework by writing their own functions. A strength of LIKWID is that user can define function with a meta language translated into assembly. This meta-language allow easier kernel writing because it avoids to the user the task to manipulate and manage registers. We could have used this framework to perform our analysis, however we had already developed framework allowing this prior to the work described in this section. Therefore we choose to use our own framework. The kind of benchmarks we needed to build for the study in Chapter 3 and Chapter 4 could not be handled with LIKWID. But we will elaborate on these reasons in Chapter 3.

Agner releases performance numbers of every new released architecture [31]. He provides the community with a great number of performance data of a wide range of architectures. In the work presented in this section, before the actual numbers we are interested in the methodology. We want to show how critical performance parameters can be automatically retrieved by mean of micro-benchmarks. This is why we will fist present our methodology and then we will evaluate the result we had by comparing them with Agner's data.

The work presented in this section was lead with Mathieu Audat and James Tombi A Mba, two students doing an internship under our supervision. The results of instruction performance measurement made with our methodology and framework were used to build the MAQAO [9] static performance model for the Xeon Phi processor.

[1]The user can also provide the number of bytes of memory accessed during the benchmark. This is used to measure memory performance, but this will be the subject of Chapters 3 and 4.

2.2.2 A methodology to measure Latency, Throughput, and to detect Execution Port assignations

Instruction latency is the time elapsed between an instruction is issued and it is retired. For a memory instruction, it cannot be predicted in processors featuring cache hierarchy or NUMA architectures: the latency depends on the location of the memory data accessed. But memory performance will be covered later, for now we only focus on latency of arithmetic and branching instructions.

Measuring `x86` Instruction Latencies Instruction latency is the number of cycles it needs to be executed completely. In order to measure it, we have to measure the number of CPU cycles needed to execute a large number of them and divide the time found by the number of instruction executed. However when running this experiment on a pipelined processor, several instructions can be executed at the same time (see Section 1.1.1). This would lead to undervaluing the latency of instruction. To avoid this error, we try to cheat the pipeline by filling it with instructions depending on other. This will force the processor to execute only one instruction at the same time. Yet register forwarding (see register forwarding in Section 1.1.1 on page 10) in the pipeline can still happen, but avoiding it is much harder. However this not a big issue. Indeed, we need the real time elapsed between the execution of two instructions depending on each other to predict the run-time a given code. Since in real code the register forwarding will be used, it is not a problem if it also happens in the measurement of instruction latency.

Instruction Syntax The syntax for operand is a comma separated list of the type of the operands. Immediate value are represented with `imm`, `SSE` registers are represented with `xmm` and general purpose registers with `r64`. We can see the syntax of several `x86` instruction in Listing 2.1. For instance, on the first line, we see that instruction `ADD` took two operands: the first one can be either an immediate value or a general purpose register and the second operand is a general purpose register. In order measure the instruction latency of `x86` code we need an instruction listing as well as the instruction syntax.

```
addpd       xmm, xmm
add         imm/r64, r64
insertps    imm, xmm, xmm
```

Listing 2.1: Instruction Syntax examples: the `ADDPD` instruction takes two `SSE` registers as argument. The `ADD` instruction can take either an immediate value or a 64 bit register as as first operand and a 64 bi. register as a second argument.

The syntax of an instruction represents: the instruction name (`ADDPD` in the example in Listing 2.1), and its operands (two `SSE` registers in example in Listing 2.1). As we can see in Table 2.2, operands can be, immediate values (represented with the `imm` symbol), `SSE` registers (represented with `xmm`), general purpose registers (`r64` for 64 bit registers, `r32` for 32 bit registers, *etc*). Since we only target non memory instruction, we do not need to represent memory reference syntax.

From a list if instruction with their syntax, as depicted as in Listing 2.1, we can automatically generate several code patterns:

1. A code pattern with instruction dependency between every instruction and its predecessor (an example can be seen in Listing 2.2). This code will allow us to measure the instruction latency.
2. We can also automatically generate a code with no dependency that will allow us to measure the maximal instruction throughput (an example can be seen in Listing 2.4).

The code generated to measure the latency of the `ADDPD` instruction is shown in Listing 2.2. This code is the body of the loop used to perform the measurement.

```
ADDPD   XMM0, XMM0
ADDPD   XMM0, XMM0
ADDPD   XMM0, XMM0
ADDPD   XMM0, XMM0
ADDPD   XMM0, XMM0
ADDPD   XMM0, XMM0
ADDPD   XMM0, XMM0
ADDPD   XMM0, XMM0
```

Listing 2.2: `x86` code used to measure `ADDPD` instruction latency.

We can see in Listing 2.2 that every `ADDPD` instruction depends on the previous one: only a single instruction can be issued at each cycle. The previous instruction has to retire before a new one can be issued. This code is put into the body of a loop and the loop is run several times. Since the loop is unrolled (*i.e.* `ADDPD` instruction is replicated 8 times in the loop body), the overhead of the loop condition and branching is small. In order to further decrease this overhead, we can unroll the loop by a higher factor. But unrolling the loop too much might end up lowering performance by exceeding the capacity of the instruction loop buffer (see section named Instruction Loop Buffer in Section 1.1.1, on page 10). We observed this effect for instance on the throughput measurement of the `ADD` instruction. With a loop unrolled by a factor of 64 we obtained a throughput of 0.33 while with an unroll of 1024 we recorded a throughput of only 0.5 instruction issued per cycle.

Table 2.1 shows the influence of the unroll factor on loop performance. We can see that, even with a low unrolling factor the performance of the loop are close to peak performance: a latency of three cycles. Only when the loop is unrolled by a factor of one – *i.e.* not unrolled – the loop performance decrease. This observation comes from the efficiency of the branch predictor and the speculative execution of the pipeline.

As long as we have a listing of the instructions we need to measure and their syntax we can automatically generate `x86` code to measure instruction performance. The only problems are with instruction referencing memory and branching instruction. Instruction referencing memory will be coverer in Section 2.5. Measuring the latency of branching instruction is still important to predict

Table 2.1: Influence of the Loop Unroll factor on Loop Performance.

Unroll factor	Latency
1	4.01
2	3.01
4	3.01
8	3.01

the loop performance. Since branching instructions affect the instruction flow, we have to be careful to avoid leaving the benchmark loop before the measurement is over. In order to do this we can build a code pattern that goes through branching instruction one after another. This code pattern is shown in Listing 2.3.

```
asm(     "i0:     JMP i1;
          i1:     JMP i2;
          i2:     JMP i3;
          i3:     JMP i4;
          i4:     JMP i5;
          i5:     JMP i6;
          i6:     JMP i7;
          i7:     JMP i8;")
i8: if(n>N) goto end;
```

Listing 2.3: Code pattern used to measure branching instruction latency.

Checking conditional branches can be done the same way, by using a conditional instruction to set the condition register to true and by going though a code code full of conditional branch that will all be taken.

2.2.3 Detecting Instruction Parallelism

On super-scalar processors, several instruction ports can execute the same instruction. For instance, on the Sandy-Bridge micro-architecture, three ports are dedicated to arithmetics. This allows several instructions of the same type to be issued and executed at the same time. To build a full computational model we need to know the number and the kind of instructions the CPU can issue within the same cycle. For this purpose, the *throughput* is an important metric. The *throughput* is the average number of cycles elapsed between two instructions can be issued. By measuring the *throughput* of instructions, we can deduce the number of execution ports dedicated to a given instruction. For instance if the *throughput* of an instruction is 0.33 cycle, this means that 3 instructions of this kind can be issued at the same cycle. Thus we can conclude that the processor has at least 3 ports that can be used to execute this instruction.

Instruction Throughput

To measure the maximal instruction throughput, we have to produce a code pattern that allow as much instructions as possible to be filled in the pipeline at the same time. Unlike when measuring latency we have to remove as much dependence as we can. In order to measure the *throughput* of the `ADDPD` instruction we can use the code shown in Listing 2.4.

```
MOV     $1, R8D
MOV     $1, R9D
MOV     $1, R10D
MOV     $1, R11D
MOV     $1, R12D
MOV     $1, R13D
MOV     $1, R14D
MOV     $1, R15D
```

Listing 2.4: Code pattern used to measure instruction throughput.

This way no instruction depends on the other and the maximal instruction throughput can be achieved.

Code Generator Overview The algorithm used to generate the code dedicated to latency measurement aims at producing code with a dependency between every consecutive instructions. In order to achieve this, the register written by an instruction has to be read by the next instruction to be generated. We have two register allocator that can be used to build such a decency chain. The first always returns the same register when a register name is to be generated. The other register allocator generates a new register name for each new instruction to be written, saves this name to use it as the source of the next instruction, and use the new allocated register as the destination register of the instruction. The algorithm used to generate code for latency measurement is described in Listing 2.5.

```
reg_list_sse = [xmm0, xmm1, xmm2, .., xmm15];
reg_idx_sse = 0;
reg_list_r64 = [rax, rbx, rcx, .., r8d, ...];
reg_idx_r64 = 0;

reg_list = [reg_list_sse, reg_list_avx, reg_list_r64, ...];

alloc_reg(reg_type) {
    i = reg_idx(reg_type);
    reg = reg_list[reg_type][*i];
    *i = (*i + 1)%sizeof(reg_list[reg_type]);
    return reg;
}

write_latency_code(instruction_syntax) {
    first_reg = 0;
```

```
    for (i=0; i<loop_unroll; i++) {
        write(instruction_syntax.instr);
        for (o=0; o<instruction_syntax.noperands; o++) {
            op = instruction_syntax.operand[o];
            if(is_reg_operand(op)) {
                if(!first_reg || is_dest_operand(op)) {
                    reg = alloc_reg(op.reg_type);
                    first_reg=reg;
                }
                if(is_dest_operand(op) && i == loop_unroll-1) { // last
                    write(first_reg); // instruction in the loop body:
                                      // write to the first register
                                      // allocated to forward
                                      // the decency chain to
                                      // the next loop iteration
                }
                else {
                    write(reg);
                }
            }
            else if (is_imm_operand(op)) {
                write_immediate_value();
            }
            else {
                ...
            }
        }
    }
}
```

Listing 2.5: Code generator pseudo-code.

For the register allocator to always generate the same register when building the decency chain, we have to provide a list of registers containing the single register we want to use.

Note: we have to outline that, for instructions having a single register in their operands, no matter how many registers we provide to the code generator, just a single one will be used.

Code Generator for Throughput measures Only minor changes have to be made to generate code able to measure instruction throughput rather than latency. To measure throughput we need a code with no dependency, but name dependencies. These dependencies that will be removed by the register renaming mechanism. In order to avoid instruction dependencies, we allocate a new register for every register operand.

Porting the code generator to other architectures Porting this code generator to other architectures is easy since the algorithms used are generic. The changes to be done are to provide

to the register allocator with the information it needs about the hardware: the register lists and types.

Table 2.2 presents some of the result we obtained on several instructions. The second column specifies the operand of the instruction. This is especially important for instruction that can take several operands. For instance the `MOVAPS` instruction we measured is a copy from one `SSE` register to another because is takes two registers as operand. But the same instruction can also take a memory location and a register as parameter in this case it would be a memory access. Columns three and four compare the instruction throughput we found with our method and the throughput provided in Agner's document. Columns five and six show the latency measured with our method and Agner's data.

Table 2.2: Comparison of Instruction Performance measured with our method and Agner's data on the Sandy-Bridge Architecture.

Instruction	Operand	Throughput		Latency	
		Our	Agner	Our	Agner
ADD	imm/r64, r64	0.34	0.33	1.00	1
MOV	imm/r64, r64	0.34	0.33	1.00	1
INSERTPS	imm, xmm, xmm	1.01	1	1.02	1
SHUFPS/D	imm, xmm, xmm	1.01	1	1.02	1
ADDPS/D	xmm, xmm	1.02	1	3.00	3
ANDPS/D	xmm, xmm	1.01	1	1.02	1
MOVAPS	xmm, xmm	1.01	1	1.02	1
ORPS/D	xmm, xmm	1.01	1	1.02	1
MAXPS/D	xmm, xmm	1.02	1	3.00	3
MINPS/D	xmm, xmm	1.02	1	3.00	3
HSUBPS/D	xmm, xmm	2.01	2	5.00	5
MOVQ	xmm, xmm	0.34	0.33	1.00	1
MOVSS/D	xmm, xmm	1.01	1	1.02	1
MULPS/D	xmm, xmm	1.02	1	5.00	5
PADDB/W/D/Q	xmm, xmm	0.51	0.5	1.01	1
PAND	xmm, xmm	0.34	0.33	1.00	1

The results presented in Table 2.2 were run on an Intel Xeon E5-2650 CPU running at 2.00 GHz. In order to achieve reproducible and stable results we fixed the processor frequency by disabling frequency scaling and Turbo Boost. The benchmarks were written in inline assembly code and compiled with Intel ICC compiler version 13.0.1. The code was run on Linux kernel version 3.2.0-3. We use the `RDTSC` instruction to access the time stamp counter to perform high resolution time measurement. The loops measuring the instruction latency and throughput are unrolled by a 64 factor and are executed 1024 times. We unrolled the loops by a large factor to minimize the overhead due to the loop end condition checking and induction variable update: a `sub` instruction and a conditional branch `jnz`. The framework we used automatically runs the benchmarks 10 times and reports the performance of the best measurement.

We can see from Table 2.2 that our measurements are very close the performance reported by Agner: the difference is at most 3%.

Impact of Register on Performance Table 2.3 presents the throughput we measured for several code version of the benchmark measuring the throughput of the `ADDPD` instruction. The code measuring throughput is made of independent instructions. Therefore they operate of different registers, as it was shown in Listing 2.4. But we can choose to build code version with more or less registers, Listing 2.4, show a code version with eight different registers. But Listing 2.6 shows the same code pattern only using 4 registers, still with an eight-unrolled loop.

```
ADDPD   XMM0, XMM0
ADDPD   XMM1, XMM1
ADDPD   XMM2, XMM2
ADDPD   XMM3, XMM3
ADDPD   XMM0, XMM0
ADDPD   XMM1, XMM1
ADDPD   XMM2, XMM2
ADDPD   XMM3, XMM3
```

Listing 2.6: Code pattern used to measure instruction throughput with only four registers.

The performance summarized in Table 2.3 presents the performance of such code version with a number of registers varying between one and eight. The code version with a single register used is code used to measure the instruction latency since there is a dependence between every instruction of the code. It is not surprising to find that the performance of this code corresponds to the latency of the `ADDPD` instruction. If we use two registers, the pipeline is able to issue two `ADDPD` instructions per cycle. Then stalls for three cycles waiting the dependence to be resolved. After the three cycles two instructions retire and two new can be issued. This leads two instruction executed every three cycles, this explains the throughput of about 1.5. When we use three registers, the processor should be able to issue three instructions per cycle, wait for three cycles to resolve the dependences and issue again three new instructions. However, as we can see this is not true since the throughput measured for this code version is 1.13. We think that this comes from small delays sometimes happening in the front-end of the pipeline. These delays can sometime avoid delay an instruction issue preventing the throughput to be exactly 1. If we increase further the number of registers used in the code, we release the stress on the instruction issue because four instruction can be executed every 3 cycles. Therefore even if one among them is delayed, another can take the spot. This explains the throughput of about 1 observed when using from four to eight registers.

Instruction Execution Port

The last critical information for predicting accurately code performance is to know what instruction can be executed at the same time as others, *i.e.* which instructions use the same execution port as others. To check whether two instructions use the same execution port, we can build a

Table 2.3: Comparison of several code versions of the `ADDPD` benchmark.

Registers	Throughput
1	3.01
2	1.51
3	1.13
4	1.02
5	1.02
6	1.02
7	1.02
8	1.02

Table 2.4: Comparison of execution time of two code versions to deduce if two instructions share an execution port.

With *mov* instruction only utilizing one execution port.	With *mov* instruction using all available execution ports.

`MOV R8D, R9D; // port 0`
`FADD EAX; // port 1`
`MOV R10D, R11D; // port 5`
`FADD EBX; // port 1`
`MOV R12D, R13D; // port 0`
`FADD ECX; // port 1`
`MOV R14D, R15D; // port 5`
`FADD EDX; // port 1`
	`MOV R8D, R9D; // port 0`
	`MOV R10D, R11D; // port 1`
	`MOV R12D, R13D; // port 5`
	`FADD EAX; // port 1`
	`MOV R8D, R9D; // port 0`
	`MOV R10D, R11D; // port 5`
	`MOV R12D, R13D; // port 0`
	`FADD EBX; // port 1`

benchmark that interleaves the two kinds of instructions. If the execution time of the kernel with the two kinds of instruction is the maximum of the run time of the kernel with only one kind of instruction, this means that the instruction be issued at the same time and are executed by different execution ports. However if the run time of the interleaved kernel is the sum of the run time of the kernels with a single instruction kind, then the instructions were issued one after the other and we can deduce that they use the same execution port. When an instruction can be issued on several execution port, it is important to fill *all* execution ports with independent instructions. For instance on Sandy Bridge micro-architecture, the *mov* instruction can be used to copy the content of a register to another register. This instruction can be executed by ports 0, 1 and 5. And the *fadd* instruction performing a floating point addition can only be executed in the port 1. Table 2.4 illustrates what would happen if not all execution ports were used and how it would lead to mistakes.

On the left hand side of Table 2.4 it would take 4 cycles to issue the 4 *mov* and the 4 *fadd* instructions. If there were only the 4 *mov* instructions, it would take 2 cycles to issue. And with only the 4 *fadd*, it would take 4 cycles. Since executing the full block of 8 instructions takes the same time as the maximum of executing the different instructions separately we deduce that *mov*

and *fadd* instruction do not share any execution port. But if we look at the right hand side of Table 2.4, issuing the 8 instructions takes 3 cycles while issuing only the *mov* instruction would take 2 cycles, and issuing only the *fadd* would also take 2 cycles. Since interleaving these instruction is slower than executing them separately, we can deduce that these instructions share an execution port. We can also understand that they do not share all execution port (otherwise the interleaved code would take 4 CPU cycles to issue the 8 instructions).

In order to know the exact number of execution ports shared by two instructions, we can generate all interleaved code versions with a number of instruction of each kind from 1 to the number of execution port it uses. For instance, with the example shown in Table 2.4, we can build a code with only 2 *mov*s and 1 *fadd*. This code would run in 2 cycles: and we can deduce that these instructions share less than 2 execution ports.

The code generator used to generate interleaved code to test is a couple of instruction share an execution port is simple: we concatenate the code used to measure the throughput of the two instructions into the same loop body.

However, with the short period of time of the internship we did not had time to build a full automatic framework to retrieve instruction latency and execution port shared by every instruction pair. Yet we presented a method that can be automated to get these information from real hardware. We ran several measurements on real processors to check if this method is able to retrieve value found in literature. We showed that we were able to measure instruction latency and throughput with a good precision since the difference between our measurements and data found in literature is at most 3%.

2.3 Case Study: Power Aware Performance Prediction on the SCC

The work presented in this section was presented in depth in paper [A1]. It is a good example of combining several models into a larger one able to model the behavior of several pieces of a real hardware. In this work we built a computational model, a memory model and a power consumption model that, combined all together are able to predict the runtime of regular code and the power consumption of the underlying SCC chip in order to let end users optimize either runtime of the application or power efficiency depending on their own needs.

As power is becoming one of the biggest challenge in high performance computing, we have created a performance model on the Single-chip Cloud Computer in order to predict both power consumption and runtime of regular codes. This model takes into account the frequency at which the cores of the SCC chip operate. Thus we can predict the execution time and power needed to run the code for each available frequency. This allows to choose the best frequency to optimize several metrics such as power efficiency or minimizing power consumption, based on the needs of the application. Our model only needs some characteristics of the code. These parameters can be found through static code analysis. We validated our model by showing that it can predict performance and find the optimal frequency divisor to optimize energy efficiency on several dense linear algebra codes.

2.3.1 Related Work

Power efficiency is a hot topic in the HPC community and has been the subject of numerous studies, and the Green500 List is released twice a year. Studies carried out at Carnegie Mellon University in collaboration with Intel [25] have already shown that the SCC is an interesting platform for power efficiency. Philipp Gschwandtner *et al.* also performed an analysis of power efficiency of the Single-chip Cloud computer in [67]. However, this work focuses on benchmarking, while our contribution aims at predicting performance according to a theoretical proposed model.

Performance prediction in the context of frequency and voltage scaling has also been actively investigated [24, 50, 70], and the model usually divides the execution time into memory (or bus, or off-chip) [37, 52], instruction and core instruction, as we did in this paper.

2.3.2 The SCC Architecture

Before going into the details of our models of the SCC chip we will briefly describe the key feature of the SCC architecture. By key features, we mean what is important to understand about this particular hardware to be able to understand our models. More details about the SCC chip can be found in [55].

The Intel Single-chip Cloud Computer (SCC) is a good example of possible next generation hardware with easy way to control power consumption. It provides a software API to control core voltage and core frequency. This opens promising opportunities to optimize power consumption and to explore new trade-offs between power and performance.

The SCC chip feature 24 dual core tiles. This tiles are connected through a 2 dimensional mesh. An overview of the chip organization is presented in figure 2.1.

Figure 2.1: Overview of the SCC chip Architecture. The Chip is organized in 24 dual core tiles connected through a two dimensional mesh. L1 and L2 caches are private and embed on the tile.

The SCC chip feature a novel memory organization. Several memory level are available: a shared off-chip DRAM memory module that can be addressed by all core of the chip. Private off-chip DRAM: chunks of memory that are only addressable by a core. Each core on the die feature

a private 16 kB level 1 cache and a private 256 kB level 2 cache. Also and this is the very novel feature of the SCC chip a shared on-chip SRAM module called the Message Passing Buffer (MPB) can be addressed by all cores. The MPB can be accessed by every core of the chip but is distributed across all cores: cores access memory addresses held in the MPB module of its own chip faster than others. The MPB module is used to perform fast inter-core communication. Figure 2.2 illustrates the memory organization of the SCC chip as seen from the programmer point of view.

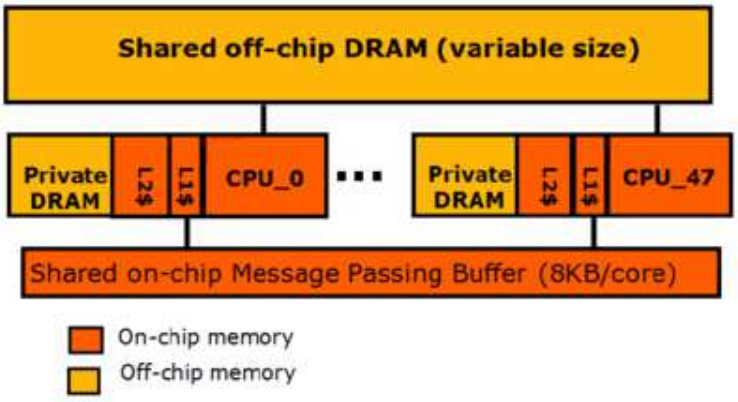


Figure 2.2: The Memory organization of the SCC.

2.3.3 Performance Model

In this section we provide a performance model in order to predict the impact of core frequency scaling on the execution time of several basic linear algebra kernels on the SCC chip. As we focus on dense linear algebra, we only need little data to predict a given code performance. The considered datasets being too large to fit in cache, we need the execution time of one iteration of the innermost loop of the kernel and the memory latency.

The performance model is divided in two parts: memory model and computational model. Although our work on actual memory models will be presented later in this chapter (see Section 2.5), a full performance model of the SCC chip is still required to evaluate code performance. Since SCC caches are not coherent the memory model is simplified.

Memory model

To build the memory model, we assume that the application can exploit perfectly data reuse and therefore we assume that each data is accessed only once. We do not take the number of cache accesses into account in the prediction of the overall memory access time because they are not actual memory accesses since the request does not have to go all the way to DRAM. Moreover the cache is not coherent. Therefore there is no overhead due to the cache coherence protocol.

According to Intel documentation, on the SCC, a memory access takes 40 core cycles + $4 \times n \times$ 2 mesh cycles + 46 memory cycles (DDR3 latency) where n is the number of jumps between the requesting core and the memory controller [78]. In our case, we are only running sequential code, therefore we are assuming that the memory access time is $40 \times c + 46 \times m$ cycles, where c is the number of core cycles and m the number of memory cycles. Accessing memory takes 40 core cycles plus 46 memory cycles.

Frequency scaling only affects core frequency, the memory frequency is a constant (in our case 800 MHz). Therefore, changing frequency mostly impacts the code performance if it is computation bound. The core frequency and the memory frequency are bound by the formula:

$$mem_freq = \frac{f_div \times core_freq}{2}$$

Therefore a memory cycle lasts $\frac{2}{f_div} = \frac{core_freq}{mem_freq}$ core cycles. Thus, the number of core cycles to perform one DDR3 RAM access is:

$$40 + 46 \times \frac{core_freq}{mem_freq} = 40 + 46 \times \frac{core_freq}{800}$$

As we can see from the formula dividing the core frequency by 8 (from 800 MHz to 100 MHz) will only reduce the memory performance by 46%.

As the P54C core used in the SCC supports two pending memory requests, we can assume that accessing x elements will take $\frac{x}{2}(40 + 46 \times \frac{core_freq}{800})$ core cycles.

Computational model

In order to predict the number of cycles needed to perform the computation itself we need the latency of each instruction. Agner Fog measured the latency of each `x86` and `x87` instruction [31]. We used his work to predict the number of cycles to perform one iteration of the innermost loops of each studied kernel (several BLAS kernels). But, as we saw in Section 2.2, we could also use the data collected with our computational model. The computational model is very simple, as most of the instructions use the same execution port, there is almost no instruction parallelism. We use such a tool to measure the execution time of one iteration of the innermost loop. As most of the execution time of the codes we consider is spent in inner loops, this performance estimation is expected to be rather accurate.

From this computational model the impact of frequency scaling on the computation performance is straightforward. The number of cycles to perform the computation is not affected by the frequency. Thus, reducing the core frequency by a factor of x will multiply the running time by x.

Power model

We use a simple power model to estimate the power saved by reducing the core frequency. Table 2.5 shows the voltage used by the tile for each frequency, these data are provided by the SCC Programmer's guide [78].

Table 2.5: Relation between voltage and frequency in the SCC chip.

Freq divisor	Tile freq (MHz)	Voltage (volts)
2	800	1.1
3	533	0.8
4	400	0.7
5	320	0.6
6	266	0.6
7	228	0.6
8	200	0.6
9	178	0.6
10	160	0.6
11	145	0.6
12	133	0.6
13	123	0.6
14	114	0.6
15	106	0.6
16	100	0.6

The power consumption model used in this paper is the general model: $P = CV^2f$ where C is a constant, V the voltage and f the frequency of the core. As shown in Table 2.5 the voltage is a function of the frequency, thus, we can express the power consumption as a function of the core frequency only.

We choose not to introduce a power model for the memory for two reasons: first we have no software control on the memory frequency at runtime. We can change the memory frequency by re-initializing the SCC platform but not at runtime. Thus, the memory energy consumption is constant and we have no control over it. It is irrelevant to try to model the memory consumption. The other reason is that until now we used models that can be transposed to other architectures. As the memory architecture of the SCC is very different from more general purpose architectures, its energy model would not fit for those architectures. Therefore, instead of complicating the model, it was decided to maintain a simplified form, which is relatively as precise as the full model and can be easily transposed to other architectures.

Overall model

In this section we describe how to use both the memory and computational models to predict the performance of a given code.

As the P54C core can execute instructions while some memory requests are pending, we assume that the execution time will be the maximum between the computation time and the memory access time:

$$model(f_c, size) = MAX\left(cycles_{comp}(size), cycles_{mem}(f_c, size)\right)$$

with f_c the core frequency.

With this runtime prediction, we estimate how a code execution is affected by changing the core frequency. Taking the decision to reduce the core frequency in order to save energy can be done with a static code analysis.

As show in the description of the SCC memory model (on page 48) in Section 2.3.3 the memory access performance is almost not affected by reducing core frequency, while reducing core frequency increases the computation time. From this observation we see that reducing core frequency for memory bound codes is highly beneficial for power consumption because it will almost not affect performance while reducing energy consumption. However, reducing core frequency for compute bound code will directly impact performance.

2.3.4 Model evaluation

In this section we compare our model with the real runtime of several regular codes in order to check its validity. We used three computation kernels, one BLAS-1: the dotproduct, one BLAS-2: the matrix-vector product, and one BLAS-3 kernel: the matrix-matrix product.

First let us describe how we applied our model to these three kernels: In the following formulas, f_{div} denotes the core frequency divisor (as shown in Table 2.5) and $power(f_{div})$ the power used by the core when running at the frequency corresponding to f_{div} (see Table 2.5). An important point is that we used large data sets that do not fit in cache. Thus, the kernel actually gets data from DRAM and not from caches. However, the matrix-matrix multiplication is tiled in order to benefit from data reuse in cache.

Dotproduct Multiplication

For the dotproduct kernel, the memory access time in cycles is:

$$cycles_{mem}(f_{div}, size) = size \times \left(40 + 46 \times \frac{2}{f_{div}}\right)$$

And the computation time in cycles is given by:

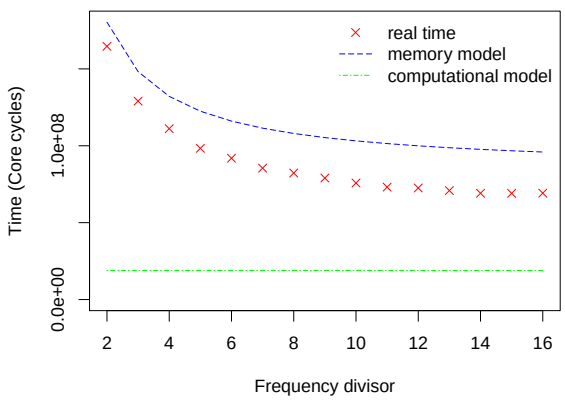
$$cycles_{comp}(size) = size \times \left(\frac{body}{unroll}\right),$$

where *body* is the execution time (in cycles) of the innermost loop body and *unroll* the unroll factor of the innermost loop. In the case shown on Figure 2.3, $body = 36$, and $unroll = 4$. Then the power efficiency is:

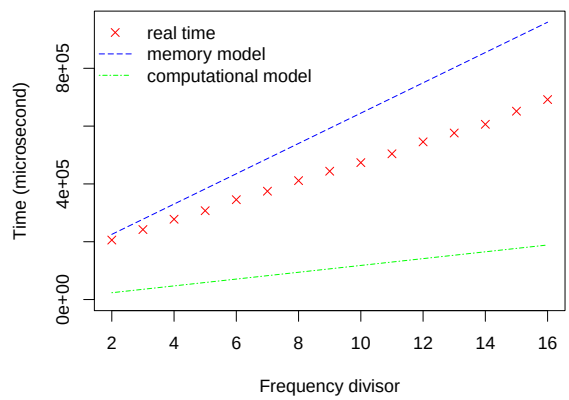
$$power_{eff}(f_{div}, size) = \frac{flop(size)}{\frac{model(f_{div}, size)}{freq} \times power(f_{div})},$$

with $power(f_{div}) = freq(f_{div})^2 \times voltage(f_{div})$, with $flop(size) = 2 \times size$, the number of floating point operations of the kernel, $model(f_{div})$ the number of cycles predicted by our model, and $freq$ the actual core frequency ($\frac{1600}{f_{div}}$). In the case shown on Figure 2.3,

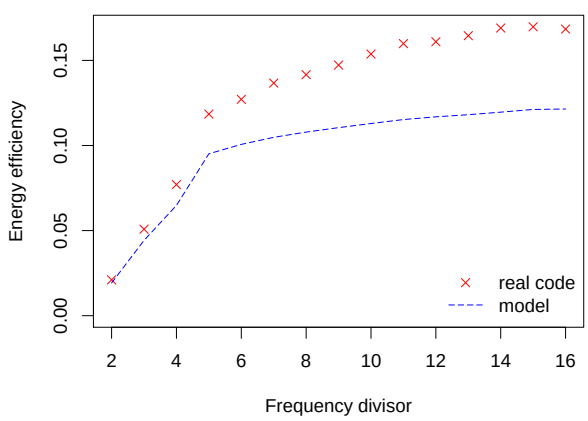
$$\begin{aligned} model(f_{div}, size) &= MAX\Big(cycles_{mem}(f_{div}, size), cycles_{comp}(size)\Big) \\ &= cycles_{mem}(f_{div}, size) \end{aligned}$$



(a) Dotproduct: the cycle count is shown according to the core frequency divisor.



(b) Dotproduct: runtime in microsecond depending on the core frequency divisor.



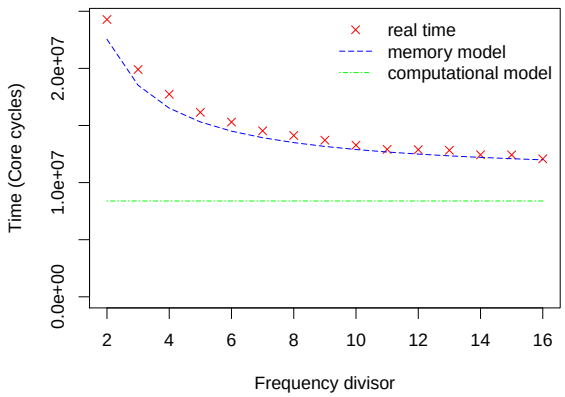
(c) Dotproduct: power efficiency (in GFlops/W) depending on the core frequency divisor.

Figure 2.3: Vector dotproduct model: sequential dotproduct on two vectors of 2^{21} double elements (16MB).

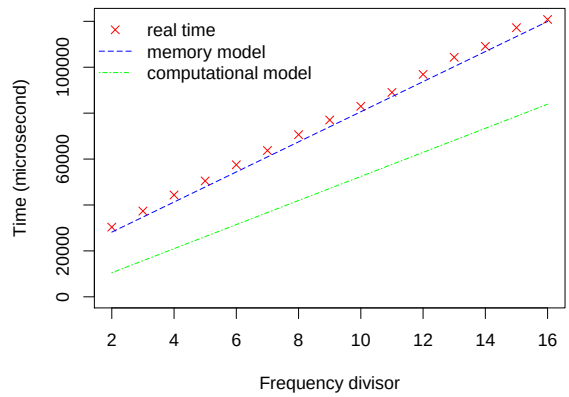
Figure 2.3a shows that the number of cycles for both the memory model and obtained through benchmark decreases when frequency decreases. The reason is that frequency scaling only affects core frequency. For memory bound codes such as dotproduct, reducing the core frequency reduces

the time spent in waiting for memory requests. However, the code is not executing faster, as shown in Figure 2.3b.

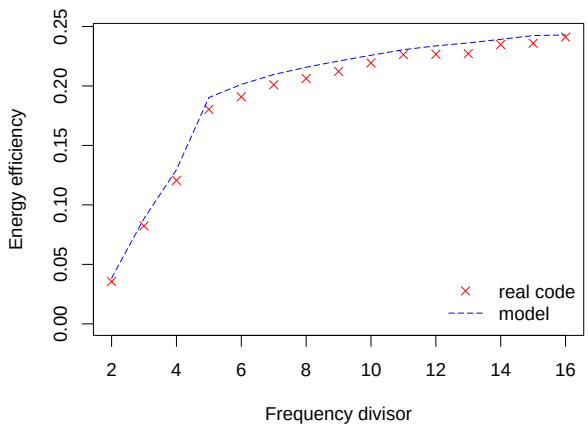
Matrix-vector product



(a) Matrix-vector product: the cycle count is given according to the core frequency divisor.



(b) Matrix-vector product: the execution time is given in microsecond depending on the core frequency divisor.



(c) Matrix-vector product: power efficiency (in GFlop-s/W) depending on the core frequency divisor.

Figure 2.4: Matrix-vector multiplication model: sequential code on a 512 by 1024 elements matrix.

Similarly the model for the matrix-vector product is:

$$cycles_{mem}(f_{div}, size) = \frac{size}{2} \times \left(40 + 46 \times \frac{2}{f_{div}}\right)$$

$$cycles_{comp}(size) = size \times \left(\frac{body}{unroll}\right)$$

With $size = 512 \times 1024$ elements, $body = 64$ cycles, and $unroll = 4$ for the case shown on Figure 2.4.

$$power_{eff}(f_{div}, size) = \frac{flop}{\frac{model(f_{div}, size)}{freq} \times power(f_{div})}$$

In this case, again, the memory access time is more important than the time for the computation, thus, the runtime is given by the memory access time (ie. $model(f_{div}) = cycles_{mem}(f_{div})$)

Figure 2.4a shows that the number of cycles for both the memory model and obtained through benchmarks decreases when frequency decreases. The reason is the same as for the dotproduct: frequency scaling only affects the core frequency. Since this code is memory bound, with slower core frequency the processor spends less time waiting for memory. However the execution time in second is not affected.

Matrix-matrix product

The model for the matrix-matrix multiplication is:

$$cycles_{mem}(f_{div}, size) = 3 \times \frac{size}{2} \times \left(40 + 46 \times \frac{2}{f_{div}}\right)$$

$$cycles_{comp}(size) = size^{\frac{3}{2}} \times \left(\frac{body}{unroll}\right)$$

$$power_{eff}(f_{div}, size) = \frac{flop}{\frac{model(f_{div}, size)}{freq} \times power(f_{div})}$$

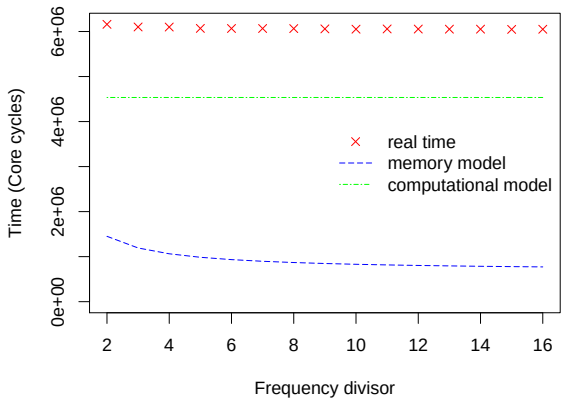
With $matrix_size = 160 \times 160$ elements (each matrix is 160×160 elements big), $body = 43$ cycles, and $unroll = 1$ for the case shown on Figure 2.5. Since this is a BLAS-3 kernel the computation time is – as expected – bigger than accessing memory. And:

$$model(f_{div}) = cycles_{comp}(f_{div})$$

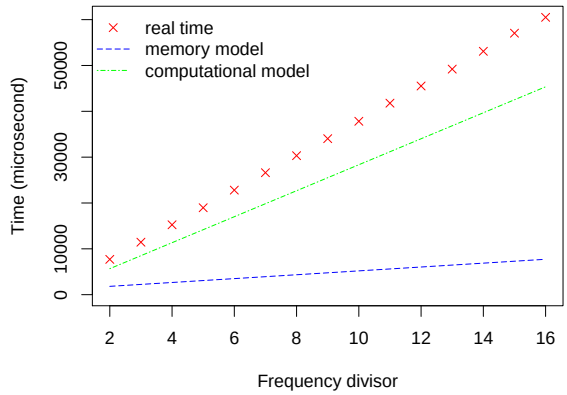
2.3.5 Power efficiency optimization

Our objective in this section is to show that the performance model we presented is able to help selecting the frequency scaling providing optimal power efficiency. Then the higher performance version is chosen among the most power efficient versions.

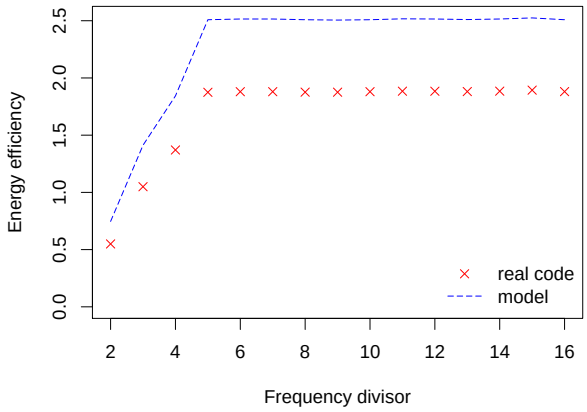
We can see that the dot and matrix-vector products are memory bound while the matrix-matrix product is compute bound. Power efficiency is measured through the ratio of GFlops/W. The best frequency optimizing power efficiency of those two kind of code are different. For the case of memory bound codes, the core frequency can be reduced by a large divisor as performance is limited by memory bandwidth which is not very sensitive to core frequency. On the contrary, for computation bound codes, the performance in GFlops decreases linearly with the frequency.



(a) Matrix-matrix product model: the cycle count is given according to the core frequency divisor.



(b) Matrix matrix product model: the time in microsecond depending on the core frequency divisor.



(c) Matrix-matrix product model: power efficiency (in GFlops/W) depending on the core frequency divisor

Figure 2.5: Matrix-matrix multiplication model: sequential code with two matrices of 160 by 160 elements.

Figures 2.3c, 2.4c and 2.5c represent power efficiency in GFlops/W for respectively dot, matrix-vector and matrix-matrix products. They show that our performance model is similar to the measured performance (from which we deducted power efficiency). Power efficiency for matrix-matrix product is optimal from a frequency divisor of 5, to 16. Among those scalings, the best performance is obtained for the scaling of 5 according to Figure 2.5a. For the Dotproduct 2.3c, codes are more energy efficient using a frequency scaling of 5, and their efficiency increases slowly as frequency is reduced. According to our performance model, around 25% of GFlops/W is gained from a frequency divisor of 5 to a frequency divisor of 16, and for this change, the time to execute the kernel has been multiplied by a factor 2.33 (according to our model). In reality, these factors measured are higher than those predicted by the model, but the frequency values for optimal energy efficiency, or some trade-off between efficiency and performance are the same. Note that for divisor lower than 5, energy efficiency changes more dramatically since the voltage also changes.

We choose to show how to optimize energy efficiency, but as our model predicts both running time and power consumption for each frequency, it is easy to build any other metric depending on power and runtime and optimize it. Indeed using this model allows to compute the metric to optimize for each frequency divisor and then to choose the one that fits the best the requirement. Even with a very simple model as we presented, we can predict the running time of simple computational kernels within an error of 38% in the worst case.

Our energy efficiency model is interesting because it shows exactly the same inflection points as the curve of the actual execution. This point allows us to predict what is the best core frequency in order to optimize the power efficiency of the target kernel.

It is also interesting to see that even with a longer running time all the kernels (even matrix multiplication which is compute bound) benefits from frequency reduction. This is due to the fact that *i)* the run time of such kernels is proportional to the frequency; *ii)* the power consumption is also proportional to the frequency. So the energy efficiency does not depend on the core frequency. But the 3 first steps of frequency reduction also reduce the voltage, which has an huge impact on power consumption.

2.3.6 Summary

We described a method to predict performance of some linear algebra codes on the Single-chip Cloud Computing architecture. This model can predict the runtime of a given code for all available frequency divisor and using the known relationship between frequency scaling and voltage, it can also predict power efficiency. Based on this prediction we can choose the frequency that best suits our needs: depending on the urgency of getting a result we can chose to save or not some energy.

Our contribution is slightly different from usual approach as we do not use any runtime information to predict the impact of frequency and/or voltage scaling on performance. As we use static code analysis to predict performance of a kernel, this could be done at compile time it and does not increase the complexity of runtime system. Static Performance prediction has also been used in the context of auto-tuning. Yotov *et al.* [88] have shown that performance models, even when using cache hierarchy, could be used to select the version of code with higher performance. Besides, In [8], the authors have shown that a performance model, using measured performance of small kernels, is accurate enough to generate high performance library codes, competing with hand-tune library codes. This demonstrates that performance models can be used in order to compare different versions, at least for regular codes (such as linear algebra codes).

Our proposition is only a first step toward a full model of the SCC ship since it only handles sequential regular code. Still, we showed that bringing power consideration into a performance model can help reduce power consumption through chip frequency and voltage control.

2.4 Summary about On-core Modeling

The 2 last sections described how to model the on-core part of processors. We described a general methodology able to automatically measure instruction performance. We applied this model to the

several `x86` micro-architecture and retrieved measurements close to those that can be found in literature. We difference of our approach compared to related work such as Agner's instruction listing [31] is that our methodology is detailed and we also provide discussion and analysis of code performance depending of parameters such as loop unrolling, and register usage. Also we developed a methodology to detect execution port sharing while Agner only provides raw information about instruction performance.

We used these information to model the Intel SCC architecture. This model was used provide information for power efficient optimization on the SCC. Also instruction performance measured with our methodology and tools were integrated into the static performance model of MAQAO [9] for the Intel Xeon Phi processor.

Yet a fully automatic tool to find instruction execution port sharing as to be developed. While we described the method and tried it on small cases, we did not implement is yet in our benchmarking framework. Supporting more architecture, *i.e.* other instruction set architecture (ISA) would also make our method and framework more generic and enlarge its use. The general approach to target other ISA would not change, only the code generator, the list of instruction syntax and architecture representation in the code generator have to be updated to match a new hardware.

2.5 Un-Core Model: Memory

In order to model the entire hardware architecture, the on-core part is not sufficient. Especially since memory performance is becoming more and more critical to computer performance (*cf.* paragraph about the memory wall on page 20 in Section 1.3). This section is dedicated to memory hierarchy performance modeling.

2.5.1 Memory Hierarchy Parameters Needed to build a Memory Model

In order to build a memory model able to reflect behavior of multi-core system, we have to investigate cache parameters affecting performance of cache hierarchy. Also we have to keep in mind that we want our model to be effort-free for users – such as compiler, performance tuning, or library developers. We will therefore investigate the availability of each these parameters or how they can be automatically discovered. Different critical parts of a memory model are studied in the following sections.

Capacity Model The capacity model of a cache hierarchy aims at predicting why and when capacity misses occurs. In order to build such a model it is crucial to know the size of each cache level. As well as the replacement policy used to flush lines out if the cache is full. Knowing the size of each cache level helps predict when the cache is full. When the cache is full and software accesses memory references not in the cache it frees a line for the new reference to get into the cache. Knowing the replacement policy used by the cache allows tracking which cache line are evicted of the cache (leading to cache capacity misses when later referenced). As well as the replacement policy, one needs to know where a cache lines goes when it is flushed out of a cache level. For instance cache hierarchies often feature *victim caches*. Memory references are only stored in *victim*

caches when they are evicted of a lower cache level (unlike in regular caches where data is stored after a cache miss).

The size of each cache level is easy to get since it is documented by processor vendors and available at run time thanks to tools abstracting the hardware architecture [13]. The replacement policy is harder to get, especially in the case where lines are stored when removed from one cache level. It also seems that newer cache architectures feature several cache replacement policies and are able to select the best one depending on metrics recorded at run-time such as the hit/miss ratio [46, 69].

Cache Associativity Conflict misses can be predicted and/or detected by embedding cache associativity and the hash function into cache models. The hash function of a cache is a function of the address requested that gives the line – or more precisely the set – where an address should be stored in the cache. If one knows the cache line where each accessed address goes, one can simulate memory accesses of a program and predict where each address is stored to detect conflict misses.

The cache associativity of each cache level is well documented by processor vendors. But the hash function is not. However cache simulators often use a formula that seems reasonable and performs well [71, 83]. This formula specifies that the line is selected depending on the bits $M{:}(M+N-1)$ of the address to be stored in the cache. Where the cache line is 2^M bytes wide and $2^N = \frac{number\ of\ cache\ lines}{cache\ assiciativity}$.

Cache associativity can be retrieved thanks to run-time measurement. Given a k-associative cache of n lines: repeating accesses to a memory segment of size $(n+1) \times cache\ line\ size$ leads to $n-k$ hits and $k+1$ misses. Indeed the n first accesses load n lines in the cache, the last access will evict a line from one set. If we assume the least recently used line is flushed out, the oldest cache line from the set where line $n+1$ should go is evicted. This means that all memory accesses going to the set where $(n+1)^{th}$ line goes are misses: we do have $k+1$ cache misses on this benchmark.

way 0	index 0	addr 0
	index 1	addr 1
way 1	index 0	addr 2
	index 1	addr 3
way 2	index 0	addr 4
	index 1	addr 5
way 3	index 0	addr 6
	index 1	addr 7

(a) After the first n memory accesses, the cache is full.

way 0	index 0	**addr 8**
	index 1	addr 1
way 1	index 0	addr 2
	index 1	addr 3
way 2	index 0	addr 4
	index 1	addr 5
way 3	index 0	addr 6
	index 1	addr 7

(b) The $n+1^{th}$ access evicts the first line of the cache (since it was the LRU line of the set).

way 0	index 0	addr 8
	index 1	**addr 0**
way 1	index 0	addr 2
	index 1	addr 3
way 2	index 0	addr 4
	index 1	addr 5
way 3	index 0	addr 6
	index 1	addr 7

(c) When the first address is accessed again, it flushes addr 1 from the first set.

Figure 2.6: Illustration of a benchmark to measure cache associativity.

Performance of each Memory Level Also to be able to predict real hardware behavior, the model has to reflect the performance of each cache level, *i.e.* the access latency and the available bandwidth.

These parameters of the cache hierarchy are highlighted by processor vendors. And they can be easily verified thanks to benchmarks [57, 76, 81].

In order to reflect performance of a parallel software, modeling raw performance of each cache level is not enough: contention has to be taken into account. Indeed when several threads access shared memory resources, they have to share the available bandwidth. This is called contention and can be the root of low performance. Cache contention can have several sources, contention on the cache itself: computing threads compete for cache space to store their data, leading to a virtually reduced cache size. This kind of contention is already well modeled [87]. But contention can also happen on the memory bus. Some research were lead by Ajmone Marsan *et al.* to understand the impact of bus contention [4]. However prediction of the impact of contention on modern computer architecture is still unclear and performance prediction of parallel applications with bus contention is still an open challenge. Yet some studies of Andersson *et al.* show that predicting an upper bound of performance degradation due to contention can be achieved [5].

The prediction of bus contention is the first hint leading us to think that building a full analytical model of memory modern CPU memory hierarchy is such a complicated problem that we want to use other methods in order to keep the model simple.

2.5.2 Cache Coherence Impact on Memory Performance

Most of the parameters and models described until now can be found either in the manufacturer's documentation, published work or even discovered by experience (except for bus contention). But the biggest deal is modeling cache coherence. Indeed the access cost to a cache line not only depends on the cache level accessed but also on the state of the cache line [38]. Depending on the hardware mechanisms involved in maintaining cache coherence, the performance of memory accesses can vary widely. Figure 2.7 illustrates this by presenting the write bandwidth available for several cache states (see Section 1.3.3) of data. As we can see, cache coherence has a big impact on memory performance and can not be ignored in memory modeling.

But the issue with cache coherence is that an important part of the protocol implemented in hardware is undocumented. Especially we are not aware of the coherence messages transferred on every cache event. For instance the performance gap between loading a dirty or a clean[2] cache line from a remote cache on Figure 2.8, can have several reasons depending on choice in the implementation of the cache coherence protocol:

- the dirty cache line is written to memory and then fetched from memory to the cache requesting it.
- or the cache line can be put on the bus for the requesting cache at the same time as the line is written to main memory.

[2]Modified cache lines are dirty: the value they hold is not consistent with main memory. Exclusive and shared cache lines are clean: the value they hold is the same as main memory.

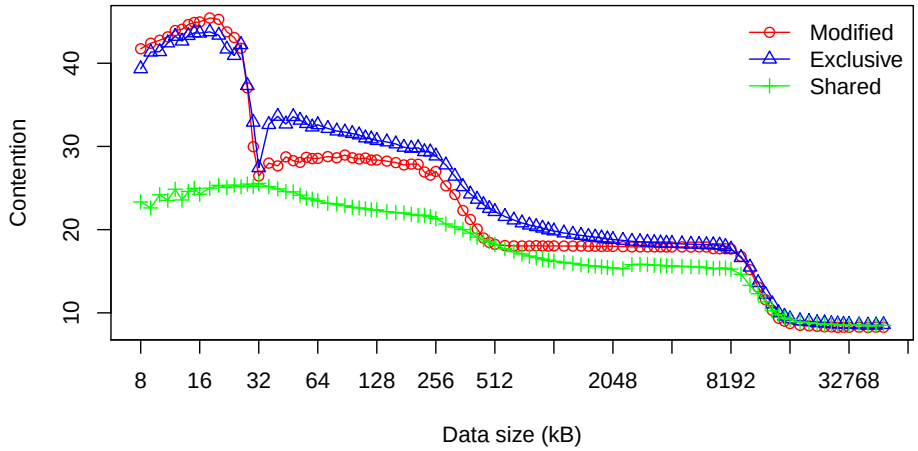


Figure 2.7: Write Bandwidth measured on a Xeon X5650 Processor (Nehalem micro-architecture) depending on the data size. Depending on the state of data in the cache, write performance can be affected by a factor up to 2.

We presented an early version of this work in [C1] to illustrate how cache coherence can affect memory and code performance.

2.5.3 Bringing Coherence into a Memory Model

In order to build an analytical model taking into account cache coherence issues highlighted in the previous section, we tried to add some extra parameters to the model. These parameters are supposed to indicate the bandwidth used by each kind of coherence messages. In order to keep the model abstracted enough to be applied to several cache architectures, we choose a general enough coherence protocol that will capture the behavior of more specialized ones that are implemented in real hardware. We choose the `MESI` protocol (*cf.* Section 1.3.3) because general purpose processors built by Intel and AMD use protocols based on this particular protocol. The coherence messages involved in this protocol are:

Write Back

This coherence message is responsible for writing a cache line back to main memory. It is triggered when a cache reads an address stored in another cache in a dirty state.

RFO

The Request For Ownership is a broadcast on the bus asking caches holding a particular cache line to put it on the bus and to invalidate this line after. It is caused by write misses.

Invalidation

This coherence message asks remote caches to invalidate lines holding a particular address. This event is triggered when a write hits the cache in a shared line.

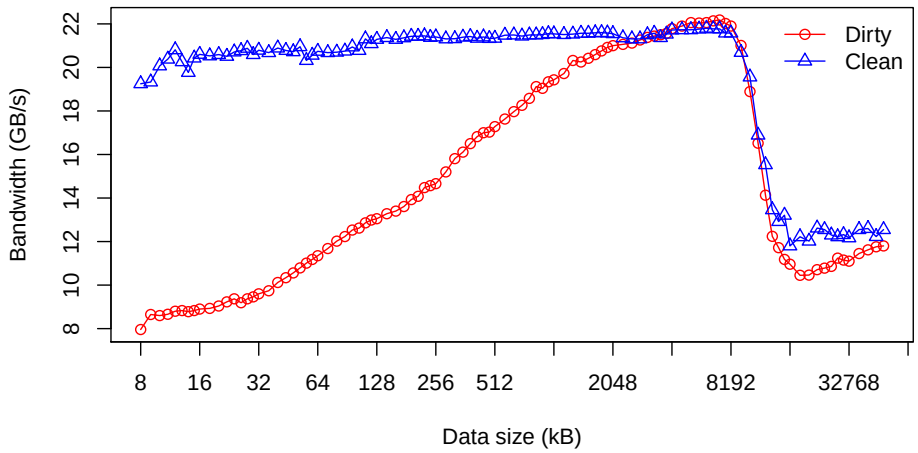


Figure 2.8: Read miss Bandwidth measured on a Xeon X5650 Processor (Nehalem micro-architecture) depending on the data size and state of data in the caches.

Real Bandwidth and Effective Bandwidth

Our idea to build a coherence aware analytical model is to compute – by means of benchmarks – the overhead of coherence messages. This overhead can be included in the time prediction to access memory. However, we found that when predicting even simple access patterns such as copies, the overhead of coherence is overlapped with other memory access. It seems that some coherence messages can be performed in parallel with some memory access, but not all kinds of them. We believe that these differences come from the path used by coherence messages: if both coherence messages and memory accesses use the same physical path (*e.g.,* the bus connecting private caches together) they cannot be performed in parallel, but when coherence and memory access use different path (*e.g.,* coherence uses interconnect bus and memory access uses the memory channel) they can be performed in parallel. However to build such a model we need to know the choice made by hardware designers about the coherence protocol: and this model would not be portable on different architectures, which is one of the hard specification of our model.

Figures 2.9 and 2.10 illustrate the issues we encountered. We can see on Figure 2.9 that the cost of an `RFO` message does not only depend on the level of cache involved but also on the state of the cache line requested. The bandwidth plotted in Figure 2.9 represents the bandwidth used by coherence messages, in this case the `RFO`. This bandwidth is the subtraction of the bandwidth of a store hit on exclusive cache lines and the bandwidth of a store miss on cache lines in one of the modified, exclusive, and shared states. This represents the bandwidth used by caches to maintain coherence. Since the cost of an `RFO` depends is different when the cache lines accessed are dirty, we deduced that it is combined with a write back: on a store miss on modified cache line, the cache line might be written back to memory before being modified by the new request. But, as shown on Figure 2.10, the cost of a write back is higher then the cost of the `RFO`. The performance of write back messages was computed as the subtraction of the bandwidth of a load hit and the performance of a load miss on modified cache lines.

To keep our model as generic as possible we choose another method to build the memory model. Instead we built a memory model based on benchmarks. This idea is already used to

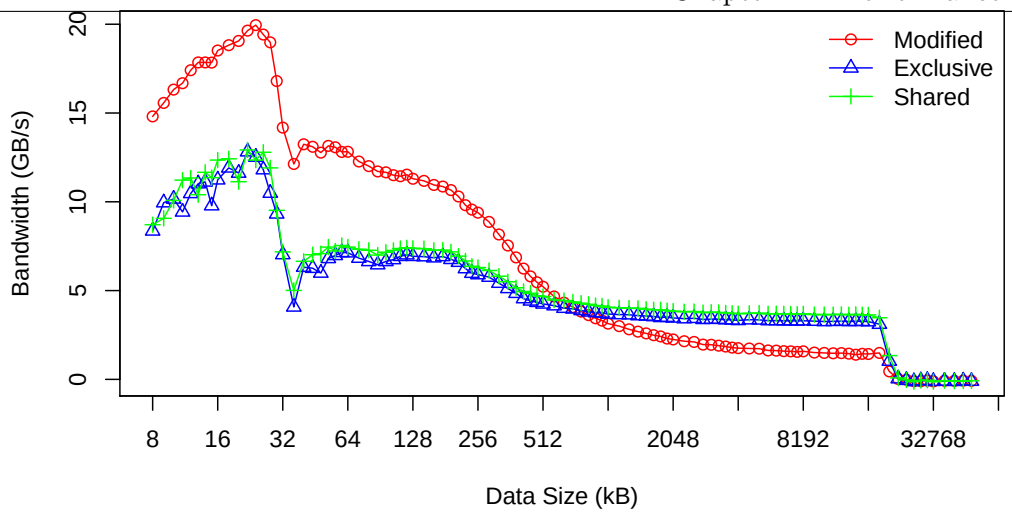


Figure 2.9: Cost of an `RFO` message depending on the state of the cache line involved.

model memory access cost in NUMA architecture [63] but – to our knowledge – had not yet been investigated for modeling cache performance. Benchmarks are designed to hide hardware complexity and mechanisms hard to understand or model. This also keeps the model generic since the same set of benchmarks can be run on several architectures and be used as the basic blocks for predicting memory access performance.

2.6 Conclusion

The contributions presented in this chapter are two-fold. We showed in a first section, how to automatically retrieve instruction performance. We also provided a method to detect instruction sharing execution ports in super-scalar pipeline. With a careful benchmarking methodology all the parameters required to build an analytical model of the on-core can be found by experience.

In a second part we presented how to use such performance data to build a power aware performance model on the SCC chip. This model allows performance and power consumption prediction for power aware performance optimization. It was presented in a paper called Performance modeling for power consumption reduction on SCC and was accepted at the 4th Many-core Applications Research Community (MARC) Symposium.

In a third section, we investigated how to model cache coherent memory hierarchies. We presented the few experiments advocating the use of benchmarks directly in a memory model rather than building a full analytical model for the memory. We analyzed the parameters that have to be taken into account for a fine modeling of cache hierarchies. By hiding memory hierarchy complexity in benchmarks and by using the output of these benchmarks as building blocks for the model we can build a generic memory model for cache-coherent memory architectures. We will describe our benchmark based memory model in Chapter 4. Understanding benchmark design and methodology helps apprehending our memory model. We will therefore first present our benchmarking framework in Chapter 3.

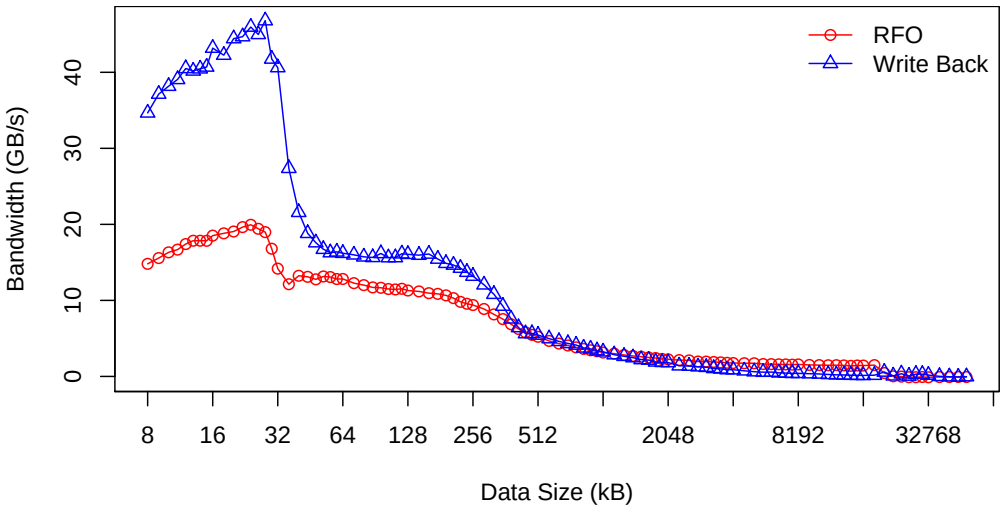


Figure 2.10: Cost of a write back message.

Contents

3.1 Problem Formulation **66**
3.1.1 Requirements of Benchmarks due to Cache Coherence 66
3.1.2 Building Reliable Benchmarks 66
3.2 Framework and Technical Choices **67**
3.2.1 Related Work 67
3.2.2 Framework Overview 69
3.2.3 Achieving Peak Memory Performance 70
3.3 A Language to ease Benchmark writing **71**
3.3.1 Language Description 71
3.3.2 Benchmark Compilation Framework 73
3.4 Benchmarking Memory Hierarchy **74**
3.4.1 Motivating Example 74
3.4.2 Automatic Generation of Coherence Protocol Benchmarks 76
3.4.3 Comparing Cache Architectures and Coherence Protocols 78
3.4.4 Guidelines for Improving Coherence Behavior 87
3.5 Conclusion **89**

Chapter 3

Designing Benchmarks for Memory Hierarchies

"It seems perfection is attained not when there is nothing more to add, but when there is nothing more to remove."
— Antoine de Saint-Exupéry

This chapter is dedicated to presenting a framework we developed to benchmark memory hierarchies of modern processors. Since we build a memory model upon the output of the benchmarks, they have strong requirements.

Section 3.1 will present problem of building benchmarks for memory hierarchies, especially what are the critical components of memory architecture that needs to be benchmarked. It also present the requirements of benchmarks in general and in the context of modeling. Section 3.2 presents the framework we developed and several implementation details about how to achieve close to peak memory performance. Section 3.3 presents our framework with an emphasis on the language we developed to ease benchmark writing and how to automatically generate benchmarks for a protocol given its automaton. This automatic generation of benchmarks can handle coherence protocols based on automaton, *i.e.* `MESI` like protocols or firefly. And Section 3.4 illustrates some possible usages of the output of our benchmarks.

3.1 Problem Formulation

This section introduces the methodology we use to benchmark memory hierarchies. Since benchmarking requires usage of a framework, we will also collect the requirements on the tool to be used to lead this study. We will also compare these requirements with existing tools and justify our technical decisions.

3.1.1 Requirements of Benchmarks due to Cache Coherence

Chapter 2 showed us that cache coherence has to be taken into account when build a memory model for cache coherent architectures. In this section, we are going to investigate the specific requirements of benchmark tools to perform correct memory experience with regards of cache coherence.

Setting Buffers in a Given State Since we aim at building a memory model taking cache coherence into account we need of being able to control the state, with regards to coherence protocol, of memory chunks involved in the benchmark. We can control the state of memory, *i.e.* of the cache lines, by running memory operation prior to the start the performance measurement. We need the framework to be able to only measure subset of the full benchmark. This way we can write a prologue responsible for setting cache lines of the system in a particular state. After what we can start the real experiment and record its performance.

Parallel Benchmarks Memory architectures are parallel, *e.g.,* several hardware threads have private caches and several software threads can access memory at the same time. Therefore, we have to be able to build parallel benchmarks. Also the location of data is important and has an impact on cache performance. To be able to reproduce every placement in the benchmarking tool-chain to chose, has to provide process placement capability. This means that for a particular parallel benchmark we need to be able to bind software threads or precess to hardware cores in different manner. This will allow us to investigate the impact of process placement on performance.

3.1.2 Building Reliable Benchmarks

We aim at building an accurate memory model upon this benchmark tool-chain, therefore we need accurate and reliable runtime measurements. To ensure this property we need to perform statistic collection on different runs of the benchmark. This is not a hard requirement of the tool-chain because it can be done by done by ourselves. But this feature would be a plus.

We should say here that, for reliability matters, selecting one performance measurement among a set of measures is a choice to make with particular care. If performance of several runs are recorded, reporting statistics like average run time, standard deviation *etc* is important. But to bring this information into a model, it is easier to select a single value. In the methodology we developed, we chose to only select the best measured performance. We choose to report only the best performance because, when modeling hardware, people are usually interested in peak performance

of the machine. Moreover the best performance measurement is an actual measurement. It can be easier to explain than the average – which is not a really observed performance – and can be a value that can never actually be observed.

A benchmark with performance exhibiting a high standard deviation should not be used as a reliable metric to model an architecture. Instead understanding what affects so much performance should be understood to better control the hardware and/or the test to find another way to capture a part of the architecture behavior.

A lack of stability in a benchmark often comes from system level issue. For instance Intel processors have a *Turbo Boost* feature that allows CPU to increase their frequency under sequential loads. Also, to save energy, most operating systems can change the frequency of processors depending on run time activity, this is called frequency scaling and can lead to performance difference between different runs. Since the goal of modeling hardware is to understand and reproduce its behavior, reproducibility of the benchmarks is a major concern. That is why we pay so much attention to keeping the standard deviation of our benchmark output low. If a benchmark leads to non-stable results, *i.e.* have a high standard deviation, we do not use it to model the architecture. We do not use benchmarks with a standard deviation higher than 10% of the average value. Indeed, with such results we would not be able to choose the correct data among the list: they would be too scattered. A high standard deviation can betray either a uncontrolled experiment environment (*e.g., Turbo Boost* still enabled), or a parameter that can vary from a run to the other. It can be the case if thread synchronization has net been handled to make the benchmark reproducible.

An Extensible and Lasting Framework We did not know the number and the list of benchmarks needed to model a memory architecture prior to building the model itself. Therefore we need a framework easily extensible, where adding more benchmarks can be done easily and quickly.

We need a stable benchmarking framework so that it can be used to build a wide range of hardware tests depending on the need of users. Since our approach relies on benchmarks to abstract hardware complexity and ease memory modeling, we do not want having to rewrite benchmarks every time an new architecture is released or when trying to model a new architecture.

3.2 Framework and Technical Choices

With the requirements highlighted in the previous section, we investigated the existing benchmarking frameworks available.

3.2.1 Related Work

LIKWID is a framework designed for rapid benchmarking [80]. It fits the need for an extensible framework as well as precise performance recording. In order to characterize performance features, a number of iteration can be specified on the command line to run the benchmark several time. For instance, Listing 3.1 shows how to measure the bandwidth of the L1 cache of a processor (assuming L1 cache is larger than 20 kB).

```
./likwid-bench  -t load -g 1 -w S1:20kB:1 -i 100
```

Listing 3.1: LIKWID usage example: Measuring L1 cache bandwidth by running the load benchmark 100 times. With 1 thread pined on socket 1 reading a 20 kB buffer.

However synchronization cannot be handled: every run of a benchmark consists in a call to a function. For this reason we cannot use LIKWID to set memory in a particular state before performing time measurement.

The STREAM benchmark targets memory [57]. But, like LIKWID, synchronization and benchmark preliminary cannot be handled with STREAM. Moreover the STREAM benchmark is not easily extensible since and code modification have to be made for every change we need to make in the benchmark set. For our purpose, a careful handling of synchronization is important. As explained in Section 3.1.1, we need to be able to set buffers in cache in a controlled state to measure the impact of cache coherence on memory performance. This cannot be done with tools such as LIKWID or STREAM and this tools are therefore not suitable to our needs.

The BenchIT benchmarking framework allows measuring a wide range of performance metrics [49]. But the exact data we need are not in the default kernel released with BenchIT. BenchIT can be extended by adding more benchmark kernels into the tool-chain but adding such kernel is very verbose. Both the kernel and thread synchronization have to be handled with standard library calls.

MicroCreator, part of the MicroPerf Tools, allows designing of low level benchmarks [11]. It takes as an input an XML file describing the benchmark kernel to be generated. It can produce a large number of kernels with a relatively small description. For instance, for the input description shown in Listing 3.2, MicroCreator generates 512 kernels (all the combination of 8 load or store).

```
<instruction>
    <operation>movapd</operation>
        <memory>
            <register>
                <name>r1</name>
            </register>
            <offset>0</offset>
        </memory>
        <register>
            <phyName>%xmm</phyName>
            <min>0</min>
            <max>8</max>
        </register>
    <swap_after_unroll/>
</instruction>

<unrolling>
     <min>1</min>
     <max>8</max>
     <progress>1</progress>
</unrolling>
```

Listing 3.2: MicroCreator kernel description.

Using this tool to generate the kernels for our framework is an promising opportunity. By adding calls to threading libraries in the prologue and epilogue, synchronization and thread spawning can be achieved. But it was not released as an open source software at the time we developed out framework, therefore we could not use it.

To our knowledge there is no existing software specifically dedicated to performance measurement of cache coherence. Yet, tools such as P-Ray focus on memory hierarchies and how to detect hardware specification through benchmarking [29]. While this approach is quite close to ours, they do not take cache coherence into account. While our approach is manly focused on cache coherence.

We will present our framework into mode details in the next section.

3.2.2 Framework Overview

The framework we developed is made out of a language, a compiler and a library. We will go into more details about it in Section 3.3. In this section we are going to show how we fulfilled each of the requirements presented in Section 3.1.

Setting Buffers in a Given State We decided to add a keyword in the language to specify what part of the benchmark has to be measured and what is the preamble. Benchmark written with our language can call benchmarking functions. The call to specific memory function in the preamble can help controlling the state of memory prior to performance measurement.

Parallel Benchmarks In order to build parallel benchmarks with our framework, the code generated by our compiler is parallelized with the OpenMP runtime. Also, the language features parallel construction: for each function call, the thread in charge to run the function is specified. The binding between hardware and software threads is delegated to the OpenMP runtime. Binding OpenMP threads can be done thanks to environment variables.

Reliability For reliability purpose our framework automatically runs several time every benchmark. For instance, for every execution of a benchmarks the performance of every single run is recorded. The performance of each of these runs are reported into a csv file with statistics such as average and standard deviation. Since this a spreadsheet format every statistic that are relevant for the end user can be automatically computed. The best performance recorded is also saved in a separated file for a quick overview of the performance of the benchmark.

Extensible Framework Building an extensible framework was the main goal we pursued. This was the primary reason why we designed a framework based on a language. Indeed, this helps user writing new benchmarks to understand a particular behavior. A compiler is used to generate the machine code corresponding to the benchmark written by the user. We also provide a library embedding functions often used in benchmarks. We developed several benchmarking functions to help users achieve peak memory performance. These functions are a variety of *load* and *store* operations. The different memory access patterns performed by these functions are:

sequential access: every byte of memory within the range given by user are accessed.

stride access: the stride is given by the user: only some bytes separated by the stride parameter are accessed.

a specialization of the stride access: where the stride is chosen to be exactly the size of the cache line. This is useful to measure the latency of a cache level because every access are made to a different cache line.

The user can add benchmarking functions to the library in order to extend the memory access pattern or operation the tool chain is able to perform. For instance we could add functions performing non temporal memory operations in order to see the impact of bypassing caches. User defined functions can be called from the benchmark description just like standard functions.

3.2.3 Achieving Peak Memory Performance

Peak memory performance needs to be reached in order to give valuable feedback to benchmarks users. In order to achieve such performance, we used several optimization already presented in Chapter 1: vector instructions, loop unrolling and avoiding TLB misses. We call here peak memory performance the maximum sustainable memory bandwidth. It mat vary depending on the cache level accessed, the spatial locality or any parameter affecting a memory access performance.

`SSE` and `AVX` We use vector instructions to access memory because it allows putting more stress on memory bandwidth by issuing larger memory access within one CPU cycle. Since we target `x86` architecture we used `SSE` (Streaming SIMD Extensions) instructions or `AVX` instructions when the architecture supports it. It is interesting to note that on the Sandy-Bridge architecture, on benchmarks solely composed of loads, using `SSE` or `AVX` instructions does not increase performance. This can be explained because this micro-architecture L1 cache features two 128 bits ports for loads per cycle. Therefore a 256 bit `AVX` load uses the two ports and only 1 AVX instruction can be serviced per cycles. While the L1 cache can sustain 2 128 bit `SSE` instructions per cycle. This leads to the exact same performance.

Avoiding TLB misses TLB misses present a high overhead because they require a full virtual-to-physical address translation by the Memory Management Unit (MMU) or by the operation system, which involves the traversal of up to 4 levels of page table stored in main memory. In order to avoid TLB misses several ways are available. The first is to rearrange memory accesses to keep accesses to the same page close to each other to avoid polluting the TLB with accesses to other pages. This can be achieved by changing the data layout or the order of accesses to variables. Since we cannot change the order of memory accesses because it is defined by the user. The only way to avoid it in our case is to reduce the number of pages accesses. To achieve this optimization, we try to map huge pages (available since kernel 2.6.23 [68]), if huge pages are not available we use regular sized pages.

Disabling Prefetchers An interesting question when benchmarking memory hierarchy is to disable or not to disable prefetchers. Disabling it usually helps better understanding of the cache behavior because prefetchers can hide some latencies. However when real applications are running prefetchers are enabled and observations made on benchmarks with prefetchers disabled can not be reproduced on real applications. Since we aim using the output of our benchmarks to hide hardware complexity and build a memory model able to reflect real hardware performance we choose to let prefetchers enabled when running the benchmarks. Moreover if we need to understand a particular behavior of memory hierarchy, we can still run benchmarks with hardware prefetchers disabled if we think this can help our understanding. But for modeling purpose we use benchmarks with prefetchers enabled.

3.3 A Language to ease Benchmark writing

In this section we describe precisely the syntax of the language we developed and the organization of the framework.

3.3.1 Language Description

Our benchmarking language allows rapid benchmark prototyping. It can be decomposed in three parts. The first one is used to declare *streams*. *Streams* are contiguous chunks of memory of a size given by the user. It can be hard-coded in the description of the benchmark or with the keyword *runtime* meaning that the size will be given on the command line of the binary. The syntax used to declare stream is described in listing 3.3. The name of streams has to follow the regular expression: `[a-zA-Z]+[a-zA-Z0-9_]*` and the specification of a constant sized stream should follow the regular expression `[0-9]+(KB|MB|GB|`ϵ`)`[1].

```
name =  runtime; |       // size will be given at run time
        constant_size;   // hard-coded size
```

Listing 3.3: Stream Declaration Syntax.

The second part of a benchmark describes action to be performed before the real benchmark. The syntax used to describe the preliminary of the benchmark is described in Listing 3.4. The regular expression describing thread that should run the benchmark function is: `[0-9]+(,[0-9]+)* | [0-9]+-[0-9]+`. It is either a comma separated list of threads (or a single thread) or a range a threads.

```
thread:threadset.benchmark_function(parameters);
```

Listing 3.4: Benchmark Preliminary Syntax.

[1] ϵ is the empty word.

And the last part of a benchmark describes the piece of the benchmark that needs to be timed. The syntax of the body of the benchmark itself is described in Listing 3.5 where *benchmark body* follows the same syntax as the preliminary description.

```
TIME(benchmark body);
```

Listing 3.5: Benchmark Body Syntax.

An example of a full benchmark description is shown in Listing 3.6. The semantic of this example is the following:

1. We declare 2 streams. The size of these streams will be given at runtime with a command line argument.
2. Threads 0 to 1 load the first stream (named *s0*).
3. Thread 0 store stream *s0*.
4. Thread 1 stores stream *s1*.
5. Only the performance of the last step is recorded. This step consists in thread 0 loading the stream *s1*.

Note that there are only synchronizations between the benchmark preliminary and body to ensure the preliminary is over before recording performance. This means that steps 3 and 4 are actually performed at the same time.

```
s0 = runtime;
s1 = runtime;

thread:0-1.load(s0);
thread:0.store(s0);
thread:1.store(s1);

TIME(
thread:0.load(s1);
);
```

Listing 3.6: A full Benchmark Example.

The benchmarking functions *load* and *store* are part of the default functions released with our library. The functions used in benchmarks have to be in the library and a benchmark only makes sense if users are aware of the meaning of the functions used in the benchmark description. Listings 3.7 and 3.8 present the assembly code used to perform the *load* and *store* memory operations. The code version presented are `SSE` versions.

```
_loop:
  movaps (%rbx), %xmm0;
  movaps 16(%rbx), %xmm0;
  movaps 32(%rbx), %xmm0;
  movaps 48(%rbx), %xmm0;
  movaps 64(%rbx), %xmm0;
  movaps 80(%rbx), %xmm0;
  movaps 96(%rbx), %xmm0;
  movaps 112(%rbx), %xmm0;
  add $128, %rbx;
  sub $128, %rcx;
  jnz _loop;
```

Listing 3.7: Load Function written in Assembly with `SSE` extension.

```
_loop:
  movaps %xmm0, (%rbx);
  movaps %xmm0, 16(%rbx);
  movaps %xmm0, 32(%rbx);
  movaps %xmm0, 48(%rbx);
  movaps %xmm0, 64(%rbx);
  movaps %xmm0, 80(%rbx);
  movaps %xmm0, 96(%rbx);
  movaps %xmm0, 112(%rbx);
  add $128, %rbx;
  sub $128, %rcx;
  jnz _loop;
```

Listing 3.8: Store Function written in Assembly with `SSE` extension.

User define micro-benchmarks can be added into the library and called from the benchmark description language with the same semantic as we showed with the *load* and *store* micro-benchmarks. To call a user-defined micro-benchmarks from the language, the name of the function defining the micro-benchmark has to be called within a thread with the usual syntax:
`thread:range.symbol(args)`.

3.3.2 Benchmark Compilation Framework

In order to run a benchmark using our framework users have to first write it with the language described in the previous section and compile it with the tools we provide. Figure 3.1 is an overview of our framework. Our compiler reads the user benchmark specification and generates the corresponding C code with OpenMP pragma. This regular C code can be compiled with any C compiler supporting OpenMP and linked against our library, this results in an binary than can be ran on the target machine. We also provide a shell script embedding the compilation of the user specification, the C code compilation and linking in a single command. This keeps the benchmark compilation as simple as it should be by hiding long and tiresome compilation command lines.

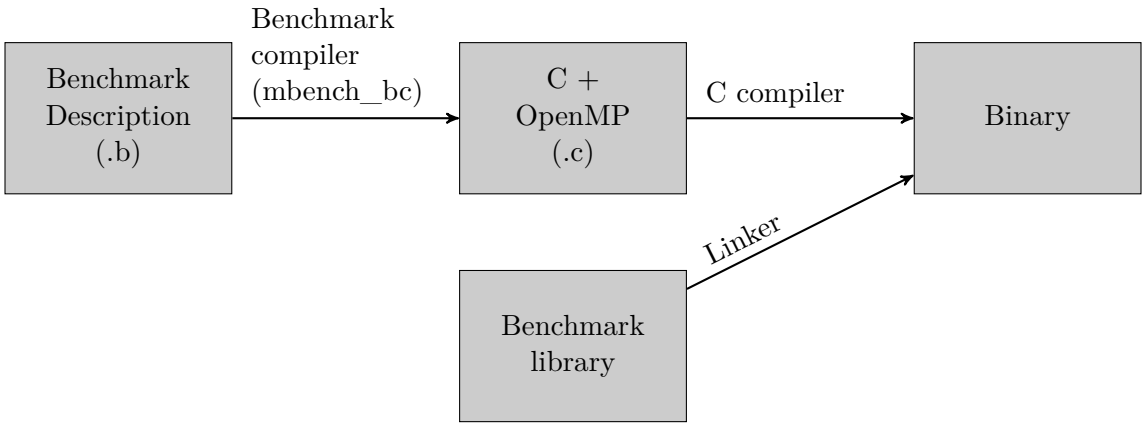


Figure 3.1: Benchmark compilation framework.

The benchmarking library is composed of two kind of functions: *i)* helper functions and *ii)* benchmarking functions. Users can add benchmarking functions to the framework, this allow them to extend the library to run specific benchmarks.

In order to try to control as much as possible the state and the data present in the cache of each processors, we flush all streams from the cache of the threads involved in the benchmark. This is done by calling a function that walks through the whole stream and use the `x86` instruction *clflush* to evince every address of the stream from the cache of the running thread. Therefore before every run of a benchmark, all caches are flushed from the data used in the experience and noise due to residual data in caches is eliminated as much as possible.

The code generated is parallelized with OpenMP directives. We chose this implementation of shared memory programming paradigm mainly because of its simplicity. But also because the binding of software threads to hardware cores can be controlled easily thanks to environment variables.

The benchmarking framework we presented in the previous section is available for download from https://github.com/bputigny/mbench.

3.4 Benchmarking Memory Hierarchy

Benchmarking memory hierarchy is not an end in itself, it is a tool to help understanding software performance and find applications optimizations. Section 3.4.1 presents the output of some benchmarks we ran on different architectures and exhibits counter-intuitive results. Section 3.4.4 presents several guidelines to help avoiding coherence traffic from spoiling memory performance. And Section 3.4.3 compares cache performance on several general purpose processors. This section also illustrates that benchmarking can help understanding poor performance of software due to poorly designed cache coherence protocols.

3.4.1 Motivating Example

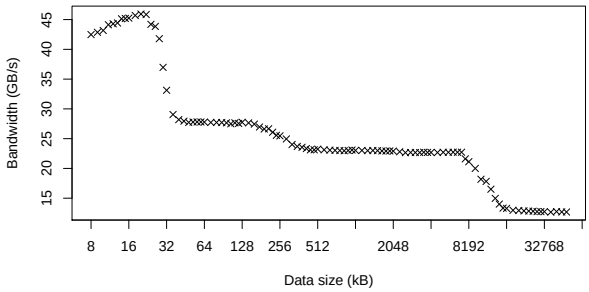
In order to get peak cache performance we run the benchmark called *load hit exclusive* (abbreviated *lhe*). This benchmark is described in Listing 3.9. It consists in bringing data to the cache of a processor and then record the performance to access this chunk of memory again.

```
s0 = runtime;

thread:0.load(s0);

time(thread:0.load(s0););
```

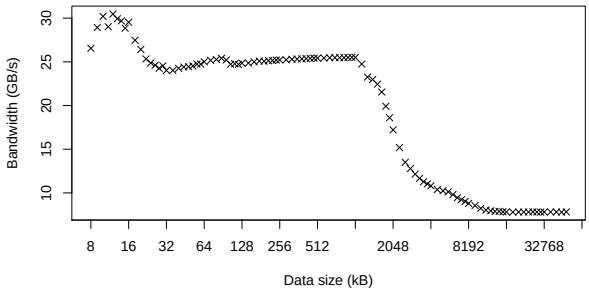
Listing 3.9: The Load Hit Exclusive Benchmark

The performance of this benchmark allows us to check the number of cache available on the hardware. It should be noted that we measure the runtime of the benchmark, but we present

the performance results as a bandwidth because Figures 3.2a and 3.2b show the output of the *lhe* benchmark on two different `x86` architectures: Intel Nehalem and AMD Bulldozer. We can see that both these architectures feature 3 levels of cache. The size of each level of cache can be found on these figures it is the data size where performance drop.



(a) On a Nehalem micro-architecture: the Intel Xeon X5650 processor.



(b) On a Bulldozer micro-architecture: the AMD Opteron 6272 processor.

Figure 3.2: Load Hit Exclusive Benchmark results on two different Micro-Architectures.

The Nehalem architecture features a 32kB L1 cache delivering up to 45GB/s bandwidth, a 256kB L2 cache with 28GB/s bandwidth and a 12MB L3. However on Figure 3.2a it seem that cache are smaller. It is especially visible for the L3 cache. This can be explained by conflict misses virtually reducing the real cache size.

The Bulldozer architecture feature a 16kB L1 cache with peak performance of 32GB/s bandwidth, a 2MB L2 cache with up to 25GB/s throughput and a 6MB L3 cache with a bandwidth of about 10GB/s. Unlike the Nehalem micro-architecture, the cache sizes observed on Figure 3.2b are the same as values provided by the constructor. This probably comes from the fact that Bulldozer's L2 and L3 caches are 16-way associative while Nehalem is 8-way associative. This can significantly reduce the number of conflict misses in the AMD architecture.

While the *lhe* benchmark is a very simple benchmark it already allows understanding the impact of hardware design choices on memory performance. But this benchmark does not involve coherence, we can build more complex memory access patterns to gauge how coherence impacts memory performance. An interesting memory access pattern consists in loading an address that is present in another cache of the processor. We call this benchmark load miss exclusive (abbreviated *lme*). Listing 3.10 shows the code of this benchmark.

```
s0 = runtime;

thread:1.load(s0);

TIME(thread:0.load(s0););
```

Listing 3.10: The Load Miss Exclusive Benchmark Code.

3.4.2 Automatic Generation of Coherence Protocol Benchmarks

The algorithm used to set memory chunks in a given state is described in Algorithm 1. This algorithm is not `MESI` specific and can be applied to any automaton describing a cache coherence protocol. The automaton describing such a protocol is defined by:

Q the set of all possible states for cache lines (*e.g.,* $Q = \{\text{M, E, S, I}\}$ is the `MESI` protocol.

Σ the set of cache events, *e.g.,* for the `MESI` protocol $\Sigma = \{$`LocalRead`, `LocalWrite`, `SnoopRead`, `RFO`, `Inv`$\}$

δ the transition function $\delta : Q \times \Sigma \rightarrow Q \times \Sigma$. This is not the usual definition of transition functions. We add an event in the return type of δ to express that transitions can trigger other coherence messages (*e.g.,* in the `MESI` protocol $\delta(\text{S}, \texttt{LocalWrite}) = (\text{M}, \texttt{Inv})$: when writing to a shared cache line, the local cache has to broadcast an invalidation).

q_0 the initial state a cache lines (*e.g.,* `I` for invalid, in the `MESI` protocol).

From this protocol definition, we can 2-partition Σ in Σ_l and Σ_r where:

$$\Sigma_r = \{e \in \Sigma | \exists (s, e) \in Q \times \Sigma, (s, e) \in \delta(Q, \Sigma)\}$$
$$\Sigma_l = \Sigma \setminus \Sigma_r$$

Σ_r represents the coherence messages, *i.e.* events that are triggered by a remote cache. And Σ_l are local event (*i.e.* triggered on the local cache by load and store instructions).

The function writeCode should be provided for the target cache architecture (*i.e.* cache coherence protocol). Listing 3.11 shows the pseudo-code implementation of the writeCode function for the `MESI` automaton.

```
printLocalEvent(int e, int thread_id, char *chunk) {
    switch(e) {
        case LocalRead:
            printf("thread:%d.load(%s);", thread_id, chunk);
            break;
        case LocalWrite:
            printf("thread:%d.store(%s);", thread_id, chunk);
            break;
        case SnoopRead:
            printf("thread:%d.load(%s);", thread_id+1, chunk);
            break;
    }
}
```

Listing 3.11: writeCode function implementation for the `MESI` protocol.

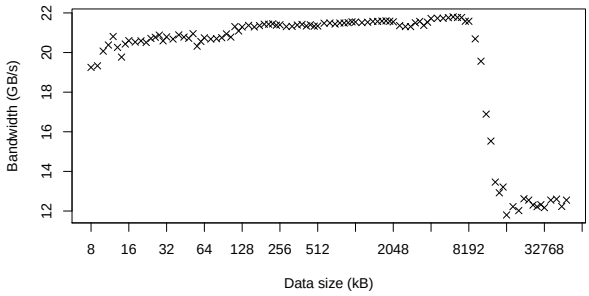
Figures 3.3a and 3.3b present the output of this benchmark on a Nehalem and a Bulldozer architecture. We ran the benchmark with the two threads bound on different cores of the same processor.

```
Given the cache coherence protocol defined by: Q, Σ, δ, q0:

setChunk(s_t, t, chunk): ;              // Sets chunk in state s_t for thread number t
Let (Q, Σ, δ, q0, F) be an automaton, with F = {s_t} ;     // The only final state is s_t
if ∃w ∈ Σ*|n ≤ |Q|, q_n ∈ F ; // w:  sequence of events to sets memory in state s_t
then
  | foreach evnet e in the sequence w do
  |   | writeCode(e, t, chunk)
  | end
else
  | Error: s_t cannot be reached
end

writeCode(e, t, chunk):
if e ∈ Σ_l then
  | printLocalEvent(e, t, chunk)
else
  | if ∃(s0, s1, e0) ∈ Q^2 × Σ_r|δ(s0, e0) = (s1, e) ;          // transition δ(s0, e0), fires e
  | then
  |   | setChunk(s0, t + 1, chunk)
  |   | writeCode(e0, t + 1, chunk)
  | else
  |   | Error: remote event e cannot be triggered
  | end
end
```

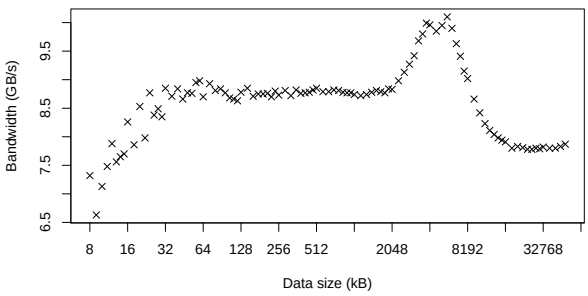
Algorithm 1: Setting chunks in a particular state of a coherence coherence protocol.

On both processors, the last level of cache is shared among all cores: this means that when data fit in L3 there is actually a hit and not a miss. Since the L3 cache of the bulldozer architecture is not inclusive, when the data set sizes between 2MB and 6MB it goes to L3 cache (after being evicted from lower cache level) and the performance we see are L3 cache hit. This explains why, on Figure 3.3b): performance is better when data fits in L3 than in L1 or L2. On smaller data set sizes, the performance is a bit better than memory performance: this means that the caches lines are probably retrieved from the inter L2 bus rather than from main memory. On the Nehalem architecture, the L3 cache is inclusive, this means that all cache lines present in all L1 and L2 caches of the socket are replicated in the shared L3. This explains why for all sizes fitting in L3 the performance is the same: it is the performance of the last level of cache. We can deduce that on a cache miss in L1 or L2, the lines are brought from L3, not through the bus from the cache holding it.

We presented the performance of a few memory access patterns to show how cache coherence affects performance. Even with the small number of memory patterns presented we saw different behavior on different hardware architectures. We also saw that even with knowledge about the architecture some results of the benchmarks are not intuitive. This emphasize the benefits from using benchmarks to both characterize hardware and build performance models.



(a) On a Nehalem micro-architecture: the Intel Xeon X5650 Processor.



(b) On a Bulldozer micro-architecture: the AMD Opteron 6272 processor.

Figure 3.3: Load Miss Exclusive Benchmark results on two different Micro-Architectures.

3.4.3 Comparing Cache Architectures and Coherence Protocols

In this section we are going to thoroughly benchmark several micro-architecture too understand the implication of cache design in terms of performance. This can also be used to determine the architecture that best suits the needs of an application in order to ease the choice of the hardware.

In order to fully characterize a cache hierarchy we run the set of benchmarks defined by the three following parameters:

the operation benchmarked: *load* or *store*. This allows getting read and write performance of the memory hierarchy.

cache hit or miss: When missing the cache the requested address can be brought to the CPU by another cache holding it. Benchmarking cache misses allows refining the model by getting the performance of such accesses rather than approximating it to a memory access.

the state of the cache line accessed: Modified, Exclusive, Shared or Invalid. We already saw that the state of cache line implies different coherence messages. By benchmarking all states of memory we can understand what memory access pattern produce a large amount of overhead on a given architecture.

This makes 16 possible benchmarks, however the benchmark *lhi* (Load Hit Invalid) does not make sense since hitting an invalid cache line is actually a miss. Moreover we do not use benchmarks on invalid data: it would only result in benchmarking the memory itself since it would never hit any cache on the processor.

We only chose to benchmark access to data in one of the three states *Modified*, *Exclusive* and *Shared* (and not other states such as *Forward* or *Owned*) because it keeps the benchmark set general enough to be applied to several hardware architectures without any adaptation. Indeed including states that are specific to a particular architecture (such as the *Owned* state that is only implemented in AMD cache architectures using the `MOESI` Protocol).

Comparing several Target Architectures

We are now going to compare different hardware architectures. This illustrates how our benchmarking tool-chain can be used to select an architecture for a particular task. We are going to compare three `x86` architectures, but we are going to see that despite the fact that these architectures share the same Instruction Set Architecture (ISA), they display significant cache behavior differences. Figures 3.4a, 3.4b 3.4c present the architecture compared: Intel Nehalem, Intel Sandy-Bridge and AMD Bulldozer.

We ran the 12 memory benchmarks briefly presented earlier:

LHE: Load Hit Exclusive. Only one thread in involved. It loads a chunk of memory – or a stream to used the same terminology as our benchmarking language – and measure the performance to access it again.

LHM: Load Hit Modified. A single thread is also involved, but unlike the *LHE* benchmark the thread stores the stream before recoding the time to read the whole stream again.

LHS: Load Hit Shared. In this benchmark, two threads are involved. Both the threads load a stream after what the performance of one thread accessing it is recorded.

LME: Load Miss Exclusive. Two threads are involved. One loads a stream, and the second measures the time needed to access this very same stream.

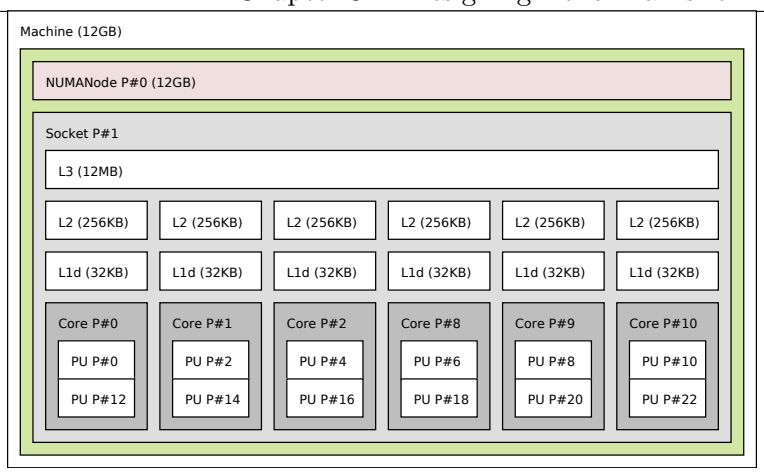
LMM: Load Miss Modified. Two threads are involved, the first one stores a stream and the second record the performance when reading this stream.

LMS: Load Miss Shared. Three threads are involved here. Two threads load a stream, then the third one records the performance to access it.

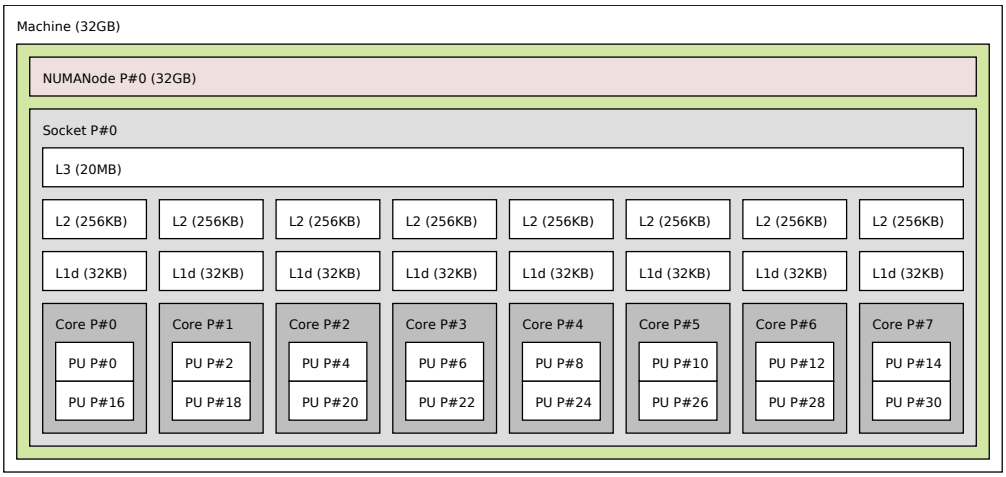
And the 6 other benchmarks are the same except that they measure the performance of writing the stream rather than reading it.

We do not aim at comparing the raw performance difference of the different hardware, instead we aim at comparing the behavior of cache regarding the memory access pattern. Therefore we will neither comment on peak cache performance nor on cache size but how the cache coherence protocol and hardware design choices affect performance. For this purpose we scaled the bandwidth presented in this section: we divided all measurements we made by the best performance recorded for the benchmark on the architecture studied. Therefore we do not present real bandwidth but a relative bandwidth comprised between 0 and 1. A relative bandwidth of 1 means that the corresponding size is where we got the best performance.

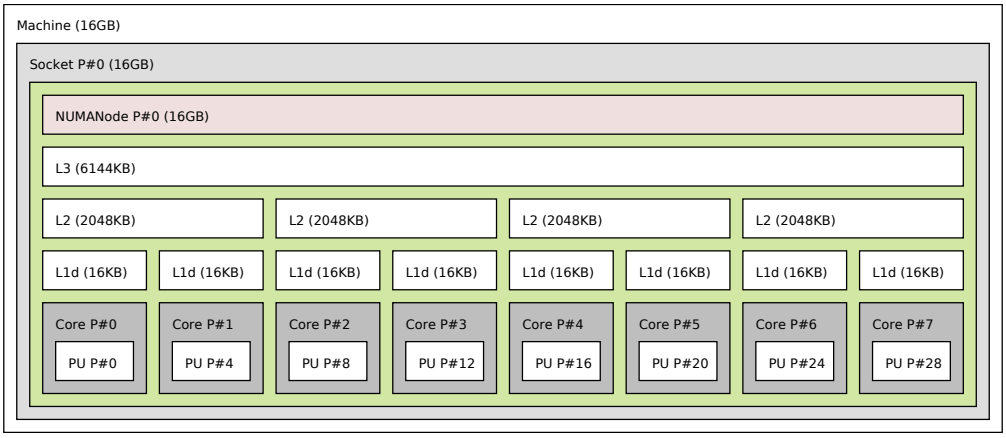
The Load Hit Benchmark Figure 3.5 presents the output of the load hit benchmarks (*i.e.* *lhm*, *lhe* and *lhs* benchmarks). Since all these load hit benchmarks do not involve cache coherence mechanisms, the output of all these benchmarks are the same and we only display the performance of one of them (*lhe*). Since no coherence traffic arises from these kind of accesses, the behavior of all architecture is the same: we obtain better performance when the data set fits in smaller cache level and the performance drops as soon as the data set accessed is too large to fit in a cache level.



(a) A Intel Nehalem micro-architecture socket.



(b) A Intel Sandy Bridge micro-architecture socket.



(c) A AMD Bulldozer micro-architecture socket.

Figure 3.4: Micro-Architectures Compared.

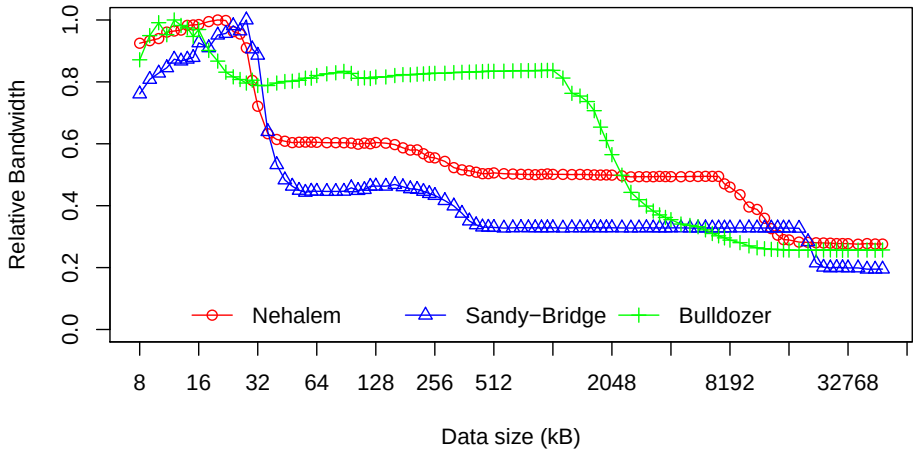
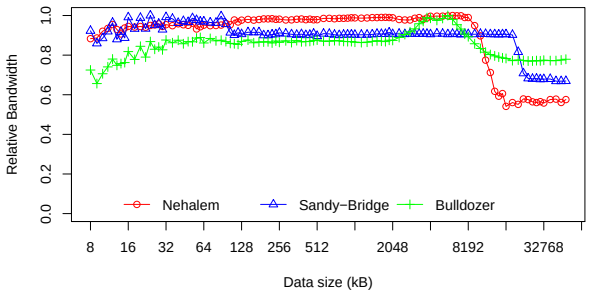
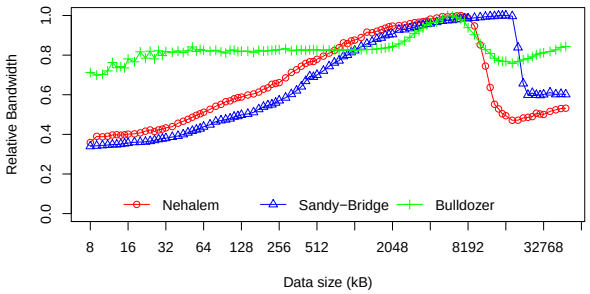


Figure 3.5: Load Hit Benchmark Comparison.

The Load Miss Benchmark The comparison of the Load Miss benchmark family is displayed on Figure 3.6. We observed the same behavior for all three architectures on the *lme* and *lms* benchmark: on both these benchmarks, caches line requested are present in a clean state in another cache of the processor. Thus it is no surprise that they behave the same way. The output of these 2 benchmarks is presented in Figure 3.6a. We present separately the output of the *lmm* benchmark in Figure 3.6b because it differs from accessing clean cache lines. Since cache accessed cache lines are dirty, it involves coherence traffic: the dirty cache lines have to be written back to memory or fetched from the cache holding the up to date value.



(a) Load Miss Benchmark on clean data (Exclusive and Shared states) for different architectures.



(b) Load Miss Benchmark on Modified data for different architectures.

Figure 3.6: Load Miss Benchmark Comparison.

It shows interesting behavior difference between the Intel architectures and the AMD one. As we can see on Figure 3.6a when loading clean cache lines from a remote processor, the Nehalem and Sandy-Bridge processor deliver a steady bandwidth[2] when data fit in L3[3]. However, on the Bulldozer processor, the performance when loading data set that fit in L1 or L2 cache are close

[2]This steady performance are equal the level 3 bandwidth, the relative scale of the plot do not allow seeing it but on a with a regular scale it can be verified.

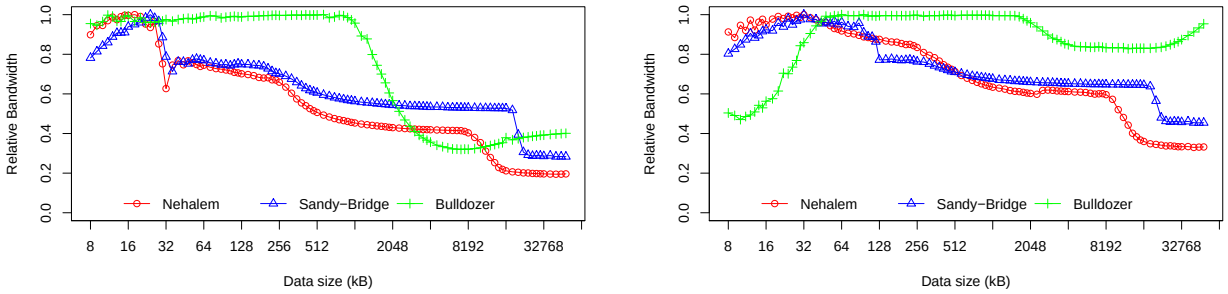
[3]We remind the reader that these benchmarks are ran on a single processor: since all cores share the L3 cache even if the benchmarks are built to perform misses, when data fit in L3 cache hit actually happen.

to memory performance. We explain these differences by the inclusive property of the last level of cache on Intel architectures. On these architectures, data that are in L1 and L2 caches are replicated in the L3 cache. Thus when a load request miss the level 1 or 2 of the cache hierarchy, the request goes to L3 which provides the cache line. This is why we obtain L3 cache performance when missing L1 or L2 caches.

The Bulldozer architecture does not have the inclusive property of the last level of cache. The argument for having inclusive caches is to speedup the inter-socket cache coherence by removing the need to check for lower level of cache when a request comes from the outside of the processor. But the drawback is that it wastes cache space because of the data replication: fewer memory addresses can be stored in inclusive cache hierarchies than in non-inclusive ones. However we can see that cache misses in L1 and L2 level are a bit faster than accessing memory: this can be explained by the cache holding the requested address suppling it to the processor through the bus.

If we now look at Figure 3.6b comparing performance of load miss on dirty cache lines we can see that the hardware implementation choice have an impact on performance. The Bulldozer processor shows exactly the same behavior when loading clean or dirty data. The coherence protocol used in AMD processors is the `MOESI` Protocol (*cf.* section cache coherence protocols in Section 1.3.3 on page 26) that allows sharing dirty cache lines. This explain that there is no difference of performance between the load miss benchmarks on this architecture. However on Intel architectures we can see that loading dirty data not present in our cache involved a high overhead compared to accessing clean data. This is due to the write back that happens in this event: the dirty cache line is written back to main memory to keep memory consistent.

Store Hit Benchmark Figure 3.7 presents the result of the store hit benchmark. The behavior



(a) Store Hit Benchmark on Non-Replicated Data (Exclusive and Modified states).

(b) Store Hit Benchmark on Replicated Data (Shared state).

Figure 3.7: Store Hit Benchmark Comparison.

of the store hit benchmark is the same as long as the cache performing the benchmark is the only one holding the addresses stored, *i.e.* the performance recorded for this benchmark are the same on the *she* and *shm* patterns. They are displayed in Figure 3.7a. Since cache lines in the exclusive or modified state are aware that they are the only cached version on the machine, the core can write directly to the cache without involving coherence. This explains why on both these benchmark the performance are the same. However when writing to shared cache lines, a choice has to be made:

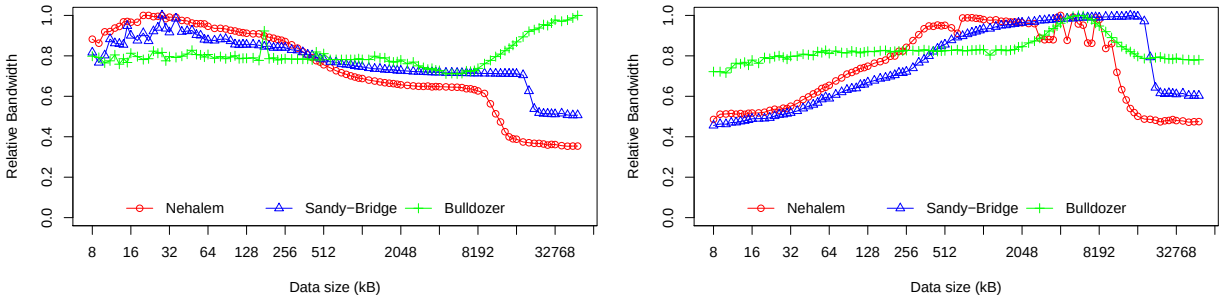
either other caches holding the address written have to invalidate their copies, or the value written has to be broadcasted to all caches sharing this particular address.

As we can see on Figure 3.7b, the Intel and AMD architectures have different behavior. This suggests that different hardware implementation choices were made. We can see that the overhead due to writing to shared cache line is far more important on the AMD architecture than on Intel architectures. In the `MOESI` Protocol (implemented in the Bulldozer architecture) shared cache lines can be dirty and cannot be written directly: before it has to transition to exclusive or modified state. In order to switch a cache line from shared to modified or exclusive state, a cache has to broadcast an invalidation for the address contained in the line. The caches holding the address respond to the invalidation and inform the cache issuing the invalidation is their version was shared or owned. If a cache had the line in the owned state, the line was dirty and the cache that issued the invalidation broadcast has to set the line into modified state. But if no cache has the line in owned state, then the line was clean and can be set to exclusive state. Thus, on the Bulldozer architecture, when writing to a shared cache line, the processor has to wait for other caches to respond to the invalidation, this explains the high overhead compared to writing to exclusive or modified cache lines. Since on the Nehalem and Sandy-Bridge architectures, cache lines are necessarily clean, writing to a shared cache line also involves an invalidation but waiting for a response to the invalidation request is not necessary. This explains why the overhead of writing to shared cache lines on Intel's architecture is faster.

Another remark about the performance of this benchmark on the Bulldozer architecture is that hitting the L1 cache is slower than hitting the L2 cache (which is intriguing since the L1 cache should be faster than level 2). The only plausible explanation we found is that when a store hits a shared cache line in the level 1 this cache broadcasts an invalidation to all the L1 caches (*i.e.* 7 others). Which takes more time than when the cache line is in L2 because the cache has to broadcast only 3 invalidation requests (since they are only four L2 caches on the processor).

Also we can explain that no matter cache line written are in exclusive or modified states, the performance in L1 and in L2 are the same. This is due to the fact that on this architecture the L1 is write through: this means that data written to L1 are also written to L2.

Store Miss Benchmark Figure 3.8 presents the results of the store miss benchmark on the 3 compared hardware architectures. Since the performance of the *sme* and *sms* are pretty similar on all the compared hardware, we only present the results of the *sme* benchmark on Figure 3.8a. We can see on this figure that the behavior the Bulldozer architecture differs from the one of the others. Remembering that the L1 is write though on this processor explains why the performance of the benchmarks in L1 and L2 cache are similar. However when the data set stored is wider than the last level of cache, performance gets better: it is faster to write to addresses that are not cached than writing to addresses hold in a cache on the processor. This can only be explained by the core writing to a cached address (cached in another cache that the one attached directly to the core store the stream) broadcasting the value it writes to the cache holding the same address. On the Nehalem and Sandy-Bridge architectures, since shared or exclusive cache lines are *clean*, writing to lines held in another cache only involves an invalidation request performance are better in lower levels of cache.



(a) Store Miss Benchmark on Clean data (Exclusive and Shared states).

(b) Store Miss Benchmark on Modified data.

Figure 3.8: Store Miss Benchmark Comparison.

However, as we can see on Figure 3.8b, the behavior is significantly different when writing to *dirty* cache lines. The Bulldozer architecture however behaves identically when writing to clean or dirty cache lines. Since the write is broadcasted the architecture performs the same operation whatsoever the state of the lines are.

Intel Dunnington micro-architecture

The Dunnington micro-architecture features up-to 6 cores per socket, every core has its own private level 1 cache of size 32 kB. The 3 MB level 2 cache is shared by pair of cores (one processor has therefore 3 separated L2 caches). And a 16 MB large level 3 cache shared among all the 6 cores. Figure 3.9 illustrates the architecture of a Dunnington processor. This architecture is based on the

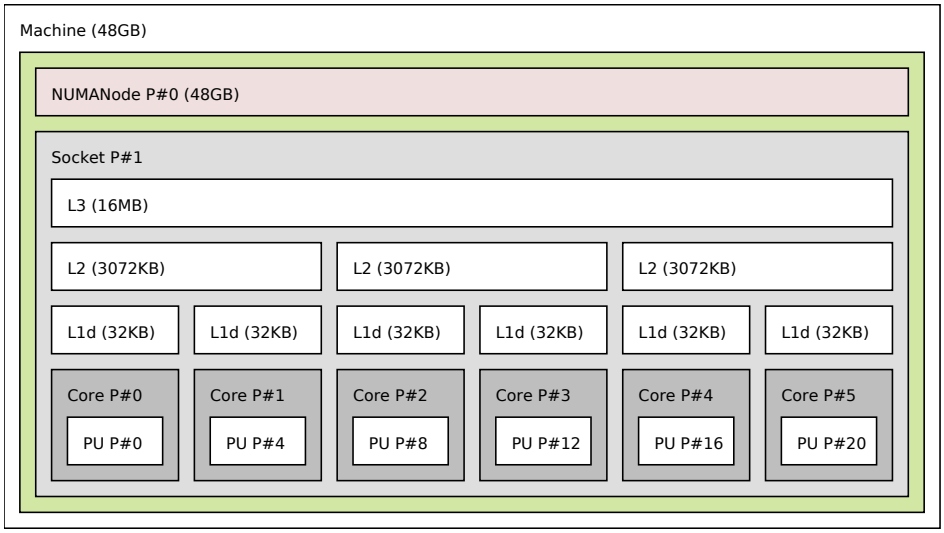


Figure 3.9: A Intel Dunnington micro-architecture Socket.

Core 2 micro-architecture that only features two levels of cache (and two cores per processor). The shared L3 cache has been added in order to decrease the cost of maintaining coherence between 6 cores.

Figure 3.10 presents the results of the store hit benchmark on this architecture. We can see

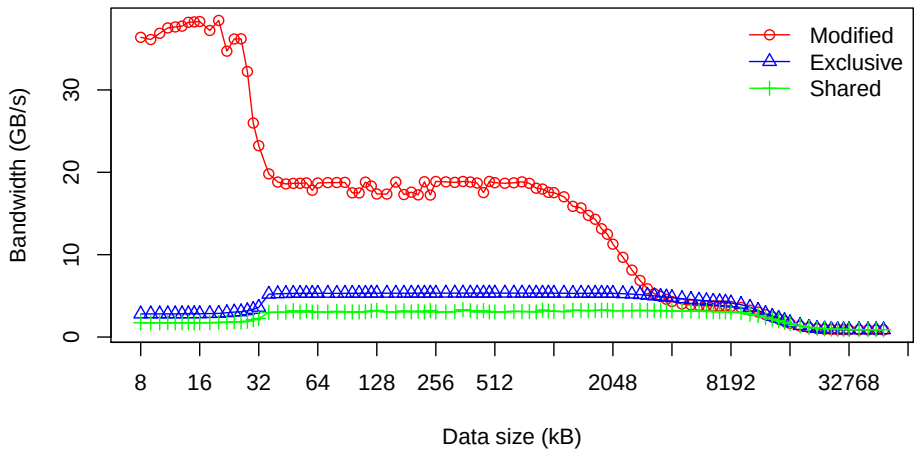


Figure 3.10: Output of the Store Hit Benchmark on the Dunnington Micro-Architecture.

that writing to exclusive is as slow as writing to shared caches lines. Of course this should not be the case since writing to exclusive cache lines should not require coherence while writing to shared cache lines requires an invalidation broadcast. This remark is true for the level 1 and level 2 caches. We believe that the exclusive state on this platform is managed just like the shared state. Indeed on the performance recorded by the whole set of benchmarks, the performance of benchmarks on exclusive and shared memory are always the same.

This example is a good illustration why modern complex hardware architectures should be carefully benchmarked and another argument for building memory models upon parameters measured by experience: it is the only way to capture the real hardware behavior, or achievable peak performance.

Parallel Benchmarks and Capacity

In order to take into account the impact of multiple threads accessing the memory hierarchy simultaneously, we also built parallel benchmarks. All threads involved in these benchmarks run the same code, with the same access pattern as sequential benchmarks.

For a given memory access pattern, we analyze the ratio between the sequential bandwidth multiplied by the number of threads, and the parallel bandwidth:

$$contention = \frac{n_{threads} \times bandwidth_{sequential}}{bandwidth_{parallel}}$$

A ratio of 1 means that parallel accesses from multiple threads do not disturb each other, *i.e.* each thread can use the same bandwidth as it would if it was running alone on the machine. A ratio greater than 1 is the factor by which each thread sees its available bandwidth divided by.

This ratio does not necessarily represent contention within caches or on the memory bus. We actually observed no cache contention on the Intel Sandy Bridge micro-architecture used for our

tests (while the AMD Bulldozer micro-architecture shows some). However the limited capacity of physical caches causes the ratio to increase when multiple threads try to place too much data in the shared L3 cache. They cause some *parallel capacity misses*, which look like cache contention on the benchmark outputs.

Listings 3.12, 3.13 show some code examples used to run the parallel benchmarks presented in Figure 3.11.

```
s0 = runtime;
...
s7 = runtime;

thread:0.load(s0);
thread:1.load(s1);
...
thread:7.load(s7);

time(
    thread:0.load(s0);
    thread:1.load(s1);
    ...
    thread:7.load(s7);
);
```

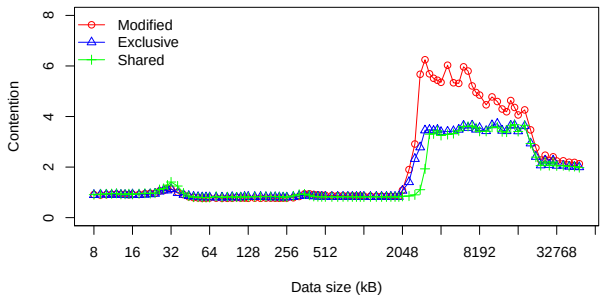
Listing 3.12: Code used to perform the *load hit exclusive* benchmark in parallel.

```
s0 = runtime;
...
s7 = runtime;

thread:1.load(s0);
thread:2.load(s0);
thread:2.load(s1);
thread:3.load(s1);
...
thread:0.load(s7);
thread:1.load(s7);

time(
    thread:0.load(s0);
    thread:1.load(s1);
    ...
    thread:7.load(s7);
);
```

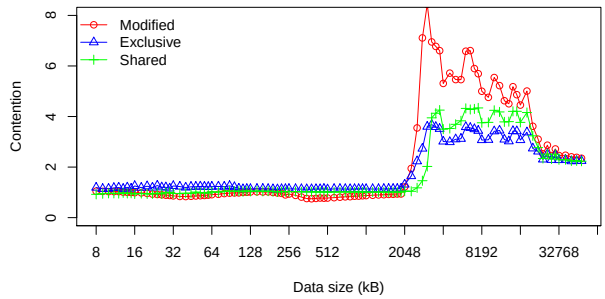
Listing 3.13: Code used to perform the *load miss shared* benchmark in parallel.

As shown on Figure 3.11, there is almost no contention on private caches: every independent cache can deliver the same bandwidth when it is accessed alone or when all private caches are accessed at the same time. This is particularly interesting because no contention appears even when coherence traffic is involved. However, as one would expect, contention seems to appear in shared resources such as L3 and memory. As explained above, the L3 cache contention is actually caused by parallel capacity misses caused by multiple threads sharing the overall L3 size.

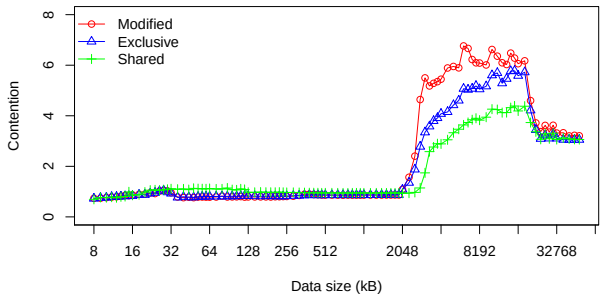
More interestingly the ratio depends on the state of cache lines accessed: accessing modified cache lines (in L3) always leads to more parallel performance decrease that accessing clean lines. Also, we can see on Figure 3.11c that there is more parallel issues when writing to exclusive cache lines than writing to shared ones. We could not explain this behavior, and it justifies further the idea to hide hardware complexity by using benchmarks: they capture more such puzzling hardware behavior than more abstracted analytical models, and we just use their outputs in our model.



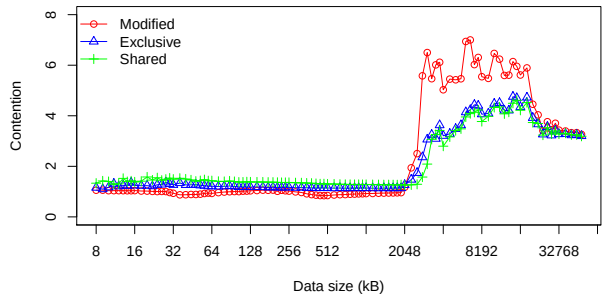
(a) Parallel Load Miss benchmarks.



(b) Parallel Load Miss benchmarks.



(c) Parallel Store Hit benchmarks.



(d) Parallel Store Miss benchmarks.

Figure 3.11: Parallel bandwidth ratio for several benchmarks on Intel Sandy Bridge processor with 8 threads running the code.

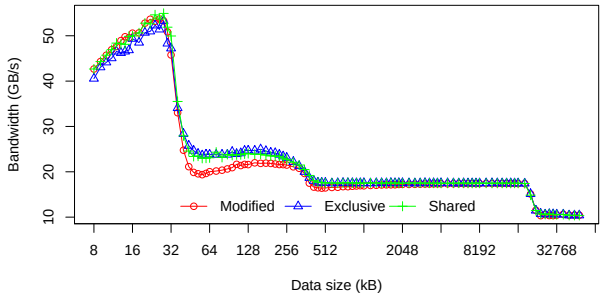
3.4.4 Guidelines for Improving Coherence Behavior

We saw that different choices in hardware design lead to different behaviors, however this report is not enough to help software engineers to better utilize hardware. Yet this can be used to detect what are the memory access patterns that lead to poor memory performance. By avoiding these patterns one can speedup memory access and thus reduce software runtime.

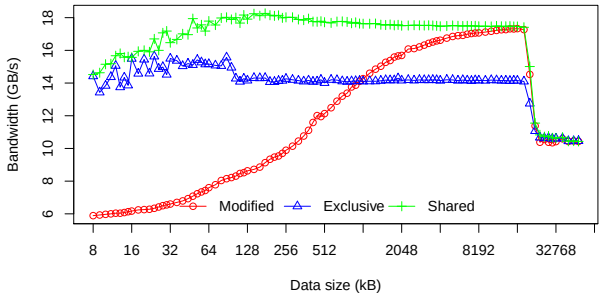
Patterns with poor performance are architecture dependent, and software has to be tuned for the targeted hardware. However, as we could see in the previous section, it seems that the overall behavior or a given memory hierarchy mainly depends on the cache coherence protocol chosen to maintain memory consistent. For instance we saw that the general behavior of the Nehalem and the Sandy-Bridge architectures are the same. This is because they use the same cache coherence protocol. However the Bulldozer processor, using a different cache coherence protocol behaves differently. Therefore, we can suppose guidelines for better utilizing memory hierarchies can be applied to all the architectures using the same cache coherence protocol.

In order to quantify the performance of memory access patterns between them, we present in Figure 3.12 the real bandwidth measured on the Sandy-Bridge processor for all benchmarks.

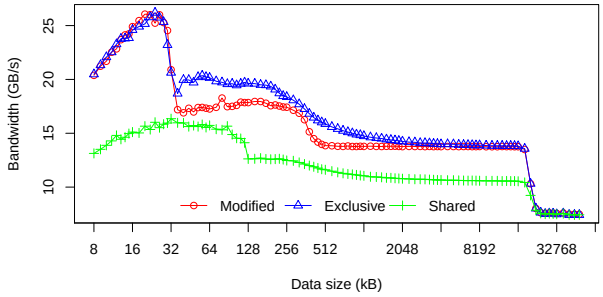
By looking at Figure 3.12b, we see that reading data modified by another core results in lower bandwidth compared to reading clean cache lines from other caches. In order to optimize an application using this kind of memory access, one can try to to change the thread process binding. However on some algorithms this is not possible. For instance, on a pipelined application or a



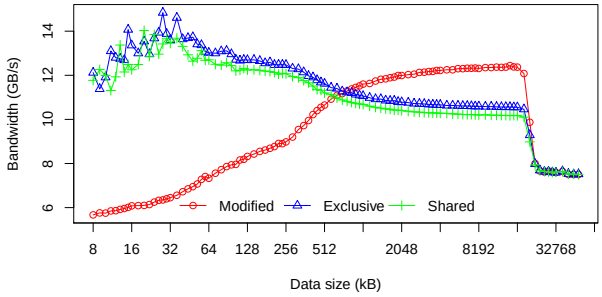
(a) Load Hit.



(b) Load Miss.



(c) Store Hit.



(d) Store Miss.

Figure 3.12: Full Benchmark Set results on Sandy-Bridge Micro-Architecture.

producer consumer work-flow this communication pattern is unavoidable. But we can see that the overhead due to coherence on this access pattern is more important in lower cache levels. If the application is tiled it can be worth trying to use larger tiles. Indeed, bigger data set will be more likely to stay in higher cache levels, it can decrease the overhead of the coherence traffic.

On the writing side, if we look at Figure 3.12c, we can see that writing to shared memory chunks is slower than writing to exclusive or modified data. This is due to the invalidation request needed on this kind of access in the `MESIF` Protocol. Yet the worst pattern when writing is writing to a memory location not present in the local cache as we can see on Figure 3.12d This remark justifies the *Owner-Computes Rule* often used for binding or scheduling in HPC software. This rule states that when computing the result of an arithmetic expression, the computation should be performed by the thread holding the left hand side of the expression (*i.e.* the variable that is stored). The key idea behind this rule is to avoid storing to memory location that are cached by another core. This memory access pattern also that happens with *false sharing*.

They are several intriguing observations about these plots: On Figure 3.12a and 3.12c, the steep increase in bandwidth in L1 cache is due to an overhead of our measurement function. This can also be observed on the 2 other figures but since the bandwidth are lower, it is less visible. On the miss benchmarks (Figure 3.12b and 3.12d) on modified cache lines, the very slow transition between L1/L2/L3 performance comes from the remaining dirty cache lines in lower cache levels that have to be written back to keep maintain cache coherence. On Figure 3.12b when data sets accessed are 32 kB to 20 MB wide, we failed to explain why would it be slower to access exclusive cache lines than shared ones. Especially since on the Nehalem architecture this performance gap

does not exist. On Figure 3.12d, writing to modified cache lines is slower than writing to exclusive caches lines in level 2 cache and in the beginning the L3. In the *shm* benchmark, if data set is larger than L1, when starting the benchmark, the L1 is full of dirty cache lines (the last 32 kB of the stream). Therefore when we start the benchmark, we write to the L1 a cache line that is no longer in the cache (the beginning of the buffer has been flushed to L2) a line is freed in order to hold the new address. Since the line freed is dirty it has to be written back to maintain cache coherence, this explains the gap between the modified and exclusive versions of this benchmark: on the exclusive version the lines are clean and do not have to be written back.

This explanation is also true for the small gap observed on Figure 3.12a between the modified and the exclusive states in L2 cache. On Figure 3.12d we did not find an explanation why store miss to modified cache lines is faster in the end of the L3 than access to clean cache lines. Again this gap is not present on the same benchmark ran on a Nehalem architecture.

3.5 Conclusion

We presented a language and tool for memory benchmarking. The tool presented is the first tool to our knowledge to measure on performance design choices for multi-core memory hierarchies, such as the coherency protocol and the different bandwidths.

Our framework differs from existing frameworks such as LIKWID [80] and STREAM [57] because we can select the part of benchmarks where performance measures have to be done. This allow us to set buffers in a controlled state to measure precisely the impact of the coherence protocol on cache performance. Because of this the benchmarks used in this chapter to characterize memory architecture could not be written with these benchmark or framework. We could have used BenchIT [49] or MicroPerf Tools [11] to write the benchmarks we used in this chapter. Because they are very extensible we could have written the micro-benchmarks we needed into these frameworks, and call them with careful synchronization to set memory is the required state. But, as we saw a large through this chapter, a large number of benchmarks have to be designed to fully characterize cache coherent memory hierarchies. Using these tools to write the wide range of benchmarks we used would take a lot of time and would be error prone.

We used our language to write several benchmarks to characterize multi-core processor memory hierarchy. The benchmarks presented in this chapter only use two hand-written micro-benchamrks (*load* and *store*), but more complex micro-benchmarks can be written and used in the benchmark description language.

By analyzing the output of memory benchmarks we can compare the choices made in hardware design and the impact it has on memory performance. This can help selecting the proper hardware for specific needs or it can guide future hardware designs. With a set of representative benchmarks we are able to characterizing the behavior of a memory hierarchy. And, with a careful analysis of this behavior we are able to provide guidelines to help utilizing memory more efficiently. Also show showed unexpected cache behavior on a particular cache architecture. This supports us in the benchmark-driven approach to memory modeling we have because this behavior could not be predicted knowing the cache coherence protocol used by the processor.

Contents

4.1 Scope and Model Overview **92**
4.2 Program and Memory Models **93**
4.2.1 Program Representation 93
4.2.2 Memory Model 95
4.2.3 Time Prediction 97
4.3 Experiments **99**
4.3.1 MKL dotproduct 100
4.3.2 MKL DAXPY 102
4.3.3 FFT Communication Pattern 103
4.3.4 Conjugate Gradient 104
4.4 Application to Shared Memory Communications **105**
4.4.1 Intra-node Communication Memory Model 106
4.4.2 Evaluation 108
4.4.3 Impact of Application Buffer Reuse 111
4.5 Conclusion **117**
4.5.1 Discussion 117
4.5.2 Related Work 118
4.5.3 Summary 119

Chapter 4

Benchmark based Performance Model

"The only source of knowledge is experience."
— Albert Einstein

High performance computing requires proper software tuning to better exploit the hardware abilities. The increasing complexity of the hardware leads to a growing need to understand and to model its behavior so as to optimize applications for performance. While the gap between memory and processor performance keeps growing, complex memory designs are introduced in order to hide the memory latency. Modern multi-core processors feature deep memory hierarchies with multiple levels of caches, with one or more levels shared between cores.

Performance models are essential because they provide feed-back to programmers and tools, and give a way to debug, understand and predict the behavior and performance of applications. Therefore we tried to automatically build full analytical models of hardware. While we were able to retrieve automatically information allowing – or easing – hardware modeling of the *on-core*, the complexity of modern memory hierarchies prevented us from doing the same with cache hierarchy. In this chapter we try to circumvent this issue by relying on benchmarks to provide performance information about the cache hierarchy modeled.

The benchmarks presented in the previous chapter is a strong basis to isolate memory performance and study it without noise from other software components. By analyzing the performance

achieved on several access pattern we can better understand and explain performance of memory hierarchies.

Some works focus on how to model cache misses [2, 6, 75, 77] for multi-threaded codes, or on how to model cache coherence traffic [51]. However, they do not consider simultaneously the impact of cache misses, of coherence and contention coming from shared caches. A 5C model, accounting for Compulsory, Capacity, Conflict misses, Contention and Coherence remains to be found in order to help study the impact of factors such as the data set size for each thread or the code scalability on multi-cores.

In this chapter we present a novel performance model that allows detailed performance prediction for memory-bound multi-threaded codes. This differs from the previous approaches by predicting the cumulated latency required to perform the data accesses of a multi-threaded code. The model resorts to micro-benchmarks in order to take into account the effects of the hierarchy of any number of caches, compulsory misses, capacity misses (to some extent), coherence traffic and contention. Micro-benchmarks offer the advantage of making the approach easy to calibrate on a new platform and able to take into account complex mechanisms. In addition these benchmarks can be used as hardware test-beds, *e.g.,* to choose between several architectures which one best suits the needs. Another usage of this test-bed is to check if a computer architecture performs as expected, or even detect some misbehavior as presented in Section 3.4.3.

This chapter is organized as follows: first, Section 4.1 presents the scope and an overview of our work. Our model is later detailed in Section 4.2 as a combination of hardware and software models. Section 4.3 details usage of our model to predict real world code performance. Finally we will show how we applied our model to model MPI communications in shared memory in Section 4.4.

4.1 Scope and Model Overview

Our model takes as input the source code to analyze. The code can be structured with any number of loops and statements, and we assume the data structures accessed are only scalars and arrays. Parallelism is assumed to be expressed in OpenMP parallel loops. The iteration count of the loops have to be known at the time of the analysis, and the array indexes have to be affine functions of surrounding loop counters and of thread ids.

Predicting the time to access memory usually requires to build a full theoretical performance model of the memory hierarchy. This work is however difficult due to the complexity of all the hardware mechanisms involved in modern architectures. Instead we choose to build a memory model that is calibrated thanks to benchmarks. These benchmarks are used to capture the hardware behavior of common memory access patterns. The benchmarks are used to recored the average latencies needed to access cache lines in on the different states of the `MESI` protocol. We then combine these latencies to predict memory access time of software.

The main difficulty with this approach is to find a set of benchmarks that characterizes hardware precisely, and to keep this set as small as possible. Indeed, the easiest way to build a model able to predict memory performance is to extract the memory access pattern of an application, and run a benchmark that has this same access pattern. However the combinatorics of such an approach

is way too large to be effectively implemented. Instead we choose a restricted set of benchmarks that provides us with information about read and write latency of the targeted architecture under common circumstances. Then we try to rebuild the application memory access pattern by combining the outputs of these basic benchmarks. We found that in cache coherent architectures, the state of the target cache lines has a large impact on performance. Concurrent accesses to shared data buffers lead to cache-line bounces between cores due to the need to maintain coherence between the existing data copies in their caches. Thus we build a set of benchmarks that gives us insight about the read and write latency to cache lines for every of the state of the `MESI` protocol [64]. Indeed most cache coherent processors use protocols that are based upon `MESI`. Since we aim at building a model that can be applied to a wide range of architectures, we do not want to embed in the model some states that are specific to a particular protocol or architecture.

In order to predict the time needed to access memory on a given application, we decompose it into a *memory access pattern*. This pattern tells us the amount of memory access and how it is accessed (*i.e.* reading or writing). If we suppose that we know the full state of memory (*i.e.* the cached addresses and their locations), we can *i)* reconstruct the state of memory after every access, and *ii)* read each access duration from the output of our benchmarks. By taking memory events one after another we can track the state of data in caches and construct a formula that will predict the time needed to access memory for the whole application. Figure 4.1 illustrates how software and hardware are modeled and how these models are combined to perform time prediction.

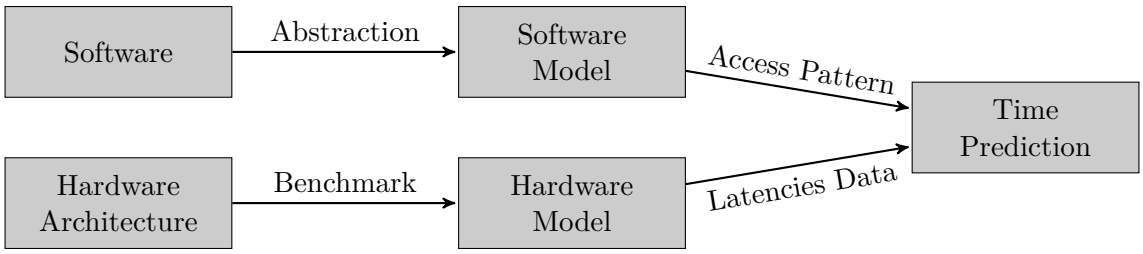


Figure 4.1: Illustration of the interaction of the different components of our Memory Model.

4.2 Program and Memory Models

In Section 4.2.1 we present how programs are represented in order to be able to apply the prediction. Then we detail the view of memory used in our work in Section 4.2.2. Finally Section 4.2.3 describes how the model is used for predicting execution times.

4.2.1 Program Representation

In order to use our memory model on real application we had to find a representation of software that reflects its interaction with memory. Since we aim at modeling memory performance we do not need to represent computation performed by threads but only the memory accessed. For each thread we model the chunks of memory accessed and how it is accessed.

Input codes are OpenMP parallel codes, using only parallel for loop constructs. Our model represents programs by only considering their memory access patterns since our work focuses on memory-bound codes. A memory access pattern is characterized by the addresses it accesses: *i.e.* the chunk and the access type, *i.e.* read or write. A memory chunk is defined by its size and stride. OpenMP codes are translated into a simplified representation, where statements are memory access steps, each step accessing a chunk of memory. These steps are characterized by the number of threads involved and, for each thread, the data read and written.

In this model, we define a chunk as the set of elements of a given array accessed by a thread in an OpenMP parallel for loop. These elements are considered as an atomic piece of memory, and a thread accesses either all the elements of a chunk or none of them. Moreover the access type has to be the same over all the elements of the chunk (only read or only write). Formally, we define a chunk as an array index region, defined by its size and stride, and defined by the triplet notation as used in compilers (interval and stride). For a given array, we assume the same chunks are used for all the analyzed code.

Memory access steps are defined as read and write statements to arrays, with a mapping associating threads from the set of threads $\mathcal{T}$ to chunks from the set $\mathcal{C}$. For instance, `read`(f, X) defines a read access to chunks of array X: For any thread $t \in \mathcal{T}$ for which $f(t)$ is defined, the thread t reads the chunk $X[f(t)]$. Since the same chunks are used for all accesses to a given array within a step, this means we assume the same mapping function f is used for all accesses to X. The following DAXPY computation with n threads illustrates this definition. This code

```
double X[SIZE],Y[SIZE];
#pragma omp parallel
for (int i=0; i<SIZE; i++) {
  Y[i] = a * X[i] + Y[i];
}
```

Listing 4.1: DAXPY kernel in C + OpenMP language.

would be represented as:

```
double X[SIZE],Y[SIZE];
read(f, X);
read(f, Y);
write(f, Y);
```

Listing 4.2: Representation of the DAXPY kernel with our formalism.

where

$$\begin{aligned}
\mathcal{T} &= \{0, 1, ...n-1\} \\
\mathcal{C} &= \{f(0), f(1), \ldots, f(n-1)\} \\
f: &\ \mathcal{T} \longrightarrow \mathcal{C} \\
& f(i) = \left[i * \frac{SIZE}{n}; (i+1) * \frac{SIZE}{n}; 8\right]
\end{aligned}$$

(assuming out of simplicity that $SIZE$ is a multiple of n).

The function f maps threads to triplets, where a triplet is defined by an interval of values and a stride (here 8, corresponding to the size of a `double`). Note that all read statements from the DAXPY code are replaced by two read statements, each accessing a section of the array. Therefore the sequence of reads and writes, where initially one read alternates with one write, has been replaced by a stream of read followed by a stream of write accesses. The code in Listing 4.1 and Listing 4.2 are not semantically equivalent. However we focus here only on performance, and the goal of the new code is to approach the performance of the initial one while being simpler to model.

For pipeline architectures, this abstraction provides an upper bound on performance that can be obtained from the initial code. Similarly, for sequential loops, loop-carried dependencies are in general not considered in our representation. Consider the following code:

```
double X[SIZE],Y[SIZE],k;
for (int i=0; i<SIZE; i++) {
  X[i] = k;
  k = Y[i];
}
```

The first statement depends on the second statement of the previous iteration. However, since the architectures we consider are pipelined, a read and a write can be issued at the same iteration. This code will be represented as:

```
double X[SIZE],Y[SIZE],k;
write(f,X);
read(f,Y);
```

where f is defined by $f(0) = [0; SIZE; 8]$ if only thread 0 executes this code.

Only parallel OpenMP loops with a static scheduling strategy can be analyzed. Defining chunk sizes in OpenMP boils down to a tiling transformation, with the outer loop a parallel loop, and the inner loop a sequential loop iterating within the chunks. Hence parallel loops with constant chunks can be translated into our representation.

Memory accesses performed in a `MASTER` OpenMP construct correspond to read/write involving only thread 0, with a direct representation in our formalism. Other keywords are not handled so far.

4.2.2 Memory Model

Once an application is modeled, we need to be able to predict its memory behavior with respect to caches and to the coherence protocol. To do so, we need to keep track of the state (in the coherence protocol) and location of each chunk.

The memory hierarchy is entirely modeled as one level of coherent, private and infinite capacity caches, with one cache per core, as depicted in Figure 4.2. We define a latency function, giving the time to access a chunk of data as a function depending:

- On the size of the chunk and on its stride,

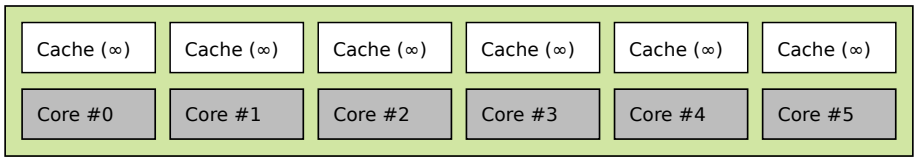


Figure 4.2: The memory hierarchy is modeled as one level coherent, private and infinite capacity caches.

- On the state of this data in the caches,
- On the number of threads that access data simultaneously.

We choose to build our model upon the `MESI` protocol since it is used as the basis for most CPUs with hardware cache coherence. Real hardware processors resort to variants of `MESI`, such as `MESIF` or `MOESI`, this protocols could be modeled likewise[1]. In the `MESI` protocol, data present in caches are in one of the four states defined in `MESI` protocol: *Modified*, *Exclusive*, *Shared*, *Invalid*. Since the programming model described in Section 4.2.1 uses data chunks as atomic blocks of data, the states used to maintain cache coherence are defined for chunks, not for individual data. We associate to each data chunk of the program a state corresponding to the `MESI` states, with the list of threads on the machine that have it in their (virtually infinite) own cache. These chunk states can be:

$M_{\{t\}}$ This data chunk is in state modified in the cache of the thread t.

$E_{\{t\}}$ This chunk is only in the cache of the tth thread (in exclusive state).

S_{Ω} The chunk is in shared state for all threads in Ω.

I This chunk is not present in any cache. At the beginning of the program, all chunks are in this state.

Therefore, for any array X and any mapping function f associated to X, the state of a chunk $X[f(t)]$, for any t, is updated according to the accesses to the accesses to it. This state will be denoted $X[f(t)].state$.

In terms of precision, this memory model takes into account some types of cache misses, whatever the number of caches in the memory hierarchy. **Compulsory misses** correspond to accesses to data in Invalid state (the I in `MESI` protocol). **Capacity misses** are not explicitly supported since the modeled caches have infinite capacity. However the latency function associated to read/write operations takes into account capacity misses by reporting the performance of different cache levels depending on the size of the chunk. Indeed large chunks result in data moving from L1 to higher levels of caches on the real platform. Therefore capacity misses occurring while accessing a single chunk are handled (while capacity misses due to the use of different chunks are ignored). **Conflict**

[1]The protocols could be *Forward* state identifies the only shared copy that is responsible for replying to requests from other caches. It reduces the traffic without changing the coherence model. Similar remarks applies for the *Owner* state.

misses are not handled. Note in general that our memory model is not able to count the number of misses, whatever their type, but their impact on the memory latency is handled. **Contention and the impact of coherence protocol** result from the state of data and of the number of threads accessing data simultaneously, and is also taken into account by the latency function.

The micro-benchmarks are defined and run in order to define this latency for all its possible values. The latency of a memory access depends on the chunk (its size and its stride), on the number of threads accessing to memory simultaneously, on the state of the chunk and the type of access (read or write). If considering a data chunk $X[f(t)]$ read by thread t with the statement `read`(f, X), the time to perform this read will be defined by the latency function $\mathbf{L}$ (for *load*):

$$\mathbf{L}(X[f(t)].state, |f|, f(t)) \tag{4.1}$$

where $|f|$ denotes the number of threads that read a chunk simultaneously, in the same state. A similar latency function $\mathbf{S}$ (for *store*) is defined for write operations.

The state of a chunk has to be updated after each access to this chunk. Consider a chunk read by threads with id in $\mathcal{T}$. If *state* is the state of this chunk before the read, the new state $state'$ is defined by a transition function δ as:

$$state' = \delta(state, \mathbf{L}, \mathcal{T}),$$

where $\mathbf{L}$ denotes a load operation. When a `read`(f, X) statement is executed, for each threads t involved in this step, the state of the chunk $X[f(t)].state$ is updated according to this transition. Similarly, when a chunk is written by a thread (only one thread at a time) during a `write`(f, X) statement, the type of operation is denoted $\mathbf{S}$.

Figure 4.3 defines how states change according to the type of access and the ids of the threads accessing the chunk. Each chunk maintains its own state and each state keeps track of the threads ids that have a valid copy of the chunk in their cache. The numbers associated to each transition correspond to the name of the benchmark (and latency function) used to predict the time to perform the access.

At the beginning of the program, each data chunk is in the initial state I, meaning it is not in any of the cache of our modeled machine. Executing read and write steps in our program model will change chunk states and the latencies for these transitions are defined by micro-benchmarks.

4.2.3 Time Prediction

To predict memory access time, we run the program description using `read` and `write` steps, update the states of the different chunks accessed and sum-up the latencies for these transitions. The associated time for each of these transition is read from the benchmark measures presented in Chapter 3.

Table 4.1 gives for each of the transition the corresponding benchmark. In the following we will call respectively lhm(n), lhe(n), lhs(n) the benchmark load hit modified, load hit exclusive and load hit shared with n threads in parallel. And lmm(n), lme(n), lms(n) and lmi(n) the benchmarks load

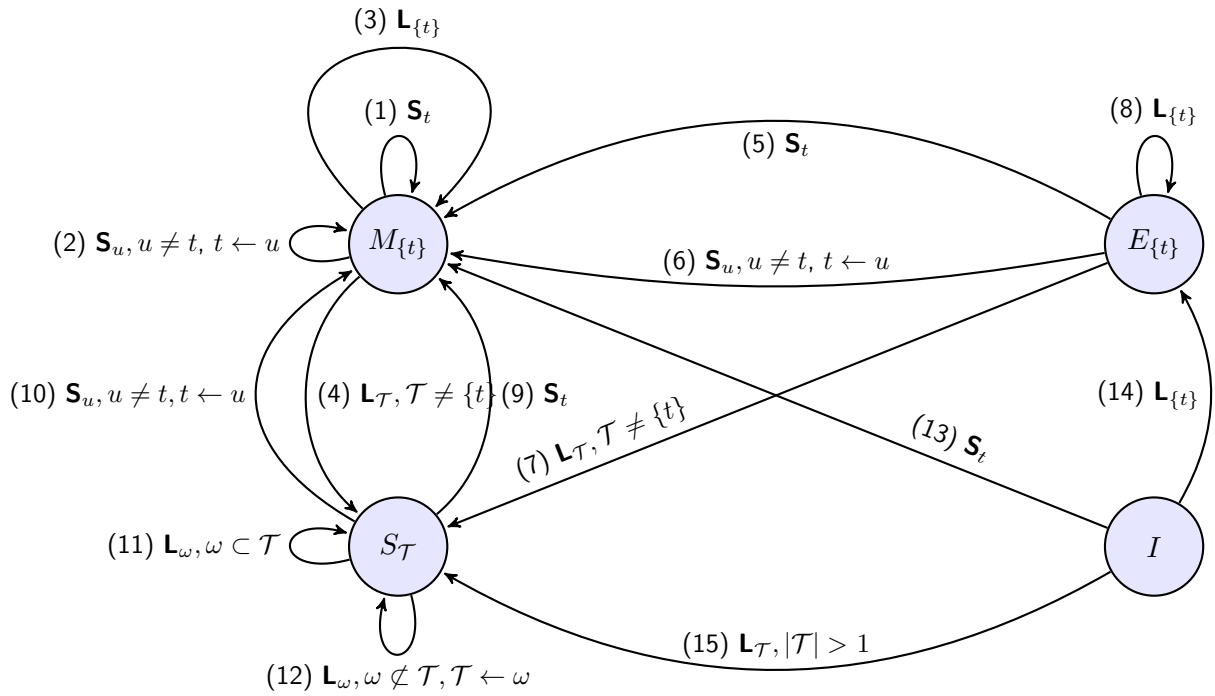


Figure 4.3: Automaton for tracking chunk's states and for selecting the benchmarks that represent each memory access step.

miss modified, exclusive, shared and invalid with n threads in parallel. Transition (1) is taken when a store ($\mathbf{S}_t$) happens on a modified chunk and if the chunk is already in the cache of the thread t performing the write: the benchmark to get the cost of this transition is therefore the *store hit modified* benchmark. If the chunk is not in the cache performing the store, transition (2) is taken and the corresponding benchmark is *store miss modified*. For each of these benchmarks, latencies are defined according to the number of threads executing a write, the size of the chunk and its stride, as shown in the definition 4.1. In transition (4), if $|\mathcal{T}| > 1$ and $t \in \mathcal{T}$, one thread will incur cache hit and the $|\mathcal{T}| - 1$ others will incur cache misses. We assume that the execution time of a program running n threads will be the execution time of the slowest thread. Thus, even if among n threads, one hits while the others miss, we assume the performance to be the one of the slowest threads, *i.e.* the performance of threads incurring cache misses.

In order to predict performance, our model must match reality. The benchmark part of the model is the first step toward this, but we also have to assemble benchmark results in a way that reflects real hardware behavior. For instance the fact that loads and stores can be performed at the same time on modern architectures has to be taken into account. When computing the total duration that predicts the run-time of a sequence of memory access steps, **the maximum of the aggregated latency of all read accesses, and of the aggregated latency of all write accesses** is therefore considered.

Besides, all chunks may not be in the same state for a read or write operation, and the memory access may not take the same time for all chunks. Indeed, some chunks are in different states

Table 4.1: Benchmark used for memory access time computation. Numbers in the left column correspond to the transitions in the state transition automaton in Figure 4.3.

Transition	Benchmark
(1)	shm
(2)	smm
(3)	lhm
(4)	lmm
(5)	she
(6)	sme
(7)	lme
(8)	lhe
(9)	shs
(10)	sms
(11)	lhs
(12)	lms
(13)	smi
(14)	lmi
(15)	lmi

within a single pattern, they will take different transitions. We only have parallel benchmarks where every chunk take the same transition (*e.g.,* only pure *load hit exclusive*). Fortunately, Zhang *et al.* showed that the sharing pattern of threads on multi-threaded application are often very regular: every chunk of memory will have the same behavior [89]. In order to ensure prediction even when different chunks are in different states, the latency for all chunk accesses is assumed to be the longest one among all accesses.

When aggregating the overall duration time of the pattern, each individual `read` or `write` steps is considered by computing the latency according to Algorithm 2 and updating the states of the involved chunks. The set of all triplets for f are denoted $f([0, \infty[)$, and enumerating these triplets lead to the enumeration of all the chunks accessed in the step. Note that **L** and **S** are the functions obtained from the micro-benchmarks.

4.3 Experiments

In this section we present several memory-bound applications that we modeled in order to predict their performance. This covers a wide range of HPC kernel types. Compute-bound applications are not considered because memory accesses are overlapped with computation and thus do not need to be optimized much.

One possible use of our model is to select the best working set size to achieve best performance. Another would be to select the minimal number of threads to use for a given computation without performance degradation. In order to illustrate these two approaches in the next sections, we will present comparison between our predictions and real applications. We only selected a few graphs

```
read(f, X) :
latency_L += max_{t∈0..|f|−1} L(X[f(t)].state, |f|, f(t)) ;
forall the c ∈ f([0, ∞[) do
|     X[c].state = δ(X[c].state, L, |f|) ;
end
write(f, X) :
latency_S += max_{t∈0..|f|−1} S(X[f(t)].state, |f|, f(t)) ;
forall the c ∈ f([0, ∞[) do
|     X[c].state = δ(X[c].state, S, |f|)
end
```

Algorithm 2: Definition of read and write steps. f is the function mapping threads to triplets, $|f|$ corresponds to the number of threads active in this step and X is an array.

in order to show interesting or unexpected results. After modeling several applications, we found that our model predicts performance of applications with an average fitness higher than 80%.

Our model computes the execution time of application. However in the remaining of this section we present the results in bandwidth because it fits better in plots and allows better performance comparison for several data set sizes.

4.3.1 MKL dotproduct

First we tried to predict performance of several BLAS subroutines. As our model predicts an upper bound of achievable performance, we choose to compare our prediction to one of the best available implementations, the MKL library.

The dotproduct computation can be modeled as shown on the following code. The dotproduct computation only consists of the load of 2 different chunks.

```
double X[SIZE],Y[SIZE];
read(f, X);
read(f, Y);
```

Listing 4.3: Representation of the dotproduct kernel with our formalism.

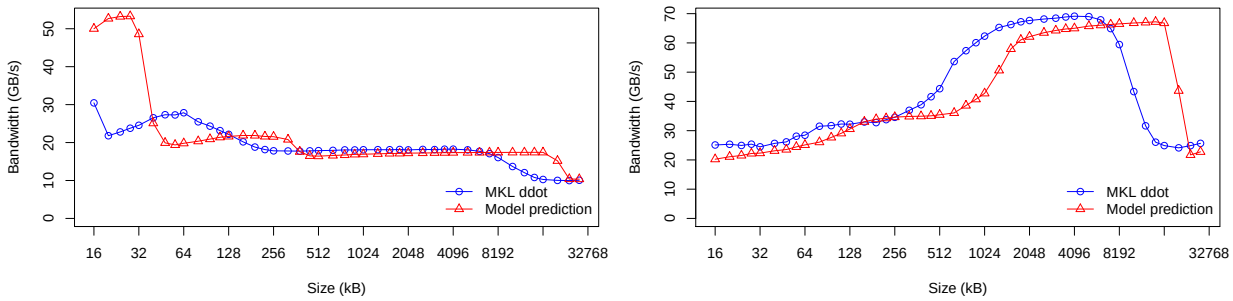
Before running the experiment, all chunks are written by the first thread (thread 0), therefore all chunks are in state $M_{\{0\}}$. We choose to initialize vectors this way because it shows first, that the initialization phase can be critical for further performance, second because that is what many users would do in the first place.

Let *size* be the size of each chunk, and n be the number of threads involved in the computation. The latency prediction formula for dotproduct code is:

$$T_{dotproduct} = lmm(n)(size) + lmm(n)(size)$$

Since threads all perform the same memory operation on chunks in the same state the execution time of each of them is the same and we simplified the MAX from the formula given the model.

$lmm(n)(size)$ is the time needed for n thread to perform a *load miss modified* on chunks of $size$ in parallel. Also since there is no store involved in this pattern, we also simplified the MAX between time to read and time to write.



(a) With 1 thread.

(b) With 4 threads.

Figure 4.4: Dotproduct pattern performance prediction compared to MKL dotproduct on Intel Sandy Bridge depending on the number of threads.

We can see on Figure 4.4 that our model is able to predict the behavior of dotproduct function calls from the MKL library. The model predicts performance changes near cache sizes (32 kB, 256 kB and 20 MB) while the experiment shows that the thresholds are actually twice lower. This may be caused by our model not taking capacity-misses explicitly into account as explained in Section 4.2.2. Aside from this shift, the graphs have very similar shapes.

The figure also shows that the size of data sets has a big influence on performance. When only using one thread, the dotproduct computation is faster when data set fits in the L1 cache. However it is faster in L3 when using several threads. Again, this comes from coherence overhead: with a single thread, there is no coherence to maintain, thus we get better performance with faster memory. However with more threads, coherence gets involved and performance is degraded. But when data only fits in the last level of cache, which is shared between all cores, then no cache coherence is required anymore, and code achieves better performance.

However, if we are careful with vector initialization (*i.e.* computing threads initialize the chunks they read), the dotproduct kernel can exhibit super-linear speedup as show on Figure 4.5 This super-linear speedup is due to the size of the chunks accessed by threads. With a single thread, the chunks accessed are 512 kB wide (2 chunks of 512 kB) and it only fits is L3 cache. However with more threads the chunk size becomes smaller and they fit in lower cache levels, leading to better performance.

Yet, with very large chunks, the dotproduct kernel can have a poor speedup even with a careful data initialization as shown on Figure 4.6. It increases up-to 4 with 6 threads and then stagnates. The predicted speed-up is close to the measured one. As seen in Section 3.4.3, parallel capacity misses appear when all threads use the same shared cache, leading to poor scalability. Even with 8 threads running this kernel, each of them still manipulates 8 MB which does not fit in local caches. This is an example showing that our model handles parallel capacity misses: when multiple threads competes for the memory blocks of a shared cache, our performance model is able to take into account the capacity misses that occur. Moreover, while this is not observed here, memory contention could be predicted likewise.

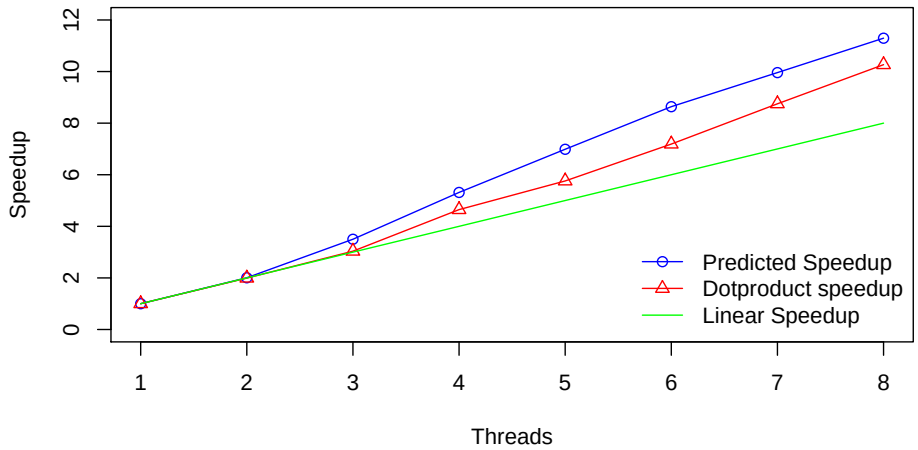


Figure 4.5: Dotproduct kernel speedup with 1 MB data set on Intel Sandy Bridge processor.

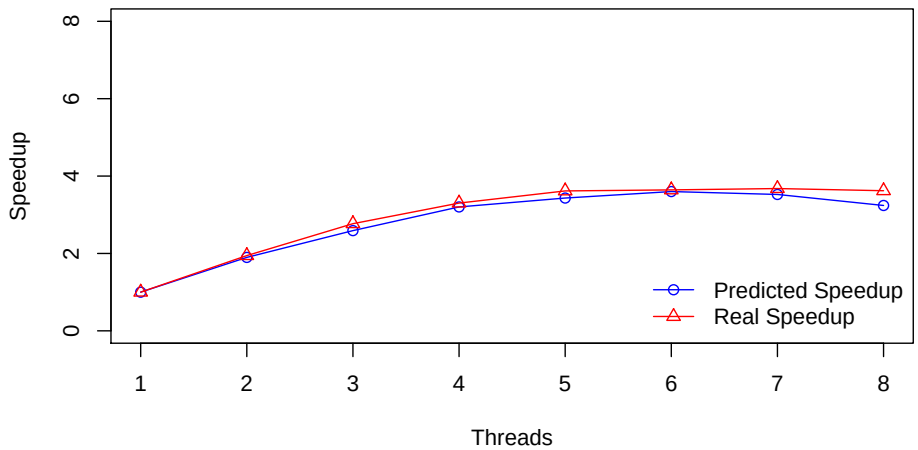


Figure 4.6: Dotproduct strong scalability on two 32 MB vectors.

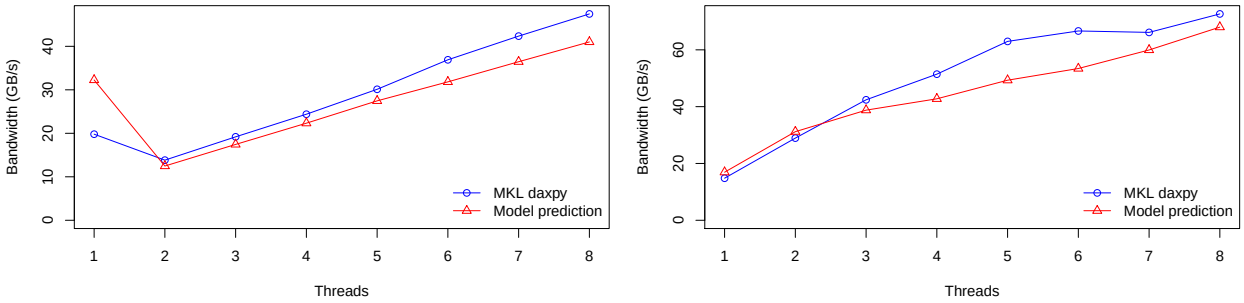
4.3.2 MKL DAXPY

In Section 4.2.1, Listing 4.2 shows the representation of a DAXPY computation. The time formula for DAXPY code is:

$$T_{DAXPY} = MAX\Big(lmm(n)(size) + lmm(n)(size),$$
$$shs(n(size)\Big)$$

Figure 4.7 shows the performance of the DAXPY operation depending on the number of threads. Again our model is able to predict the behavior of the MKL.

The figure also shows interesting facts about scalability and coherence overhead. First, for small data sets (32 kB) DAXPY run with 2 threads is slower than with a single thread. This comes from the fact that vectors are initialized out of any parallel section, therefore only the first thread touches



(a) Vector size: 32KB. (b) Vector size: 1 MB.

Figure 4.7: DAXPY-pattern strong scalability prediction compared to MKL DAXPY scalability on Intel Sandy Bridge.

them before calling the DAXPY function. Thus, with one thread, there is no overhead due to cache coherence. However with more threads, we have to pay the price of cache coherence. In order to optimize this application, one should be careful when initializing vectors. This also explains why, even for larger data sets, this code scales poorly: coherence overhead reduces effective memory bandwidth.

4.3.3 FFT Communication Pattern

Next, we tried to model more complex memory patterns. A part of the FFT computation is memory-bound, it is known as the *butterfly* pattern. The code performing a single butterfly on an array of size $2N$ is shown in Listing 4.4.

```
#pragma omp parallel for private(i)
for(i=0; i<N; i++) {
  j = A[i]
  k = A[i+N]
  B[i]   = p(j, k);
  B[i+N] = q(j, k);
}
```

Listing 4.4: The FFT *twiddle* or *butterfly* communication pattern written in C + OpenMP.

This communication pattern consists of reading 2 chunks performing computation on it and storing the result in two other chunks. We can see on the following code how it is modeled for our prediction:

```
read(f0, A);
read(fN, A);
write(f0, B);
write(fN, B);
```

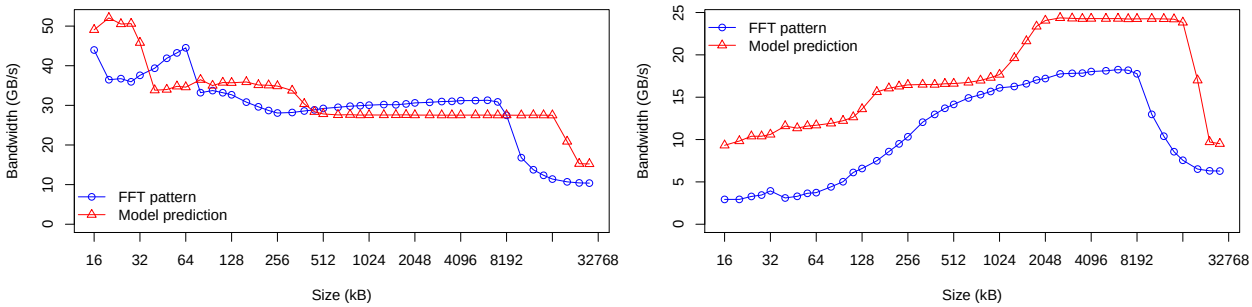
Listing 4.5: The same FFT communication pattern written with our program representation.

where `f0` is the function mapping thread 0 to the first chunk of A, thread 1 to the second chunk of A, *etc*, and `fN` the function mapping thread 0 to the $(N+1)^{th}$ chunk, thread 1 to the $(N+2)^{th}$

chunk, etc (using a modulo on the chunk number). Once again, the initialization consists in thread 0 writing all chunks. Thus chunks are all in the $M_{\{0\}}$ state before starting the computation.

$$T_{fft} = MAX\Big(lmm(n)\left(\frac{size}{2}\right) + lmm(n)\left(\frac{size}{2}\right), \\ smm(n)\left(\frac{size}{2}\right) + smm(n)\left(\frac{size}{2}\right)\Big)$$

We compare the predicted bandwidth and the real code bandwidth on Figure 4.8, both given per thread. The prediction curve is again a bit shifted on the right, but we still are able to predict the overall behavior of this access pattern. We can see that the FFT communication also presents a big cache coherence overhead: we get better performance when data sets fit the shared cache if using several threads while it achieves better performance in L1 cache with only 1 thread.



(a) With 1 thread.

(b) With 4 threads.

Figure 4.8: FFT pattern performance prediction on Intel Sandy Bridge depending on number of threads.

4.3.4 Conjugate Gradient

we applied our model to the Conjugate Gradient (*CG*) benchmark of the NAS parallel benchmark [7]. We were able to predict the speedup of this benchmark for all data set sizes (called *class* in the NPB configuration) as shown on Figure 4.9. The speedup of this benchmark is close to be ideal. This is due to the small amount of data sharing among threads. The CG benchmark is composed of 8 steps some of these steps are represented in the following codes 4.6 and 4.7.

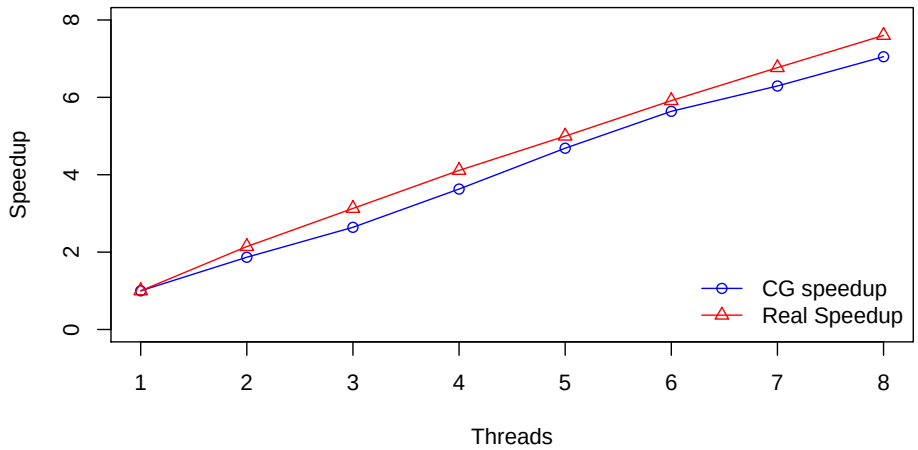


Figure 4.9: Speedup of the Conjugate Gradient (CG, class C) NAS benchmarks on an Intel Sandy Bridge architecture, with respect to the number of threads used. The gap between the predicted speed-up and the measured one is less than 12%.

```
while(..) {
  write(f, q);
  write(f, z);
  read(f, x) ;
  write(f, r);
  read(f, r) ;
  write(f, p);
  .. // Steps 2-8 of CG
}
```

Listing 4.6: Conjugate gradient (CG) step 1.

```
while(..) {
  .. // Steps 1-4
  read(f, z) ;
  read(f, p) ;
  write(f, z);
  read(f, r) ;
  read(f, q) ;
  write(f, r);
  read(f, r) ;
  .. // Steps 6-8
}
```

Listing 4.7: Conjugate gradient (CG) step 5.

Note that all mapping functions f are the same for all memory accesses. Steps 3 and 7 of the *CG* benchmark are matrix-vector products with variable length vector size, because this does not exactly fit our model we recorded the average length of the vector for each class and used it as a constant sized vector access. Still, our model was able to predict the real scalability of this algorithm.

4.4 Application to Shared Memory Communications

The performance of MPI communication in parallel scientific applications is often a key criteria for the overall software performance. Communication tuning has often been investigated for achieving better performance. Indeed most MPI implementations adapt their communication strategies to the underlying architecture and to the operation parameters. For instance processes running on the same node communicate through shared memory instead of through the network interface.

In order to help understanding and tuning of shared memory MPI communication we choose to analyze them through the use of the memory model presented in Section 4.2.

Communication inside nodes usually relies on two memory copies across a shared-memory buffer. These copies involve cache coherence mechanisms that have an important impact on the actual performance of memory transfers. Unfortunately MPI implementations tune shared memory communication strategies based on metrics that rarely take caches into account, merely by considering their sizes. Tuning of shared memory communication actually requires understanding the performance implications of cache coherence. Apprehending this impact can be cumbersome because modern memory architectures are increasingly complex, with multiple hierarchical levels of shared caches.

Proper automatic tuning of intra-node MPI communication strategy is very difficult because it depends on many factors: Is the transfer running alone on the machine or is it part of a large parallel communication scheme causing contention? Does the application want overlap? Does the hardware efficiently support these needs? Depending on the answers to these questions, the performance of a communication strategy may vary significantly.

We believe that cache coherence is the key to understanding these behaviors. Cache effects are often used as the easy cause of complex behaviors in memory-bound codes, especially shared-memory communication, without actually explaining them for real. Indeed the characteristics of caches (and of the cache coherence protocols that assembles them) is hidden in the hardware and rarely fully documented. Therefore cache coherence causes effects that cannot be easily modeled or even explained. Indeed we show later in this article that even modeling basic data transfers such as memory copies is difficult.

We want to tackle this problem with the same approach as presented in Section 4.2. We present in the next sections how we use it for analyzing and better understanding shared-memory-based intra-node MPI communication.

4.4.1 Intra-node Communication Memory Model

Shared-memory MPI communication uses an intermediate buffer that is shared between the sender and receiver processes. The sender process writes the message to the shared buffer before the receiver process reads it. As described on Figure 4.10, every byte in the transferred message therefore sees the following cache states:

1. The sender reads the data from its memory. Temporal locality implies that it may have been generated (written) recently. If so, this step is a *Load Hit* from a local Modified cache-line. If not available in the local caches anymore, this is a *Load Miss* that goes up to main memory.

2. The sender then writes the data to the shared buffer. That buffer was used by prior transfers. It is therefore usually available in the local cache as well as in the cache of another core. This is a *Store Hit* to a local Shared cache-line. The cache-line gets evicted from the remote caches and goes to the Modified state in the local caches.

Table 4.2: Memory access parallelism during a pipelined transfer when the message is divided into 3 chunks and the processor can execute one load and one store in parallel.

Time step	Sender Core	Receiver Core
1	Load + Store (chunk #1)	
2	Load + Store (chunk #2)	Load + Store (chunk #1)
3	Load + Store (chunk #3)	Load + Store (chunk #2)
4		Load + Store (chunk #3)

3. The receiver reads the shared-buffer from the sender core. This is a *Load Miss* from a remote Modified cache-line. The remote cache line gets copied in the local caches and both copies switch to the Shared state (this explain the state before step 2).

4. Finally the receiver writes the data to its receive buffer. If the target buffer was recently used, this is a *Store Hit* (usually to a local *Modified* cache-line). Otherwise it is a *Store Miss* to main memory.

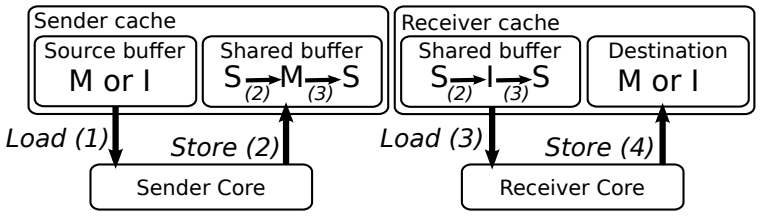


Figure 4.10: Cache state transitions for the source, and destination buffers of both sender and receiver cores during the memory accesses involved in a shared-memory-based data transfer.

Most modern MPI implementations follow this model. MPICH2 [15] and Open MPI [34] both allocate one large buffer shared between all local processes. It is then divided into one set of fixed-size buffers (chunks) per sender. It means that each process always reuses the same buffers for all transfers, even toward different destination processes. Other strategies exist for various kinds of communication (for instance dedicating one larger buffer to each directed connection, *etc*), but we will focus on this one when describing our model.

When the message is larger than fixed-size buffers, multiple ones are used and a pipeline protocol makes sure the receiver can read previous buffers while the sender fills the next ones. MPICH2 uses 64 kB *cells* while Open MPI uses 32 kB *fragments*[2] by default. As depicted in Table 4.2, this pipelined model means that there may be 4 concurrent memory accesses during a single transfer: Sender and receiver cores can execute their own copy in parallel. Each copy involves loads and stores that can be executed in parallel by modern cores. We will analyze the actual parallelism in Section 4.4.2. Each step translates into a benchmark output as listed in Table 4.3.

Given a message of size M and a maximal pipeline chunk of size C, there are $n = \lfloor M/C \rfloor$ chunks of size C_i (usually the first and/or last chunks are smaller than C if M is not an exact

[2] Open MPI uses a smaller first fragment so that the receiver can prepare the receiving of the next fragments before they actually arrive.

Table 4.3: Transitions involved in our model for each transfer step.

Step	Core	State transition
1	Sender	Load Hit Modified if recently generated, Load Miss Modified otherwise
2	Sender	Store Hit Shared
3	Receiver	Load Miss Modified
4	Receiver	Store Hit Modified if recently used, Store Miss Modified otherwise

multiple of C). The overall transfer time is estimated to

$$T = S(C_1) + \sum_{i=2}^{n} max\Big(S(C_i), R(C_{i-1})\Big) + R(C_n)$$

where S and R are the times to copy a chunk on the sender and receiver side respectively. When there is a single chunk, the sender and receiver times are added: the overall time is a sequential aggregation of both sides. When there are many chunks[3], the first and last terms can be neglected, and the overall duration is the maximum of the sender and receiver copy times.

Finally our model allows us to estimate S and R as we did for other kernels in Section 4.3. The representation of copies in our model is shown in Listing 4.8.

```
read(f, SRC)
write(f, DEST)
```

Listing 4.8: Copy representation within our model.

Since the copy is performed by a single thread, f maps the thread performing the copy to the whole buffer. For instance if the source and destination buffers have been used recently, S and R are estimated to

$$S(C_i) = MAX(lhm(2)(M), shs(2)(M)) \tag{4.2}$$

$$R(C_j) = MAX(lmm(2)(M), shm(2)(M)) \tag{4.3}$$

where $lhm(2)$, $shs(2)$, $lmm(2)$, and $shm(2)$ are the benchmark-measured time for a *Load Hit Modified*, *Store Hit Shared*, *Load Miss Modified* and *Store Hit Modified* respectively when two processes access memory at the same time.

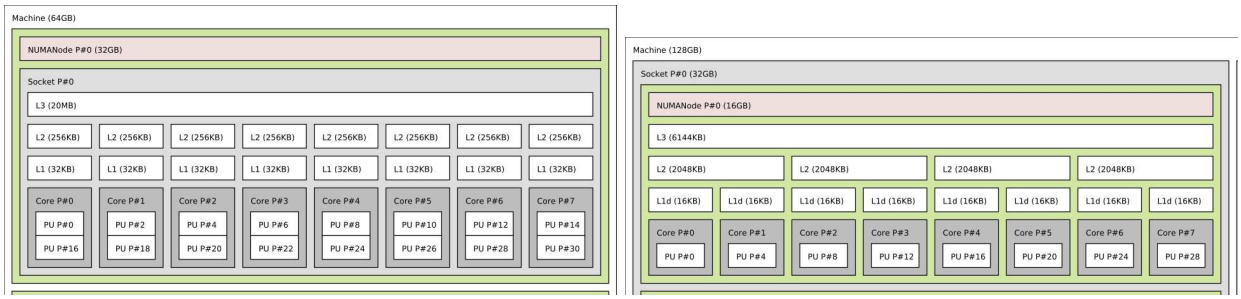
4.4.2 Evaluation

We now evaluate our model and use it to exhibit and analyze some possible optimization hints based on the impact of cache-coherence protocols on shared-memory MPI communication.

Our evaluation platform is summarized in Figure 4.11. It consists in two kinds of nodes. The first contains two 8-core 2 GHz Intel Xeon E5-2650 processors (Sandy-Bridge micro-architecture, 16

[3] A 1 MB message uses 32 chunks in Open MPI and 16 in MPICH2.

cores total, a single Hyper-Thread used per core). The second kind is made of four 16-core 2.1 GHz AMD Opteron 6272 processors (Bulldozer micro-architecture, 64 cores total). CPU frequency scaling as well as Intel Turbo Boost and AMD Turbo CORE technologies were disabled during tests so that the CPU and memory absolute performance does not vary.



(a) Intel Xeon *Sandy Bridge* E5-2650. (b) AMD Opteron *Bulldozer* 6272.

Figure 4.11: One socket of each kind of node in the evaluation platform.

To evaluate the model presented in Section 4.4.1 we compare its prediction with the performance of an experiment. However our model only predict the performance of the actual data transfer while MPI implementations add a lot of control code (such as eager message management, rendezvous messages, synchronization) around it. We therefore designed a synthetic experiment that only mimics the data transfer within the Open MPI 1.7 implementation (32 kB pipeline chunks). The performance behavior is similar, but the synthetic program gets higher performance thanks to the removal of the Open MPI control overhead.

Figure 4.12 presents the performance prediction of the model between 2 cores inside the same Intel socket. The top line is the *parallel prediction* which means both sender and receiver copies are executed fully in parallel. This is the asymptotic prediction for large messages. The bottom line is the *sequential prediction* which means copies are performed sequentially by the cores. This is the behavior for small messages when there is a single chunk. The different predictions plotted in this figure were computed using performance measurement of memory copies. The code of the experiment can be seen in Listing 4.9.

```
if (rank == 0) {
    MPI_Send(buf, SIZE, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
}
else {
    MPI_Recv(buf, SIZE, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
}
```

Listing 4.9: MPI transfer Experiment Code.

As explained in Section 4.4.1, the prediction model is a mix of these two cases transitioning from one to the other between 32 kB (single chunk) and 4 MB (128 chunks) message sizes. We observed that our model accurately predicts the performance except between 256 kB and 16 MB where the actual experiment is slower. These sizes corresponds to buffers that go into the L3 cache. We explain our misprediction by the fact that the L3 is shared between the two involved cores. It

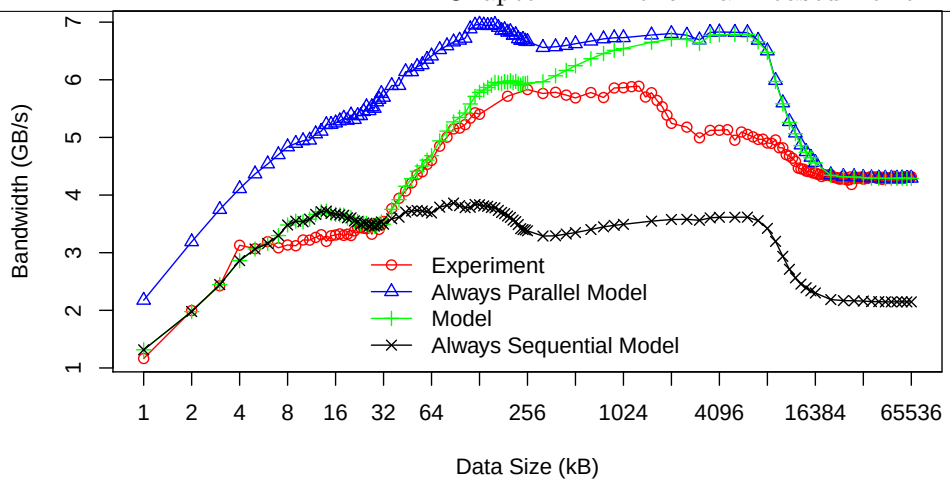


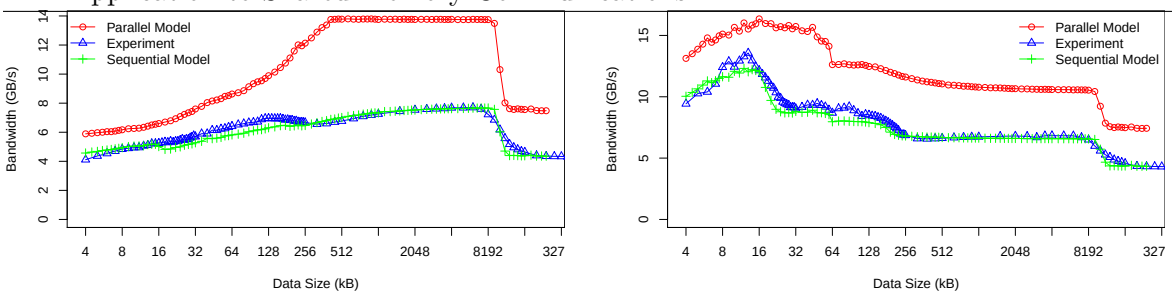
Figure 4.12: Comparison of the benchmark-based prediction model, the sequential model, the parallel model, and the actual shared-memory transfer. Intel platform.

causes contention and capacity misses that our benchmark-based memory model does not really take into account accurately. However, our model works well when the message fits in L1 and L2 cache and in main memory.

One thing that makes our model hard to apply is the difficulty to predict the performance of memory copies that are involved on both sides and accumulated in the analytic formula (S and R functions in Equations (4.2) and (4.3)). Figure 4.13 presents the prediction of each of the individual memory copies involved in the data transfer. It questions the presupposed ability of the processor to perform one load and one store in parallel as explained at the end of Section 4.4.1. Up to 128 kB messages (inside the L1 and L2 private caches), the observed throughput is the parallel bandwidth reduced by 20%. However, for larger messages, in L3 and in main memory, we only measure only 10% above the sequential throughput while the parallel one would be twice higher. Again, this is related to contention in the shared L3 cache and on the memory channels, which do not optimally support heavy parallel loads and stores. As we can see, the performance of each memory copy are very close to the sequential model, where the time prediction is the sum of the time to perform the load and the time to perform the store.

To summarize, our memory model can predict the performance of data transfer, assuming memory copy performance is understood, except when the shared L3 and main memory disturb parallel access performance. This shows why understanding shared-memory communication performance is always difficult: current memory architectures cannot be easily modeled, too many hidden hardware parameters are involved. Overall, we predict the performance behavior but not the absolute value very accurately. Fortunately, this is enough to analyze that behavior and discuss possible optimization hints in the next sections.

To increase precision of our prediction we chose to model Open MPI transfers performance with output of copy benchmark as a basic block. Yet we were able to model Open MPI shared memory communication and showed that benchmarking is an effective way to quickly understand complex mechanisms.



(a) Benchmark-based prediction of the receiver-side memory copy performance. The source buffer was recently written by another core (*Load Miss Modified*) while the destination buffer was recently used locally (*Store Hit Modified*).

(b) Benchmark-based prediction of the sender-side memory copy performance. The source buffer was recently written by the local core (*Load Hit Modified*) while the destination buffer was recently read by the receiver (*Store Hit Shared*).

Figure 4.13: Benchmark-based prediction of the sender-side memory and receiver-side memory copies performance. On an Intel platform: Sandy-Bridge micro-architecture.

4.4.3 Impact of Application Buffer Reuse

One common source of mis-understanding of shared-memory MPI communication performance is the reuse (or not) of application buffers in multiple iterations. As explained in Section 4.4.1, this changes the involved MESI cache states and causes individual memory access performance to vary significantly. It makes performance comparison meaningless when it is not clear whether the same buffers were reused multiple times. Some benchmarks [62] always reuse the same buffer while others such as IMB [43] have options to configure/avoid this reuse. We now look deeper at the actual impact of buffer reuse on the overall transfer time.

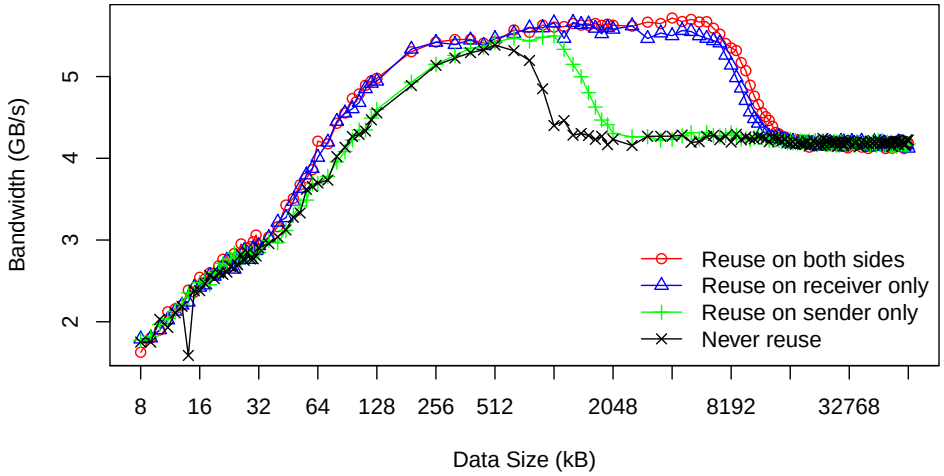


Figure 4.14: Impact of buffer reuse on IMB Pingpong throughput with Open MPI 1.7.3. IMB was modified to support buffer reuse on one side without the other. Intel platform.

Figure 4.14 compares the throughput depending on buffer reuse on both sides. We observe that the receiver buffer state is much more important than the sender. Unfortunately this result

is not convenient for application tuning because locality is easier to maintain on the send side: the application can usually send the data as soon as it is ready, while it often does not receive exactly when it needs it immediately. The receiver buffer state is more important than the sender because the receiver-side is slower. Therefore improving the sender locality to improve its transfer side will not significantly improve the overall transfer time. Indeed our micro-benchmarks reveal that the receiver side memory accesses (steps 3 and 4 on Table 4.3) hardly pass 15 GB/s for large messages, while the sender side (steps 1 and 2) often achieves close to 20 GB/s. This imbalance between send and receive side copy durations could be a reason to switch to variable-size pipeline chunks as previously proposed for InfiniBand communication [26]. Unfortunately existing MPI implementations require deep intrusive changes before we could experiment this idea.

Load Miss of Sender-written data

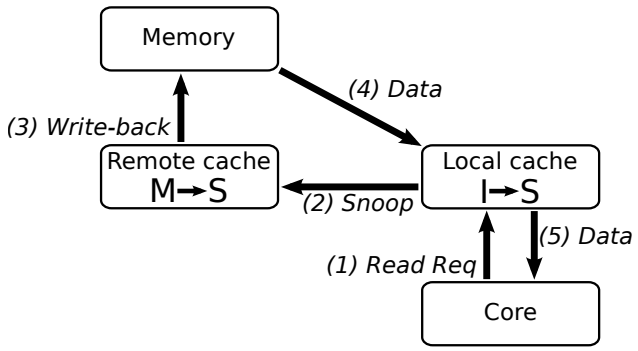


Figure 4.15: Anatomy of a *Load Miss Modified* (step 3) in the MESI protocol.

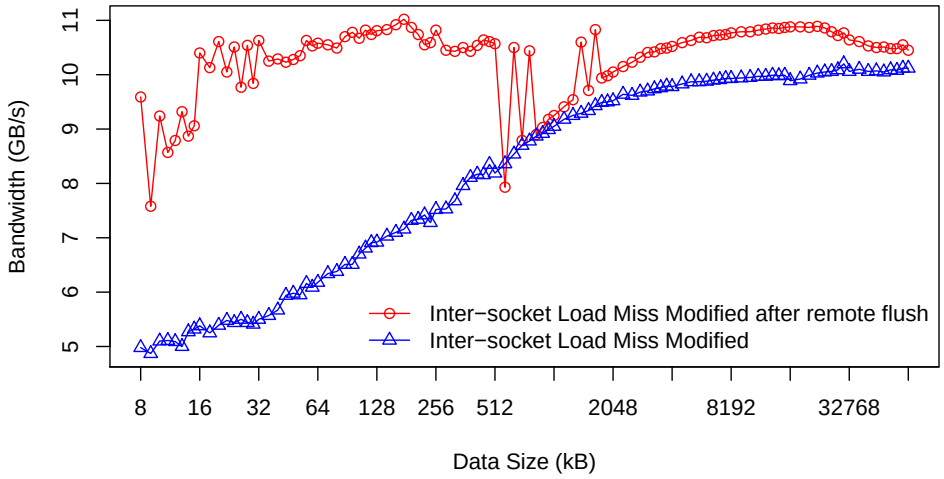


Figure 4.16: Impact of a flush of modified data on the performance of reading from another core, on the Intel platform.

We now focus on one of the transfer step that matters to the overall performance: when the receiver loads data that was previously written by another core (*Load Miss Modified*, step 3). The remotely-modified data have to be written back to memory before it can be shared by both cores (see Figure 4.15). If a cache is shared between the cores, the write-back is not actually required. If

no cache is shared, for instance when processes run on different sockets, the write-back is required, and Figure 4.16 confirms that it is expensive: An explicit flush of the remote copy increases the local *Load Miss Modified* by 10% to 100%.

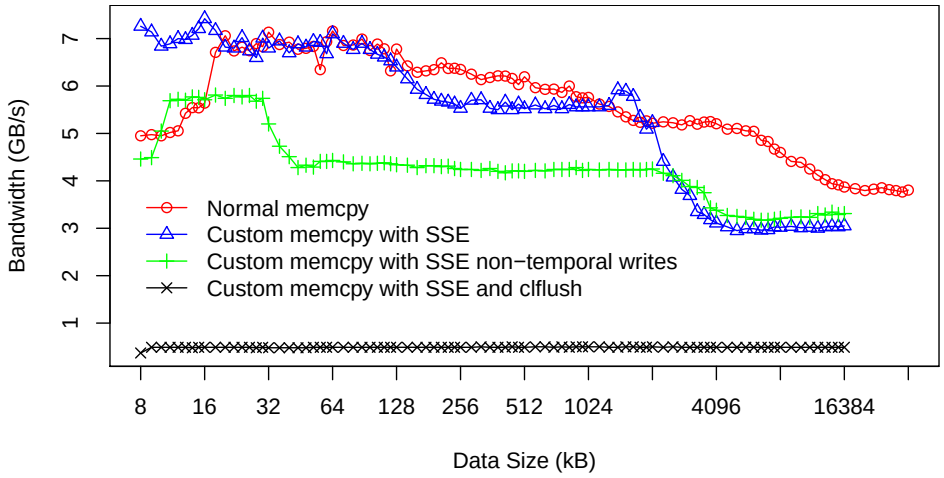


Figure 4.17: Impact of non-temporal stores and manually flushing on the performance of the sender write step 2, on the Intel platform.

One optimization would consist in moving this expensive remote write-back from the receiver load (step 3) back to the sender store (step 2), by anticipating it using one of the following ideas:
1) The sender could explicitly flush these cache-lines, e.g. with `clflush` x86 instructions. Unfortunately, this severely slows down the sender copy as depicted on Figure 4.17.
2) The sender could use a larger number of buffers so that the first buffers are automatically evicted when last ones are used. Unfortunately, current processors have very large caches that would require hundreds of buffers for this to work[4]
3) The sender could use non-temporal store instructions to directly reach main memory. This idea has often been considered in the past but very rarely used in production. Figure 4.17 shows that our custom copy with non-temporal writes is only about 30% slower than the usual copy, so the idea looks indeed interesting. Thus we modified Open MPI to perform a non-temporal store during step 2. However Figure 4.18 reveals that it actually divides the overall performance by a factor of 2. We could not explain this phenomenon. Unfortunately the behavior of non-temporal instructions with respect to cache-coherence protocol implementations is not widely documented.

Still, one has to keep in mind that moving the write-back to the sender-side may have the undesirable effect of moving the bottleneck from the receiver to the sender. It is therefore important to make sure that we do not slow the sender down too much. One idea to explore is to force the write-back only when the sender is waiting for the receiver to progress: Once the sender filled all shared-buffers, it may have to wait until the receiver gives some of them back, it may therefore start manually flushing with `clflush` in the meantime.

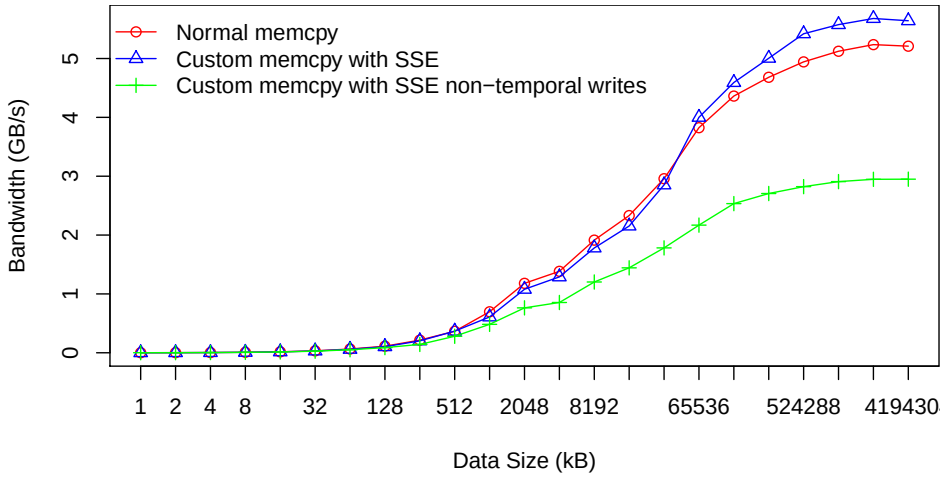


Figure 4.18: Impact of non-temporal stores in the sender write step 2 on the performance of IMB pingpong between 2 cores on different sockets, on the Intel platform, with a modified Open MPI 1.7.3.

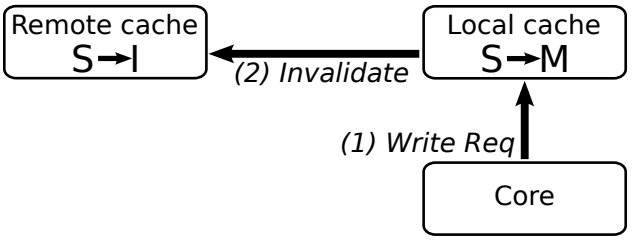


Figure 4.19: Anatomy of a *Store Hit Shared* (step 2) in the MESI protocol.

Rewrite of Receiver-read data

We now focus on the other critical transition, when the sender writes to a buffer that was previously used (*Store Hit Shared*). The remote copy has to be invalidated before the local copy can switch from Shared to Modified (see Figure 4.19). Fortunately some modern processors such as Intel Xeon E5 feature a directory in their L3 cache so as to filter such invalidation requests when it is known that there are no other copies. So a former remote flushing could reduce the overhead.

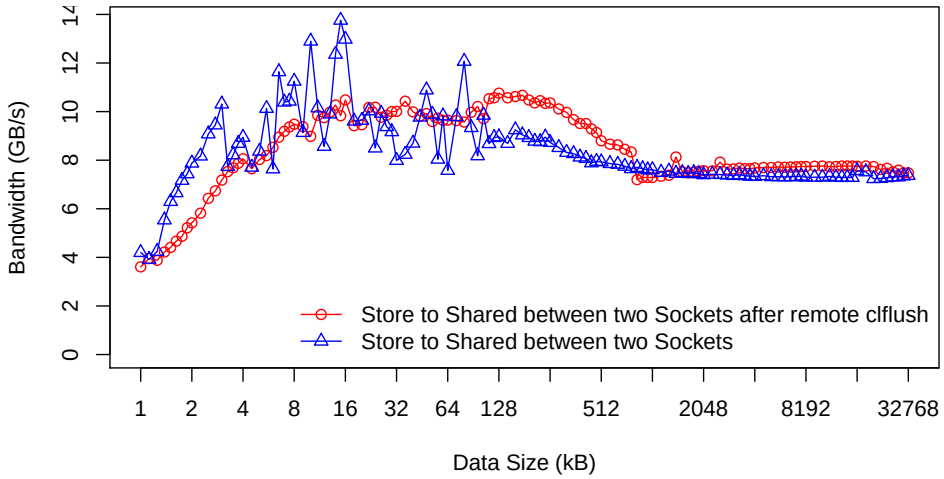


Figure 4.20: Impact of remote flushing on the performance of a local *Store Hit Shared* on the Intel platform.

Figure 4.20 shows that this idea could indeed improve performance by 5-10% for significant buffer size (larger than the 64 kB L1, we do not know why the graph is not smooth for smaller buffers). So one could think of adding some flushing on the receiver side. However, as discussed in the previous section, this would slow down the receiver bottleneck even more.

Another problem to consider here is that flushing instructions such as `clflush` may also flush lines out of other core caches that are below a higher-level inclusive shared cache, which would further degrade performance. For instance it would flush out all copies inside the entire Intel socket on our platform because the L3 is inclusive. On AMD, only the L2 is *mostly*-inclusive. This idea should therefore only be considered when the MPI implementation knows for sure that the involved cores do not shared an inclusive cache.

To summarize, optimizing the *Store Hit Shared* state (2) is hardly feasible in the context of the MESI protocol. However we have to revisit this result in next section due to certain characteristics of MESI variant implementations.

Shared-buffer Reuse Order and MOESI Protocol

AMD platforms use the MOESI protocol that was (notably) designed to ease sharing of modified data. This feature looks very interesting in our study because step 3 needs to read a remotely

[4] 640 and 192 buffers of 32 kB are needed on our Intel and AMD platforms respectively.

modified buffer. MOESI avoids the aforementioned write-back to memory by allowing immediate sharing of these dirty cache-lines with other cores. The original modified lines switch to the new *Owned* state (that is responsible for doing the write-back to memory eventually) while the shared copies go to *Shared* state. Unlike MESI where both sender and receiver copies are in the same Shared state after step 3, MOESI therefore introduces an asymmetry.

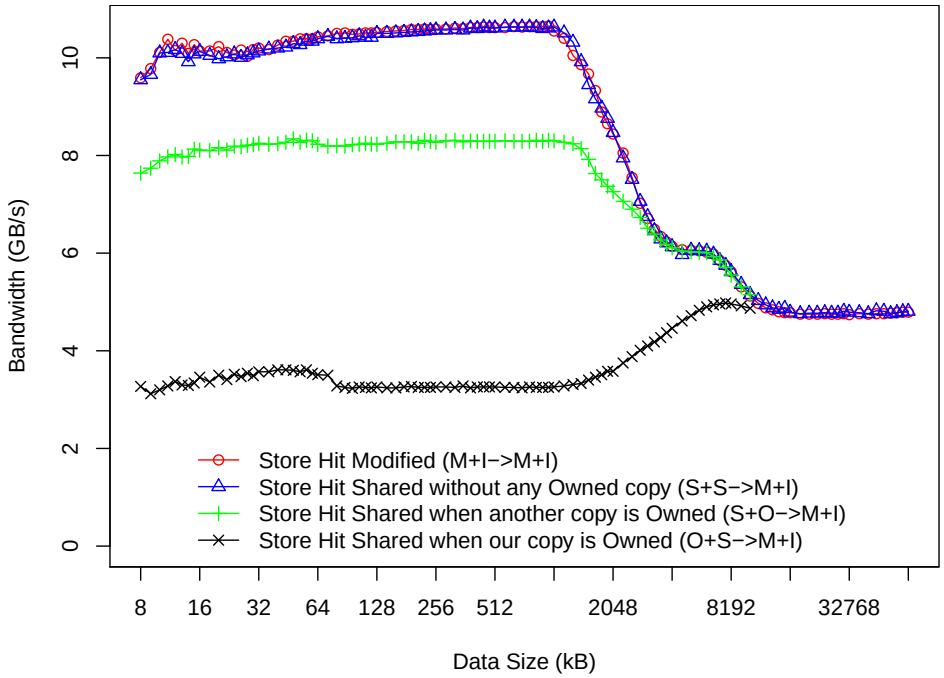


Figure 4.21: Store Hit performance depending on Shared, Owned and Modified state, inside a shared L3 cache, on AMD platform.

When a new transfer occurs through this shared-buffer, one of these asymmetric copies switches to Modified again during step 2 while the other gets invalidated. Given that the Modified state is similar to Owned (and not to Shared), one would expect that transitioning from Owned to Modified would be at least as quick as transitioning from Shared to Modified. Surprisingly Figure 4.21 shows the contrary: It is much faster (3x inside a socket, and 4x between sockets) to write to the Shared copy rather than the Owned one. We assume that a write-back always occurs when a cache-line leaves the Owned state and raises a non-documented phenomenon in this MOESI implementation.

This unexpected behavior leads to another unexpected result on Figure 4.22: On our AMD platform, data transfers are faster when shared-buffers are used in alternating direction (5 to 50% faster). This behavior seems very specific to AMD current micro-architecture *Bulldozer*. Intel nodes and some older AMD hosts (*Barcelona* micro-architecture) do not show such an asymmetric performance depending on buffer reuse direction[5].

This result confirms the interest of our idea of hiding hardware complexity inside micro-benchmark outputs: Extracting the performance behavior from this black-box is much easier on

[5] Intel nodes actually show a small performance difference as well, possibly because the MESIF protocol also breaks the symmetry between Shared copies (the *Forward* copy is the only one that replies to bus requests).

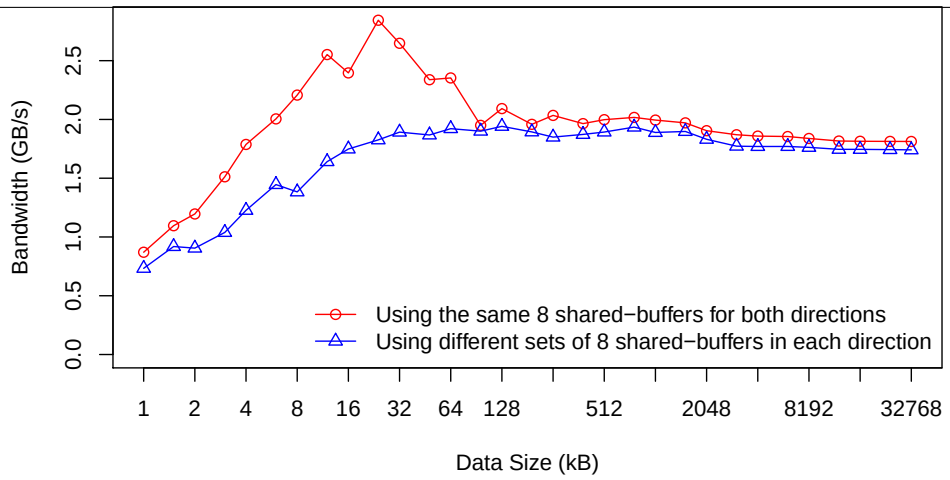


Figure 4.22: Performance of shared-memory data transfer depending on buffer reuse direction, inside a shared L3 cache, on AMD platform.

modern platform that trying to formally understand and model the hardware.

We showed that tuning MPI shared-memory communications can be eased thanks to cache coherence benchmarks. Indeed, the insight given by our benchmark set allows spotting bottlenecks in memory transfers. This helps exploring new strategies and quantifying the performance to expect .

4.5 Conclusion

Before concluding on the contribution we presented in this chapter we would like to bring the attention to a particularity of our approach. With our approach no hardware knowledge is needed to build the performance model but it is needed to build the benchmarks used by the model. This issue is discussed in Section 4.5.1, we will see the positioning of our model compared to related work in section 4.5.2, and conclude in Section 4.5.3.

4.5.1 Discussion

Our performance modeling aims at hiding hardware complexity. The key idea is to use benchmarks as a black box to characterize processors with no need to fully understand hardware. However in order to build a set of benchmarks able to reflect real hardware behavior, understanding processor bottleneck is crucial. Therefore it seems that we just shifted the problem of understanding hardware from the modeling part to the benchmarking part of our approach. We are now going to explain why this is not a weakness in our approach.

A good knowledge of the architecture is important to build the benchmarks used the describe the hardware peak performance and to have the intuition on how to combine them to produce a hardware model able to reflect real architecture behavior. However automatic methods could

be used to generate automatically micro-benchmarks. Tools allowing rapid exploration of a wide range of code versions, such as X-Language [27] or MicroPerf tools [11] could help writing the micro-benchmark. Therefore, writing the micro-benchmarks code is not a problem and could be done without a deep knowledge of the target Architecture.

However, getting the intuition on how to compose benchmarks to build a performance model is harder. Indeed human resources are involved. For the time being, there is no other alternative but to do it by ourselves. However the way to compose the benchmark black boxes seems to be the same with the same processor family. For instance we use the same model to predict performance of all codes running on Intel processor, no matter if it is a Core2, a Nehalem or a Sandy-bridge architecture. For this reason, using benchmarks as the basis for architecture modeling is still interesting because it factorizes the modeling effort.

4.5.2 Related Work

Several methods are commonly used to optimize software by observing and predicting performance. One is to simulate the full hardware, for instance with cycle accurate simulators [23, 41]. Such predictions are very precise and permit collection of large amount of performance metrics. However they are time consuming. Another problem is that it requires a deep knowledge of the hardware in order to implement all architecture features, including prefetchers or cache replacement policies, with enough precision to provide cycle accurate simulation. Developing such simulation software is a long process for each newly supported platform, and it highly depends on the hardly-available hardware documentation. Our approach hides this complexity in the benchmarks and tries to remain portable by using a memory model that matches most widely-available modern processors and coherence protocols.

One can also use profiling in order to record performance metrics aiming at tuning HPC applications. Profiling has the same drawbacks as simulation since it slows down application performance. Tools such as valgrind or cachegrind [61] can present an overhead up to 100 times the normal program execution time. Our approach relieves users from running the software. Indeed, given a representation of the software, we are able to model its behavior on a particular architecture and help application tuning.

Another method for tuning software is to use hardware counters. Tools have been developed in order to ease the use of hardware counters [47, 60]. The advantage of hardware counters compared to simulators is that it is lightweight: there is no overhead in such methods aside from the library initialization. However performance counters are not enough to optimize software. Indeed once a bottleneck of the application is found (let say too many TLB misses), one needs a way to link the information back to source code in order to tackle the problem. Also, as discussed in Chapter 3, Section 3.4, the overhead of misses significantly varies with cache states.

Daniel Hackenberg *et al* compared cache coherence of real world CPUs in [38]. They show that cache coherence and cache data states are to be taken into account when modeling memory hierarchy Williams *et al* did an ingenious work in modeling both memory and computation in order to predict best achievable performance of a given code depending on its arithmetic intensity [85]. Aleksandar Ilic *et al* extended the model to support caches and data reuse [42]. Compute-bound

applications are handled while we are not able to predict computation performance. However, our model is able to predict in a better way applications with heavy coherence traffic. This also allows us to point out that bad performance of some applications can come from a huge overhead due to cache coherence. The references confirm the relevance of our approach for modeling memory access performance.

Concerning tuning MPI implementations, many configuration options are available and some of them even target conflicting use cases with respect to point-to-point operations vs collectives, blocking vs non-blocking, caching for intra-node communication, *etc.* When predicting a good configuration is not feasible, auto-tuning may be used to adapt the software to specific application needs. The OPTO framework [19] tests all possible configuration combinations so as to automatically find the best one. Machine learning was also proposed as an alternative method [66]. A training tool finds out important characteristics of the platform before matching them with specific application needs.

The only work that is really close to our research mostly focuses on Xeon Phi accelerator cards [36]. However only synchronization issues (concurrent polling on shared receive queues) and small messages (up to 8 kB) are modeled. Our feeling is that modern memory architectures have a performance that is far too complex for such analytical models because of heavy and hardly-understandable behaviors when switching from L1 to L2, L3 or even main memory, or when looking at parallel accesses. This is why we hide this complexity inside micro-benchmark outputs.

Our approach is rather a qualitative approach that tries to understand cache-related issues instead of blindly finding the best tuning for specific applications. One common way to evaluate intra-node communication performance is to look at cache misses [65]. However we explained in Chapter 3 that this is hardly a reliable piece of information. In this chapter, we gave some basic optimization hints to application developers.

Our approach differs from existing ones as it is based on benchmarks to build the memory model. Benchmarks, especially the ones focusing on memory, have been developed in order to understand memory or application performance [45, 57, 81]. They are a great way to understand architecture behavior, however they can not be directly used to optimize software. As shown in Section 4.3, once our model is built for a given architecture, we are able to predict both software scalability and achievable memory bandwidth. By understanding the memory model or predicted scalability, one can see if performance is limited by memory contention or because of a cache coherence unfriendly memory access pattern.

4.5.3 Summary

As computer architecture and software become more and more complex, optimizing software to get the best performance out of a given machine gets more and more challenging. Code simulation and performance prediction become critical to performance analysis and software tuning. In Section 4.2, we presented an innovative model that predicts the performance of memory-bound applications by composing the output of micro-benchmarks based on the state of data buffers in hardware caches. In this model memory accesses are considered on chunks with the same access type (*e.g.,* only lead or store). These chunks are in a given state to represent the state of cache

lines in the coherence protocol. The caches are not modeled with their hardware feature but with a latency function that depends on chunks size and state.

We showed in Section 4.3 that the model successfully predicts the behavior of several commonly-used application patterns without the need to understand and describe all hidden complexity in the hardware mechanisms such as prefetchers and cache coherency protocol implementations. We were also able to demonstrate the efficient use of micro-benchmarks to understand performance of shared memory communication. As demonstrated in Section 4.4, our micro-benchmarks are able to produce results for inter-socket memory transfers that can be used to provide insight into shared-memory communication performance.

One of the weaknesses of our model is that we do not handle capacity misses explicitly. If an application loads a 32 kB buffer (that fits in L1) and then accesses a large amount of data, the first buffer will be evicted from the cache, and further accesses to this buffer will be slower than our prediction. This weakness can explain some of our mis-predictions, for instance the horizontal shifting of prediction graphs such as in Figure 4.8. We are working on extending the model in order to fix this problem: Modeling caches with a stack of chunks could help tracking record of the size of chunks accesses. When a thread accesses a chunk it checks its stack, if the chunk accessed is in the stack, instead of using a size of the chunk as parameter for the latency function, the sum of the size of all chunk between the top of the stack and the chunk is used. And if the chunk is not in the stack, it is push onto it. Therefore is thread 0 accesses chunk s_0, s_1, and s_0 again, sequentially (with an empty cache in the beginning), s_0 will be pushed, then s_1 will be pushed. Therefore, when accessing s_0 again, the size parameter for the average latency function will be the size of $s_0 + s_1$. And we have to be careful to remove chunks from the stack when an coherence message requires an eviction.

The model could also be enhanced for supporting some specific coherence issues such as I/O DMA transfers or non-temporal processor instructions that cause unexpected eviction of lines out of the caches.

Until now our model only predicts the performance of applications whose threads all run on the same socket. While benchmarks are already ready for multiple sockets, we need to plug them into the model. Another issue with multi-socket support is to avoid the combinatorial explosion that could appear when the number of cores in the hardware model increases and the topology is not flat anymore. Secondly, we are thinking of adding automatic ways to detect coherence issues and their impact on performance. This idea behind this is to run an application with hardware counter instrumentation to measure the number of different cache events. For instance PAPI [60] can record the number of requests for exclusive access to shared cache line. Our model could be used in this context to provide a metric allowing prediction of performance for the same application if we could discard these invalidation messages.

Conclusion

This dissertation fits into the High Performance Computing area. HPC is used in a growing number of scientific areas where simulation requires a large amount of computation. The most important point of HPC is definitively performance. Because of the limits of sequential computation, more and more features were embedded into modern processors to increase their computational power. This increasing complexity makes it increasingly harder to find the code version that will lead to optimal performance.

Hardware models are used to take optimization decisions. But the growing number of features in CPU chips and the frequent release of new hardware makes it tiresome to understand and model every newly released hardware. Also architecture documentation is not always available or complete. We present an innovative memory model based upon benchmarks. The iteration of benchmarks into a hardware model allows capturing architecture behavior and peak performance. This information can then be used to predict application behavior on the benchmarked architecture. Benchmarks can also be used to find undocumented hardware characteristics.

Still designing specific benchmarks, running them, understanding the results and their implications at the hardware level, and integrating them into hardware models is a long and difficult task. The main contribution of this thesis is to show how benchmarks can be directly integrated into performance models. With a small number of assumptions on the target architecture we are able to use benchmark results as a black box representing hardware performance. This abstraction of hardware performance allow us to combine benchmark output to predict software behavior on a particular architecture. The assumption we made about the hardware modeled are the usage of `MESI` based protocol to maintain cache coherence. This allows us to target a large range of `x86` processors.

Contributions of this Dissertation

This dissertation is organized around three main contributions that address these questions: *i)* how to automatically retrieve critical hardware parameters to build hardware models (Chapter 2), *ii)* how to build benchmarks that can capture hardware performance, especially the performance of the memory hierarchy levels (Chapter 3), *iii)* how to integrate benchmarks into a performance model that can predict complex application behavior (Chapter 4).

We showed that information about the hardware can be extracted with a careful benchmarking methodology. In particular the latency of instructions can be measured automatically to feed a

computational model – called on-core modeling in the dissertation. Yet instruction latency is not the only input parameter needed to build an on-core model of processors. We also need the number of execution ports available as well as the port used by instruction be be able to judge the quality of a code version. We saw that these information are harder to collect automatically but can be done with a proper methodology.

Cache coherence protocols were developed to keep memory coherent from the programmer point of view. These protocols are most of the time implemented in hardware and involve memory traffic that is not visible to the programmer and that can lead to significant overhead. Also the coherence protocol used in real hardware are not always extensively documented – especially the interaction between the different memory components (*e.g.,* how cache coherence between sockets is implemented). We presented a framework and more generally a methodology to benchmark cache coherent memory systems. From the results of the benchmark designed, we could extract guidelines to help better use of cache coherent architectures. We also were able to find some unexpected poor cache performance on the Dunnington architecture.

We also presented a memory model built upon benchmarks. This model allows us to predict the behavior of programs running on cache coherent multi-core systems. To be able to predict application behavior we had to combine several models together: A software model representing how the program uses memory. And a, hardware model that abstracts the view of the memory organization of multi-core systems. It helps tracking the state of memory chunks accessed by the program. We validated our memory model by being able to predict the scalability of several linear algebra codes as well as more complex applications such as a Conjugate Gradient computation. The limits of our model lie in it ability to reflect application memory behavior. The model presented in this dissertation only handles regular OpenMP code. This model could be used to handle more complex or irregular code with little effort. Indeed we can represent several threads performing different memory operation. The hard part is to get the performance of heterogeneous parallel memory access patterns. By heterogeneous parallel memory access pattern we mean where every thread to not perform the same memory operation. To get performance of such access, we need to run benchmarks with different threads performing different memory operation. The other solution would be to combine the homogeneous access pattern we already described to predict performance of more complex ones.

Perspectives

The trend toward more and more complex hardware design seems to be unavoidable. Software optimization will therefore still be a challenging research area for years. The more can be achieved automatically, the more developer productivity will be improved. While the contributions of this dissertation mainly focus on a restricted set of architectures, the ideas behind it can be applied to a wide range of hardware.

Integration of the Model into existing Tools A short term perspective would be to implement our cache coherence memory model into existing tools to automatically validate our model on a wide range of software. Several approaches can be considered: the first would be to implement a compiler pass to perform static OpenMP code analysis to build the software representation

needed by our model. This is the last step to have a fully automatic tool able to model software to help its optimization. The advantage of this approach is that it is based on static code analysis. Therefore it capture the general behavior of the application: it is does not depend on a single run – unlike trace based approaches. But the drawback is that only regular code with affine loops can be handled. Input dependent application could not be modeled with static code analysis.

Another approach would be to use memory traces and build the memory access pattern by replaying these traces. Some frameworks allow statistics collection during code execution, these data could also be used to construct the software representation needed in our model. The advantage of this approach is that it does not require any property on the application to be modeled. However traces are collected for one single run of the application. Therefore we cannot expect it to represent the application for every possible input and/or run. There is a trade-off to make between these two approaches: code with enough properties can be handled with static code analysis and provide a generic model, and code input dependent have to be handled through trace collection, which is a more specific approach.

The strength of our memory model over other existing ones is its ability to bring cache coherence into the heart of the memory model. For instance cache models based on counting cache event, such as hit and miss for every cache level, miss the accurate time cost of each of these events. Indeed in our model we are able to distinguish different misses events depending on their latency.

Improving the Model A finer grain memory model could also improve the accuracy of the time predicted by our model. Especially, designing a finer capacity model could help with prediction accuracy. Another approach to predict memory performance could be to implement a cache simulator (in tools such as cachegrind). Results from our benchmark could be used as a cost function to estimate the time needed for every cache hit or miss.

We saw that in order to extend the model to irregular applications (*i.e.* where computing processes do not perform the same operation) we need to be able to associate a time or cost function to irregular access memory access patterns. The solution to build benchmarks to measure performance of all memory access pattern is not sustainable. As the number of core per processors increases the combinatorics involved would become too important. The idea to circumvent this problem is to find a realistic way to combine regular parallel benchmarks to predict performance of irregular codes. The simplest idea would be to take the longest execution time of all threads involved in the heterogeneous access pattern. However contention depends on the state of cache line accessed and it is unclear if this approach could be realistic on many hardware architectures.

Software handled by our model follow the fork/join model. While most OpenMP application can be modeled this way, other parallel paradigm lead to other software models. For instance task parallelism without synchronization with all threads cannot be model with our current software model. This brings two issues: first our software model has to be restructured to handle task parallelism. This can be achieved with a task graph. But the second issue is much harder to tackle: how to retrieve hardware performance we can expect from a particular software pattern from a fixed-size benchmark set. Without synchronization we are not able to deduce precisely the work performed by every threads. Since performance of threads depends on the workload of others, we do not yet have an answer to the issue of handling general task parallelism.

Application to Large Scale platforms and Applications In this dissertation, we mainly focused on small and regular applications running inside the same node (*i.e.* computer). But most HPC applications are much more complex, feature complex interaction with hardware, and are distributed across nodes connected through networks. In order to bridge the gap between the range of application handled by our model and real HPC application we should be able to handle more complex kernels and extend the model to inter-node communications. For now our approach has difficulty to model irregular application or data dependent software. Modeling of such complex code can rely on simulation platforms.

Simulation of large scale systems is used to predict application behavior of future HPC platforms [17]. We are thinking about using our memory model to help simulation of large platforms in the SimGrid simulation platform in the context of the SONGS ANR project[6]. But for now the simulation is too high level for our low level model memory model to be integrated easily. SimGrid currently allows simulation of clusters composed of several mono-core nodes. We are studying how to integrate our model into this platform to extend it be able to simulate nodes featuring CMP nodes.

Future of HPC Platforms

One of the big challenges of computer science has always been to control complexity. This explains why code modularization and software development methodology (*e.g.,* Agile) were investigated. Hardware design follow the same trend as software: more and more features are added to the existing. The increasing complexity of hardware following prevents clear overview of processor performance and makes code optimization complex.

Another issue with such complexity embedding in processors is that – unlike software – hardware is fixed and therefore cannot adapt to application needs. Two approaches could tackle this issue: with FPGA technology we could program hardware to behave differently depending on software, or we could completely let software control the hardware. The most striking example about the increasing complexity of hardware is cache handling: cache architecture can feature very complex designs in order to adapt to several software behavior [46, 69].

Cache Coherence As cache coherence is harder and harder to maintain as the number of cores per node increases. Since it seems to be the trend for future computer architectures, cache coherence scalability issue is often discussed. Some think that hardware cache coherence can scale even with a large number of cores [54]. But since several architectures without hardware maintained cache coherence were released, the opportunity to study trade-offs of software versus hardware cache coherence is wide open. The choice to make between hardware or software can be a great opportunity for memory benchmarks. Indeed, benchmarks can be a great way to quickly prototype software cache coherence. This can also be used adapt the coherence protocol to the needs of a particular application.

The advantage of hardware managed caches is that it is automatic: no intervention from end user is needed to keep memory consistent. While keeping memory consistent on software managed

[6] http://infra-songs.gforge.inria.fr/

cache requires the addition of special instructions into software to maintain the coherence. However maintaining cache coherence by software can avoid unnecessary coherence traffic. For instance by avoiding invalidation of a cache lines that will not be read any more in the future.

Even hardware without caches but scratchpads exists: scratchpads can be seen as fully software managed cache, the Cell and Cyclops64 processors feature scratchpads memories. Scratchpads are private fast memory but data that has to be explicitly stored into it (with caches – event software managed ones – memory is automatically caches). With scratchpad based memory architectures, even more control is given to software. However the model we presented in this dissertation is by far too complex to handle scratchpad based architecture. Indeed on scratchpads, actions performed by hardware are explicit copies between local storage and shared memory. Therefore the benchmarks needed to characterize memory transfers is much more restricted. However we can imagine runtime systems or library performing automatic data movement to store critical data in faster memories. In this environment, using the exact same approach as we did can help modeling performance of transfers made by the runtime. We can expect the caching strategy of such system as well as the coherence protocol used to be similar to existing hardware cache. Therefore the model itself could be used out of the box. But the benchmark design would have to be re factored.

Letting the end user handle completely the memory hierarchy is not an alternative. Most of the time end users of HPC applications are not computer science experts and would not be able to write software using efficiently the underlying hardware. Even worse with software managed cache coherence systems, forgetting instructions to keep memory coherent between processes would lead to errors. But runtime systems or libraries can be used to automatically perform cache coherence or to automatically bring reused data to faster memories.

The advantage managing faster memories by software rather than by hardware is that software can perform finer optimization than hardware since it is aware of data that will benefits from being caches or not. This would avoid cache trashing and avoidable cache coherence traffic.

Energy Memory will not be the only issue with future large scale systems. Power consumption is now a major constraint of next generation platforms [12]. The out-of-order front-end of the instruction pipeline is responsible for a significant fraction of power consumption [16, 32]. Architectures targeting HPC applications were released with an in-order pipeline in order to reduce the energy consumption of such chips. The Xeon Phi released by Intel is an example. Since optimal instruction scheduling can be found by static code analysis by the compiler [10, 53, 84], we can imagine that this hardware feature will tent to disappear from processor dedicated to HPC.

The tend of hardware to become more and more complex seems carry one, HPC software will therefore keep challenging software engineers and researchers for the years to come in order to achieve fast computation. We believe that no matter how hardware and software systems dedicated to HPC will evolve benchmarks will remain a relevant source of knowledge.

Bibliography

[1] F. Agakov et al. "Using machine learning to focus iterative optimization". In: *In Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 2006, pp. 295–305 (cit. on p. 14).

[2] Agarwal, A. and Hennessy, J. and Horowitz, M. "An analytical cache model". In: *ACM Trans. Comput. Syst.* 7.2 (May 1989), pp. 184–215 (cit. on p. 92).

[3] Anant Agarwal et al. "An Evaluation of Directory Schemes for Cache Coherence". In: *In Proceedings of the 15th Annual International Symposium on Computer Architecture*. 1988, pp. 280–289 (cit. on p. 30).

[4] M. Ajmone Marsan et al. "Modeling Bus Contention and Memory Interference in a Multiprocessor System". In: *Computers, IEEE Transactions on* C-32.1 (1983), pp. 60–72 (cit. on p. 59).

[5] Björn Andersson, Arvind Easwaran, and Jinkyu Lee. "Finding an upper bound on the increase in execution time due to contention on the memory bus in COTS-based multicore systems". In: *SIGBED Rev.* 7.1 (Jan. 2010), 4:1–4:4 (cit. on p. 59).

[6] Diego Andrade, Basilio B. Fraguela, and Ramon Doallo. "Accurate prediction of the behavior of multithreaded applications in shared caches". In: *Parallel Computing* 39.1 (2013), pp. 36–57 (cit. on p. 92).

[7] D. H. Bailey et al. *The NAS Parallel Benchmarks*. Tech. rep. The Intl Journal of Supercomputer Applications, 1991 (cit. on p. 104).

[8] Denis Barthou et al. "Loop Optimization using Adaptive Compilation and Kernel Decomposition". In: *ACM/IEEE Intl. Symp. on Code Optimization and Generation*. San Jose, California: IEEE Computer Society, Mar. 2007, pp. 170–184 (cit. on p. 56).

[9] Denis Barthou et al. "Performance Tuning of x86 OpenMP Codes with MAQAO". In: *Tools for High Performance Computing 2009*. Ed. by Matthias S. Müller et al. Springer Berlin Heidelberg, 2010, pp. 95–113 (cit. on pp. 37, 57).

[10] Peter van Beek and Kent Wilken. *Fast Optimal Instruction Scheduling for Single-issue Processors with Arbitrary Latencies*. 2001 (cit. on pp. 14, 125).

[11] J.C. Beyler et al. "MicroTools: Automating Program Generation and Performance Measurement". In: *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. 2012, pp. 424–433 (cit. on pp. 37, 68, 89, 118).

[12] S. Borkar. "The Exascale challenge". In: *VLSI Design Automation and Test (VLSI-DAT), 2010 International Symposium on*. Apr. 2010, pp. 2–3 (cit. on p. 125).

[13] François Broquedis et al. "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications". In: *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*. Pisa, Italia: IEEE Computer Society Press, Feb. 2010, pp. 180–186 (cit. on p. 58).

[14] James R. Bulpin and Ian A. Pratt. "Hyper-threading aware process scheduling heuristics". In: *Proceedings of the annual conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, pp. 27–27 (cit. on p. 18).

[15] Darius Buntinas, Guillaume Mercier, and William Gropp. "Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface: Proc. 13th European PVM/MPI Users Group Meeting*. Bonn, Germany, Sept. 2006 (cit. on p. 107).

[16] A. Buyuktosunoglu et al. "Energy efficient co-adaptive instruction fetch and issue". In: *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. 2003, pp. 147–156 (cit. on p. 125).

[17] Henri Casanova, Arnaud Legrand, and Martin Quinson. "SimGrid: a Generic Framework for Large-Scale Distributed Experiments". In: *Proceedings of the Tenth International Conference on Computer Modeling and Simulation*. UKSIM '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 126–131 (cit. on p. 124).

[18] J. Cavazos et al. "Rapidly Selecting Good Compiler Optimizations using Performance Counters". In: *Code Generation and Optimization, 2007. CGO '07. International Symposium on*. 2007, pp. 185–197 (cit. on p. 14).

[19] Mohamad Chaarawi et al. "A Tool for Optimizing Runtime Parameters of Open MPI". In: *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Dublin, Ireland: Springer-Verlag, 2008, pp. 210–217 (cit. on p. 119).

[20] David Chaiken, John Kubiatowicz, and Anant Agarwal. *LimitLESS Directories: A Scalable Cache Coherence Scheme*. 1991 (cit. on p. 30).

[21] T. Chen et al. "Cell Broadband Engine Architecture and its first implementation – A performance view". In: *IBM Journal of Research and Development* 51.5 (2007), pp. 559–572 (cit. on pp. 19, 31).

[22] Intel Corporation. *SCC External Architecture Specification (EAS) Revision 1.1*. http://communities.intel.com/docs/DOC-5852. 2010 (cit. on pp. 30, 31).

[23] R.G. Covington et al. "The Efficient Simulation of Parallel Computer Systems". In: *International Journal in Computer Simulation*. 1991, pp. 31–58 (cit. on p. 118).

[24] Matthew Curtis-Maury et al. "Prediction-Based Power-Performance Adaptation of Multithreaded Scientific Codes". In: *IEEE Trans. Parallel Distrib. Syst.* 19 (10 Oct. 2008), pp. 1396–1410 (cit. on p. 47).

[25] R. David et al. "Dynamic power management of voltage-frequency island partitioned Networks-on-Chip using Intel's Single-chip Cloud Computer". In: *Networks on Chip (NoCS), 2011 Fifth IEEE/ACM International Symposium on*. May 2011, pp. 257–258 (cit. on p. 47).

[26] Alexandre Denis. "A High Performance Superpipeline Protocol for InfiniBand". In: *Proceedings of the 17th International Euro-Par Conference*. Lecture Notes in Computer Science 6853. Bordeaux, France: Springer, Aug. 2011, pp. 276–287 (cit. on p. 112).

[27] Sebastien Donadio et al. "A Language for the Compact Representation of Multiple Program Versions". In: *Intl. Workshop on Languages and Compilers for Parallel Computing*. Vol. 4339. Lect. Notes in Computer Science. Hawthorne, New York: Springer-Verlag, Oct. 2005, pp. 136–151 (cit. on p. 118).

[28] Ulrich Drepper. *What Every Programmer Should Know About Memory*. 2007 (cit. on p. 21).

[29] Alexandre X. Duchateau et al. "Languages and Compilers for Parallel Computing". In: ed. by José Nelson Amaral. Berlin, Heidelberg: Springer-Verlag, 2008. Chap. P-Ray: A Software Suite for Multi-core Architecture Characterization, pp. 187–201 (cit. on p. 69).

[30] Guillaume Mercier Emmanuel Jeannot and François Tessier. *Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques.* 2013 (cit. on p. 20).

[31] Agner Fog. *Instruction tables Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. http://www.agner.org/optimize/. 2011 (cit. on pp. 14, 37, 49, 57).

[32] D. Folegnani and A. Gonzalez. "Energy-effective issue logic". In: *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*. 2001, pp. 230–239 (cit. on p. 125).

[33] Grigori Fursin and Albert Cohen. "Building a Practical Iterative Interactive Compiler". Anglais. In: *International Workshop on Statistical and Machine Learning Approaches Applied to Architectures and C ompilation (SMART'07)*. Ghent, Belgium, Jan. 2007 (cit. on p. 14).

[34] Edgar Gabriel et al. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, Sept. 2004, pp. 97–104 (cit. on p. 107).

[35] A. Gara et al. "Overview of the Blue Gene/L system architecture". In: *IBM Journal of Research and Development* 49.2.3 (2005), pp. 195–212 (cit. on p. 30).

[36] S. Ramos Garea and T. Hoefler. "Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi". In: *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. New York City, NY, USA: ACM, June 2013, pp. 97–108 (cit. on p. 119).

[37] R. Ge and K.W. Cameron. "Power-Aware Speedup". In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. Mar. 2007, pp. 1–10 (cit. on p. 47).

[38] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. "Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems". In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 42. New York, New York: ACM, 2009, pp. 413–422 (cit. on pp. 59, 118).

[39] Tsuyoshi Hamada and Naohito Nakasato. "InfiniBand Trade Association, InfiniBand Architecture Specification, Volume 1, Release 1.0, http://www.infinibandta.com". In: *in International Conference on Field Programmable Logic and Applications, 2005*, pp. 366–373 (cit. on p. 20).

[40] Ziang Hu et al. *Programming Experience on Cyclops-64 Multi-Core Chip Architecture.* (Cit. on p. 32).

[41] C.J. Hughes et al. "Rsim: simulating shared-memory multiprocessors with ILP processors". In: *Computer* 35.2 (2002), pp. 40–49 (cit. on p. 118).

[42] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. "Cache-aware Roofline model: Upgrading the loft". In: *IEEE Computer Architecture Letters* 99.RapidPosts (2013), p. 1 (cit. on p. 118).

[43] *Intel MPI Benchmarks.* http://software.intel.com/en-us/articles/intel-mpi-benchmarks/ (cit. on p. 111).

[44] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes:1, 2A, 2B, 2C, 3A, 3B, and 3C.* http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. 2013 (cit. on p. 14).

[45] W. Jalby et al. "Wbtk: a new set of microbenchmarks to explore memory system performance for scientific computing". In: *Int. J. High Perform. Comput. Appl* 18 (2004) (cit. on p. 119).

[46] Aamer Jaleel et al. "High performance cache replacement using re-reference interval prediction (RRIP)". In: *SIGARCH Comput. Archit. News* 38.3 (June 2010), pp. 60–71 (cit. on pp. 58, 124).

[47] Sverre Jarp, Ryszard Jurga, and Andrzej Nowak. "Perfmon2: A leap forward in performance monitoring". In: *J.Phys.Conf.Ser.* 119 (2008), p. 042017 (cit. on p. 118).

[48] Daniel A. Jiménez. "Piecewise linear branch prediction". In: *In The 1st JILP Championship Branch Prediction Competition (CBP-1).* 2005, pp. 382–393 (cit. on p. 9).

[49] Guido Juckeland et al. "BenchIT - Performance Measurements and Comparison for Scientific Applications." In: *PARCO.* Ed. by Gerhard R. Joubert et al. Vol. 13. Advances in Parallel Computing. Elsevier, Feb. 7, 2005, pp. 501–508 (cit. on pp. 68, 89).

[50] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. "Interval-based models for run-time DVFS orchestration in superscalar processors". In: *Conf. Computing Frontiers.* 2010, pp. 287–296 (cit. on p. 47).

[51] Janghaeng Lee et al. "Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications". In: *Proceedings of the 37th annual international symposium on Computer architecture.* ISCA '10. Saint-Malo, France: ACM, 2010, pp. 270–279 (cit. on p. 92).

[52] Sang-jeong Lee, Hae-kag Lee, and Pen-chung Yew. "Runtime Performance Projection Model for Dynamic Power Management". In: *Asia-Pacific Computer Systems Architectures Conference.* 2007, pp. 186–197 (cit. on p. 47).

[53] Abid M. Malik. *Optimal basic block instruction scheduling for multiple-issue processors using constraint programming.* Tech. rep. In: Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence, 2005 (cit. on pp. 14, 125).

[54] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. "Why on-chip cache coherence is here to stay". In: *Commun. ACM* 55.7 (July 2012), pp. 78–89 (cit. on pp. 30, 124).

[55] T.G. Mattson et al. "The 48-core SCC Processor: the Programmer's View". In: *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for.* 2010, pp. 1–11 (cit. on p. 47).

[56] Tim Mattson and Rob van der Wijngaart. *RCCE: a Small Library for Many-Core Communication*. http://communities.intel.com/docs/DOC-5628. 2010 (cit. on p. 31).

[57] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Tech. rep. A continually updated technical report. http://www.cs.virginia.edu/stream/. Charlottesville, Virginia: University of Virginia, 1991-2007 (cit. on pp. 59, 68, 89, 119).

[58] Gregoire P Millet et al. "Combining hypoxic methods for peak performance". In: *Sports medicine* 40.1 (2010), pp. 1–25 (cit. on p. 14).

[59] D. Molka et al. "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System". In: *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*. 2009, pp. 261–270 (cit. on p. 20).

[60] Philip J. Mucci et al. "PAPI: A Portable Interface to Hardware Performance Counters". In: *In Proceedings of the Department of Defense HPCMP Users Group Conference*. 1999, pp. 7–10 (cit. on pp. 118, 120).

[61] Nicholas Nethercote and Julian Seward. "Valgrind: A program supervision framework". In: *In Third Workshop on Runtime Verification (RV'03)*. 2003 (cit. on p. 118).

[62] *OSU Micro-Benchmarks*. http://mvapich.cse.ohio-state.edu/benchmarks/ (cit. on p. 111).

[63] R. Braithwaite P. McCormick and W. Feng. *Empirical memory-access cost models in multicore numa architectures* (cit. on p. 62).

[64] Mark S. Papamarcos and Janak H. Patel. "A low-overhead coherence solution for multiprocessors with private cache memories". In: *SIGARCH Comput. Archit. News* 12.3 (Jan. 1984), pp. 348–354 (cit. on pp. 28, 93).

[65] S. Pellegrini, T. Hoefler, and T. Fahringer. "On the Effects of CPU Caches on MPI Point-to-Point Communications". In: *Proceedings of the 2012 IEEE International Conference on Cluster Computing*. Beijing, China: IEEE Computer Society, Sept. 2012, pp. 495–503 (cit. on p. 119).

[66] Simone Pellegrini et al. "Optimizing MPI Runtime Parameter Settings by Using Machine Learning". In: *EuroPVM/MPI*. Vol. 5759. Lecture Notes in Computer Science. Espoo, Finland: Springer, Sept. 2009, pp. 196–206 (cit. on p. 119).

[67] Radu Prodan Philipp Gschwandtner Thomas Fahringer. "Performance Analysis and Benchmarking of the Intel SCC". In: *Conference on Cluster Computing*. 2011, pp. 139–149 (cit. on p. 47).

[68] Debian project. https://wiki.debian.org/Hugepages (cit. on p. 70).

[69] Moinuddin K. Qureshi et al. "Adaptive insertion policies for high performance caching". In: *SIGARCH Comput. Archit. News* 35.2 (June 2007), pp. 381–391 (cit. on pp. 58, 124).

[70] B. Rountree et al. "Practical performance prediction under Dynamic Voltage Frequency Scaling". In: *Green Computing Conference and Workshops (IGCC), 2011 International*. July 2011, pp. 1–8 (cit. on p. 47).

[71] J. Seward, N. Nethercote, and J. Weidendorfer. *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux applications*. Network Theory Ltd., 2008 (cit. on p. 58).

[72] André Seznec and Pierre Michaud. "A case for (partially) TAgged GEometric history length branch prediction". In: *Journal of Instruction Level Parallelism* 8 (2006), pp. 1–23 (cit. on p. 9).

[73] André Seznec et al. *About Effective Cache Miss Penalty on Out-of-Order Superscalar Processors*. 1995 (cit. on p. 11).

[74] André Seznec et al. "Design tradeoffs for the Alpha EV8 conditional branch predictor". In: *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. IEEE. 2002, pp. 295–306 (cit. on p. 9).

[75] J.P. Singh, Harold S. Stone, and D.F. Thiebaut. "A model of workloads and its use in miss-rate prediction for fully associative caches". In: *Computers, IEEE Transactions on* 41.7 (1992), pp. 811–825 (cit. on p. 92).

[76] Carl Staelin and Hewlett-packard Laboratories. "lmbench: Portable Tools for Performance Analysis". In: *In USENIX Annual Technical Conference*. 1996, pp. 279–294 (cit. on p. 59).

[77] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. "Analytical cache models with applications to cache partitioning". In: *Proceedings of the 15th international conference on Supercomputing*. ICS '01. Sorrento, Italy: ACM, 2001, pp. 1–12 (cit. on p. 92).

[78] *The SCC Programmer's Guide*. 2011 (cit. on p. 49).

[79] Top500. *Top 500 Supercomputer Sites*. http://www.top500.org/. 2010 (cit. on p. 20).

[80] J. Treibig, G. Hager, and G. Wellein. "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments". In: *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*. San Diego CA, 2010 (cit. on pp. 14, 37, 67, 89).

[81] Jan Treibig, Georg Hager, and Gerhard Wellein. "Performance patterns and hardware metrics on modern multicore processors: best practices for performance engineering". In: *Proceedings of the 18th international conference on Parallel processing workshops*. Euro-Par'12. Rhodes Island, Greece: Springer-Verlag, 2013, pp. 451–460 (cit. on pp. 59, 119).

[82] Spyridon Triantafyllis et al. "Compiler optimization-space exploration". In: *In Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2003, pp. 204–215 (cit. on p. 14).

[83] J. Weidendorfer. http://valgrind.org/docs/manual/cg-manual.html (cit. on p. 58).

[84] Kent Wilken, Jack Liu, and Mark He. "Optimal Instruction Scheduling Using Integer Programming". In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. ACM Press, 2000, pp. 121–133 (cit. on pp. 14, 125).

[85] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76 (cit. on p. 118).

[86] Wm. A. Wulf and Sally A. McKee. "Hitting the memory wall: implications of the obvious". In: *SIGARCH Comput. Archit. News* 23.1 (Mar. 1995), pp. 20–24 (cit. on p. 20).

[87] Chi Xu et al. "Cache contention and application performance prediction for multi-core systems". In: *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*. 2010, pp. 76–86 (cit. on p. 59).

[88] Kamen Yotov et al. "A Comparison of Empirical and Model-driven Optimization". In: *SIGPLAN Not.* 38.5 (May 2003), pp. 63–76 (cit. on p. 56).

[89] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. "Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs?" In: *SIGPLAN Not.* 45.5 (Jan. 2010), pp. 203–212 (cit. on p. 99).

List of Publications

International Conference with Committee

[A1] Bertrand Putigny, Brice Goglin, and Denis Barthou. "Performance modeling for power consumption reduction on SCC". In: *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium*. Ed. by Peter Tröger and Andreas Polze. Technical Reports of University of Postdam Hasso Plattner Institute 55. Potsdam, Germany, Feb. 2012, pp. 21–26 (cit. on p. 46).

[A2] Bertrand Putigny, Benoit Ruelle, and Brice Goglin. "Analysis of MPI Shared-Memory Communication Performance from a Cache Coherence Perspective". In: *Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2014)*. Phoenix, USA, May 2014.

National Communications with Committee

[B1] Brice Goglin and Bertrand Putigny. "Idée reçue: Comparer la puissance de deux ordinateurs, c'est facile !" Français. In: *Interstices* (Apr. 2013). http://interstices.info/idee-recue-informatique-26.

Non-Refereed National Communications

[C1] Bertrand Putigny, Denis Barthou, and Brice Goglin. *Modélisation du coût de la cohérence de cache pour améliorer le tuilage de boucles*. Quatrième rencontres de la communauté française de compilation, Saint-Hippolyte, France. Dec. 2011 (cit. on p. 60).