Behavioral Application-dependent superscolor core modeling

Ricardo Andrés Velásquez Vélez

▶ To cite this version:

Ricardo Andrés Velásquez Vélez. Behavioral Application-dependent superscolor core modeling. Other [cs.OH]. Université de Rennes, 2013. English. NNT : 2013REN1S100 . tel-00942289

HAL Id: tel-00942289
https://theses.hal.science/tel-00942289

Submitted on 5 Feb 2014

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.

ANNÉE 2013

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : INFORMATIQUE

École doctorale Matisse

présentée par

Ricardo Andrés VELÁSQUEZ VÉLEZ

préparée à l'unité de recherche INRIA – Bretagne Atlantique
Institut National de Recherche en Informatique et Automatique
ISTIC

Behavioral Application-dependent Superscalar Core Modeling

Thèse soutenue à Rennes le 19 April 2013

devant le jury composé de :

Smail NIAR
Professeur à l'Université de Valenciennes / *Rapporteur*

Lieven EECKHOUT
Professeur à l'Université de Gent / *Rapporteur*

Frédéric PÉTROT
Professeur à l'Institut Polytechnique de Grenoble / *Examinateur*

Steven DERRIEN
Professeur à l'université de Rennes 1 / *Examinateur*

André SEZNEC
Directeur de recherche à l'INRIA / *Directeur de thèse*

Pierre MICHAUD
Chargée de recherche à l'INRIA / *Co-directeur de thèse*

Contents

1 Introduction **5**
- 1.1 Context . . . 5
- 1.2 Research Questions . . . 7
- 1.3 Contributions . . . 7
- 1.4 Thesis Outline . . . 8

2 State of the Art **9**
- 2.1 Introduction . . . 9
- 2.2 Computer architecture simulators . . . 10
  - 2.2.1 Some simulator terminology . . . 11
  - 2.2.2 Simulator architectures . . . 11
    - 2.2.2.1 Integrated simulation . . . 12
    - 2.2.2.2 Functional-first . . . 13
    - 2.2.2.3 Timing-first . . . 14
    - 2.2.2.4 Timing-directed . . . 14
  - 2.2.3 Improving simulators performance . . . 15
  - 2.2.4 Approximate simulators . . . 15
    - 2.2.4.1 Analytical models . . . 15
    - 2.2.4.2 Structural core models . . . 16
    - 2.2.4.3 Behavioral core models . . . 17
    - 2.2.4.4 Behavioral core models for multicore simulation . . . 18
- 2.3 Simulation methodologies . . . 18
  - 2.3.1 Workload design . . . 18
    - 2.3.1.1 Single-program workloads . . . 19
    - 2.3.1.2 Multiprogram workloads . . . 20
  - 2.3.2 Sampling simulation . . . 21
    - 2.3.2.1 Statistical sampling . . . 21
    - 2.3.2.2 Representative Sampling . . . 22
  - 2.3.3 Statistical simulation . . . 23
- 2.4 Performance metrics . . . 24
  - 2.4.1 Single-thread workloads . . . 24

2.4.2 Multi-thread workloads . . . 25
2.4.3 Multiprogram workloads . . . 25
2.4.3.1 Prevalent metrics . . . 25
2.4.3.2 Other metrics . . . 26
2.4.4 Average performance . . . 26

3 Behavioral Core Models 29
3.1 Introduction . . . 29
3.2 The limits of approximate microarchitecture modeling . . . 31
3.3 The PDCM behavioral core model . . . 33
3.3.1 PDCM simulation flow . . . 34
3.3.2 Adapting PDCM for detailed OoO core . . . 34
3.3.2.1 TLB misses and inter-request dependencies . . . 35
3.3.2.2 Write-backs . . . 36
3.3.2.3 Branch miss predictions . . . 36
3.3.2.4 Prefetching . . . 36
3.3.2.5 Delayed hits . . . 37
3.3.3 PDCM limitations . . . 38
3.4 BADCO: a new behavioral core model . . . 38
3.4.1 The BADCO machine . . . 39
3.4.2 BADCO model building . . . 41
3.5 Experimental evaluation . . . 43
3.5.1 Metrics . . . 43
3.5.2 Quantitative accuracy . . . 45
3.5.3 Qualitative accuracy . . . 45
3.5.4 Simulation speed . . . 47
3.6 Modeling multicore architectures with BADCO . . . 49
3.6.1 Experimental setup . . . 50
3.6.2 Experimental results . . . 51
3.6.3 Multicore simulation speed . . . 52
3.7 Summary . . . 54

4 Multiprogram Workload Design 55
4.1 Introduction . . . 55
4.2 The problem of multiprogram workload design . . . 56
4.3 Random sampling . . . 57
4.4 Experimental evaluation . . . 59
4.4.1 Simulation setup . . . 59
4.5 Experimental results for random sampling . . . 60
4.5.1 Random sampling model validation . . . 60
4.5.2 Performance difference impacts the sample size . . . 61
4.5.3 Different metrics may require different sample sizes . . . 64

4.6 Alternative sampling methods . . . 64
4.6.1 Balanced random sampling . . . 64
4.6.2 Stratified random sampling . . . 65
4.6.2.1 Benchmark stratification . . . 66
4.6.2.2 Workload stratification . . . 67
4.6.3 Actual degree of confidence . . . 67
4.7 Practical guidelines in multiprogram workload selection . . . 68
4.7.1 Simulation overhead: example . . . 69
4.8 Summary . . . 70

5 Conclusion **73**

A Résumé en français **77**
A.1 Introduction . . . 77
A.2 Contributions . . . 78
A.3 Modèles comportementaux . . . 79
A.3.1 Modèle comportemental PDCM . . . 80
A.3.2 BADCO : un nouveau modèle comportemental . . . 82
A.3.3 Évaluation expérimentale . . . 83
A.3.3.1 Précision simple cœur . . . 83
A.3.3.2 Précision multi-cœur . . . 84
A.3.3.3 Vitesse de simulation . . . 85
A.4 Sélection de charges de travail multiprogrammées . . . 86
A.4.1 Méthodes d'échantillonnage . . . 87
A.4.1.1 Échantillonnage aléatoire simple . . . 87
A.4.1.2 Échantillonnage aléatoire équilibré . . . 87
A.4.1.3 Échantillonnage aléatoire stratifié . . . 87
A.4.2 Évaluation expérimentale . . . 88
A.5 Conclusions . . . 89

Bibliography **99**

List of Figures **101**

List of Tables **103**

Chapter 1

Introduction

Many engineering fields allow us to build prototypes that are identical to the target design. They may cost more, but it is still feasible to build them. These prototypes can be tested under normal and extreme conditions. Hence, it is possible to verify that the design works properly and to define its physical limits. However, most other engineering fields make extensive use of simulation. Simulation has brought significant improvements to cars, airplanes, tires, computer systems, etc. If the target complexity remains constant, like in the physical world for example, then the simulation performance improves as computers get faster. This is not the case for computer systems simulation, because computers' complexity increases each new generation, and it increases faster than computers' performance. Moreover, the production of prototypes is extremely expensive and time consuming.

In the beginning of the computer age, computer architects relied on intuition and simple models to choose among different design points. Current processors are too complex to trust intuition. Computer architects require proper performance evaluation tools and methodologies to overcome processor complexity, and to make correct design decisions. Simulators allow computer architects to verify their intuition, and to catch issues that were not considered at all or incorrectly. Meanwhile, correct methodologies give confidence and generality to research conclusions.

1.1 Context

In recent years, research in microarchitecture has shifted from single-core to multicore processors. More specifically, the research focus has moved from core microarchitecture to *uncore* microarchitecture. Cycle-accurate models for many-core processors featuring hundreds or even thousands of cores are out of reach for simulating realistic workloads. A large portion of the simulation time is spend in the cores, and it is this portion that grows linear with every processor generation. Approximate simulation methodologies, which trade off accuracy for simulation speed, are necessary for conducting certain

research. In particular, they are needed for studying the impact of resource sharing between cores, where the shared resources can be caches, on-chip network, memory bus, power, temperature, etc.

Behavioral superscalar core modeling is a possible way to trade off accuracy for simulation speed in situations where the focus of the study is not the core itself, but what is outside the core, i.e., the *uncore*. In this modeling approach, a superscalar core is viewed as a black box emitting requests to the *uncore* at certain times. One or more behavioral core models can be connected to a cycle-accurate *uncore* model. Behavioral core models are built from detailed simulations. Once the time to build the model is amortized, important simulation speedups can be obtained. Moreover, behavioral core models enable vendors to share core models of its processors in order that third parties can work on specific design of the *uncore*.

Multicore processors also demand for more advanced and rigorous simulation methodologies. Many popular methodologies designed by computer architects for simulation of single core architectures must be adapted or even rethought for simulation of multicore architectures. For instance, *sampled simulation* and its different implementations have been created to reduce the amount of simulation time required, while still providing accurate simulation performance values for single thread programs. However, very few works have focused on how sampled simulation can be applied to multiprogram execution. Furthermore, some of the problems associated with sampled simulation, such as the cold start effect, have not been studied in the context of multicore architecture simulation yet.

An important methodology problem that has not received enough attention is the problem of selecting multiprogram workloads for the evaluation of multicore throughput. The population of possible multiprogram workloads may be very large. Hence, most studies use a relatively small sample of a few tens, or sometimes a few hundreds of workloads. Assuming that all the benchmarks are equally important, we would like this sample to be representative of the whole workload population. Yet, there is no standard method in the computer architecture community for defining multiprogram workloads. There are some common practices, but not really a common method. More important, authors rarely demonstrate the representativeness of their workload samples. Indeed, it is difficult to assess the representativeness of a workload sample without simulating a much larger number of workloads, which is precisely what we want to avoid. Approximate microarchitecture simulation methods that trade accuracy for simulation speed offer a solution to this dilemma. We show in this thesis that approximate simulation can help select representative multiprogram workloads for situations that require the accuracy of cycle-accurate simulation.

1.2 Research Questions

Currently, many research studies are focused on multicore processors. The complexity of multicore architectures impose a huge challenge to simulation techniques and methodologies. Due to time cost, cycle accurate simulators are out of consideration for tasks such as design space exploration, while approximate simulation exists as an option for faster simulation, but at the expense of accuracy. Moreover, common simulation methodologies such as sampling, warming and workload design must be reviewed and updated to target multicore experiments. The aim of this thesis is to provide computer architects with new simulation tools and methodologies that allow for faster and more rigorous evaluation of research ideas on multicore architectures. In particular, we first tackle the problem of slow simulation speed with behavioral core models; and second the problem of selecting multiprogram workloads.

Consequently, the main research questions of the thesis are:

- How, and at what cost, can behavioral core models model realistic superscalar core architectures?
- How, and at what cost, can behavioral core models speed up multicore simulation?
- How can we select a representative sample of multiprogram workloads for performance evaluation of multicore architectures?

1.3 Contributions

The main contributions of this thesis can be summarized as follows:

BADCO: a new method for defining behavioral core models We describe and study a new method for defining behavioral models for modern superscalar cores. The proposed Behavioral Application-Dependent Superscalar Core model, BADCO, predicts the execution time of a thread running on a superscalar core with an average error of less than 3.5 % in most cases. We show that BADCO is qualitatively accurate, being able to predict how performance changes when we change the *uncore*. The simulation speedups we obtained are typically between one and two orders of magnitude.

Adapting PDCM to model realistic core architectures We study the PDCM model, a previously proposed behavioral core model, evaluating its accuracy for modeling modern superscalar core architectures. We identify the approximations that reduce PDCM's accuracy for modeling realistic architectures. Then, we propose and implement some modifications to the PDCM model core features such as branch miss prediction and prefetch modules in level-1 caches. We reduce the average error from approximately 8 % with the original PDCM, to roughly 4 % with our improved PDCM model.

Workload stratification: a new methodology for selecting multiprogram workloads We propose and compare different sampling methods for defining multiprogram workloads for multicore architectures. We evaluate their effectiveness on a case study that compares several multicore last-level cache replacement policies. We show that random sampling, the simplest method, is robust enough to define a representative sample of workloads, provided the sample is big enough. We propose a new method, workload stratification, which is very effective at reducing the sample size in situations where random sampling would require a large sample size. Workload stratification uses approximate simulation for estimating the required sample size.

New analytical model for computing the degree of confidence of random samples Confidence intervals are the most common method to compute the degree of confidence of random samples. We propose an alternative method where the degree of confidence is defined as the probability of drawing correct conclusions when comparing two design points. This analytical method computes the degree of confidence as a function of the sample size and the coefficient of variation. The method can be used either to compute the confidence of a sample or the sample size provided that we can measure the coefficient of variation. We show that an approximate simulator can help in the estimation of the coefficient of variation.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. First, Chapter 2 presents the main theory and techniques related to computer simulation. Chapter 3 presents, evaluates and compares two behavioral core models in the context of single and multicore simulation. Then, Chapter 4 presents and compares different sampling methodologies for selecting multiprogram workloads. Finally, Chapter 5 concludes this thesis by presenting a summary of contributions, and provides some directions for future work.

Chapter 2

State of the Art

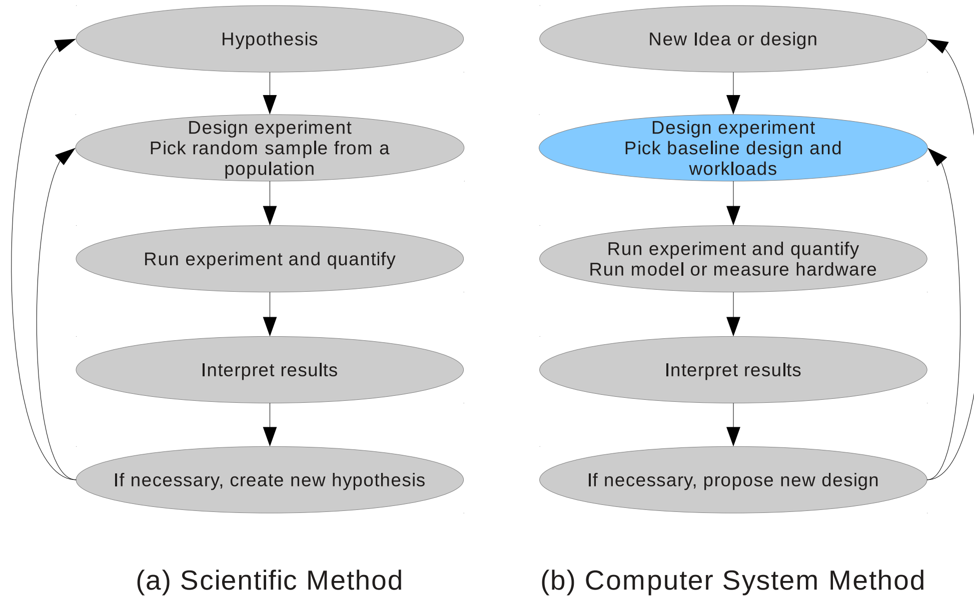
2.1 Introduction

Many simulation tools and methodologies have been proposed to evaluate the performance of computer systems accurately. In general, a rigorous performance evaluation for an idea/design implies that a computer architect has to make four important decision: choose the proper modeling/simulation technique, select an adequate baseline configuration, define a representative workload sample, and select a meaningful performance metric.

The modeling/simulation technique determines the balance between speed and accuracy. In order to overcome the problem of slow simulation tools and huge design space, computer architects use simulation techniques that increase the abstraction level and thus sacrifice accuracy to get speedup. The main simulation techniques include: detailed simulation, analytical modeling, approximate simulation, statistical simulation, sampled simulation, etc. Note that some of these techniques are orthogonal and may be combined.

In [24], Eeckhout makes a comparison between the *scientific method*, in figure 2.1(a), and the *computer system method*, in figure 2.1(b). He notes that compared to the scientific method, the computer method losses rigorousness with selection of the baseline systems and the workload sample. The selection of the baseline system is generally arbitrary, and in most cases something similar happens with the workload sample selection. Because conclusions may dependent on the baseline systems, the workload sample, or both, the computer method does not guarantee the generality of the conclusions. There is a strong need for making less subjective the task of workload selection. Rigorous performance evaluation is crucial for correct design and for driving research in the right direction.

In this chapter we present the state of the art of prevalent simulation tools and methodologies in the field of computer architecture. The chapter is organized as follows: Section 2.2 presents a taxonomy of computer architecture simulators, and dis-



(a) Scientific Method (b) Computer System Method

Figure 2.1: Scientific method versus computer systems method.

cusses about common techniques to accelerate simulators; Then, Section 2.3 introduces prevalent simulation methodologies; Finally, Section 2.4 describes the most popular throughput metrics.

2.2 Computer architecture simulators

The target of computer simulators is to predict the behavior of computer systems. Generally what we want to predict is timing performance, but other interesting behaviors are: power consumption, energy efficiency, temperature, reliability, yield, etc. Computer architects and software developers use simulators to verify intuitions about changes or new ideas in a computer's micro/architecture (architects); and new software or software optimizations (software developers).

We want a simulator to be fast, accurate, and easy to expand with new functionalities. A fast simulator enables: wider exploration, deeper exploration, stronger confidence and automation. For software development, slowdowns of 10 to 100 are tolerated provided there is enough functionality. For computer architects, accuracy is the most desirable characteristic. However, very often computer architects face the need to trade off accuracy for speed. There are different levels of accuracy depending on the level of abstraction used by the simulator. For example, cycle-accurate simulators exactly match RTL cycle count for performance. It is difficult to quantify which is the minimum level of accuracy tolerated. In general, we want enough accuracy to make comparison and identify tendencies correctly.

2.2.1 Some simulator terminology

Functional-only simulators only execute the instructions in the target *instruction set architecture* (ISA) and sometimes emulate some other computer peripherals such as hard-disk, network, usb, etc. Functional-only simulators do not provide timing estimates. Many *functional-only* simulators have evolved into virtual machines, and many others have been extended with temporal models [66, 5, 6]. *Functional-only* simulators have average slowdowns between 1.25x and 10x depending in the provided functionality [2].

Application simulators or user-level simulators provide basic support for the OS and system calls. *Application* simulators do not simulate what happens upon an operating system call or interrupt. The most common approach is to ignore interrupts and emulate the effect of system calls [3, 64]. *Application* simulators have sufficient functionality for some workloads, e.g., SPEC CPU benchmarks spend little time executing system-level code [24].

Full-system simulators give full support of the OS and the peripherals. A *Full-system* simulator can simulate a whole computer system such that complete software stacks can run on the simulator [24]. Software applications, being simulated in a *full-system* simulator, have the illusion of running on real hardware. Well-known examples of *full-system* simulator are SimOS [83], Virtutech's SimICs [66], AMD's SimNow [5], etc.

Trace-driven simulators take program instructions and address traces, and feeds the full benchmark trace into a detailed microarchitecture timing simulator [24]. *Trace-driven* simulators separate functional simulation from timing simulation. The main advantage is that the functional simulation is performed once and can be used to evaluate performance of many microarchitecture configurations. Some disadvantages of *trace-driven* simulation include: need for storing the trace files; the impossibility of modeling the effects along mispredicted paths; and impossibility to model the impact of the microarchitecture on inter-thread ordering when simulating multi-thread workloads.

Execution-driven simulators combine functional with timing simulation. *Execution-driven* simulators do not have the disadvantages of trace-driven simulators and are the de-facto simulation approach [24]. The higher accuracy of *execution-driven* simulators comes at the cost of increased development time and lower simulation speed.

2.2.2 Simulator architectures

The simulator architecture characterizes simulators based on their major internal interfaces. These interfaces enable reuse and ease modifications and expansions. The

selection of the architecture has an impact on the simulator's speed; the ability to start with a simple model and progressive increase level of detail (accuracy/refinability); the capacity to simulate a wide range of targets (generality); and the amount of work required to write the simulator (effort).

Simulators are software programs characterized by successive state updates of the physical components they model. The way those states are updated may change from simulator to simulator, and depending on the abstraction level. For example, the updates can be generated by an approximate model at memory transaction level, or by an RTL model at register level. The number of state updates is correlated with the accuracy and speed of the simulator. Many state updates means higher accuracy and lower simulation speed. In general, computers architects prefer accuracy over speed and system developers prefer speed over accuracy.

A common characteristic among different simulator's architectures is to split the work in two: *functional model* (FM) and *timing model* (TM). The *functional model* models the instruction set architecture (ISA) and the peripherals; It can execute directly the code or a trace of instructions. The *timing model* models the performance: number of cycles, power/thermal, reliability, etc. The timing models are approximations to the real counterparts, and the concept of accuracy of a timing simulation is needed to measure the fidelity of these simulators with respect to existing systems.

There exists at least four widely used architectures for cycle-accurate simulators: Integrated [92, 101], Functional-First [78, 2], Timing-First [67], and Timing-Directed [74, 8, 64, 3].

2.2.2.1 Integrated simulation

Integrated simulators model the functionality and the timing together. Hence, there is a single instance for each architectural/micro-architectural state. Such an instance defines the logical and physical behavior. Data paths, timing, functionality must all be correct. If the timing is broken, functionality likely will be broken too. As a consequence, integrated simulators are self-verifying.

In general, *integrated* simulators require a lot of work and they are also difficult to customize and parallelize. An *integrated* simulator requires an amount of work equivalent to that of an RTL simulator. The accuracy will be also close to RTL simulators but faster. Integration makes this kind of simulators difficult to customize and parallelize. There is a lot of work to make *integrated* simulators modular [92, 101]. Parallelization is possible in hardware, but then becomes an implementation.

Vachharajani et al. present Liberty Simulation Environment (LSE) [92]. More than a simulator, LSE is a programming model specialized to develop *integrated* simulators. The main target of LSE is to increase the re-usability of components. LSE abstracts the timing control through stalls. There are two kinds of stall: semantic and structural. Semantic stalls are related to the computation time of the components. Structural stalls are the ones that traditionally occur due to limited resources. LSE automatizes

the generation of most structural stalls.

PTLsim [101] is a full-system simulator for x86-64 architectures. The level of complexity is comparable to an RTL simulator. In order to increase performance and re-usability, PTLsim's source code provides two libraries: Super Standard Template Library (SuperSTL) and Logic Standard Template Library (logicSTL). LogicSTL is an internally developed add-on to SuperSTL which supports a variety of structures useful for modeling sequential logic. Some of its primitives may look familiar to Verilog or VHDL programmers [100].

2.2.2.2 Functional-first

In a *functional-first* simulator, the TM fetches instructions from a trace. The trace can be generated on the fly (execution-driven) or it may be stored on disk and piped to the TM (trace-driven). The trace is used to predict the temporal behavior of the target. The TM may back-pressure the FM but otherwise doesn't have any control over the functional model.

Functional-first simulators enable the FM and TM to be developed and optimized independently. There are no round-trip dependencies between both models and back-pressure is the only communication between models. The FM just needs to generate a trace. This job can be performed by a *functional-only* simulator, a *binary instrumented tool* such as pin, or a *trace-capable hardware.*

In brief, *functional-first* simulators make easy to parallelize between FM and TM, i.e., you may run FM and TM in different threads or even run the FM in a processor and the TM in an FPGA. They are fast simulators and require less development effort. On the other side, the accuracy, refinability, and generality are low. *Functional-first* simulators incur two main inaccuracies: modeling of mispredicted paths and the divergence in the ordering of memory accesses performed by the FM with respect to the TM [13]. This situation is especially critical for multicore systems. Cotson [2] and FAST [14] are examples of *functional-first* simulators.

COTSon is a multicore *full-system* simulator developed by AMD and HP [2]. COTSon uses SimNow [5] as functional simulator and supports timing models with different levels of detail. The FM drives the simulation. COTSon uses dynamic sampling to measure the performance through the TM. Hence, TMs just work for small intervals, allowing the simulator to run faster. The IPC captured by the TMs is fed back to the FM. The FM uses the IPCs to control the progress of the different threads. Additionally to samples' IPCs, COTSon also uses parameter fitting techniques to predict performance between samples.

FAST is a full system simulator for x86 architectures, whose TM runs on an FPGA [14]. FAST addresses one of the inaccuracies of *functional-first* simulators modeling the mispredicted path. Hence, the TM on the FPGA informs the FM when a misprediction occurs, then the FM provides the flow of instructions on the wrong-path. Once the TM solves the branch and communicates this to the FM, the FM starts to feed the TM

again with the correct path.

2.2.2.3 Timing-first

A *timing-first* simulator is generally an *integrated* simulator (full simulator) that runs in parallel with a *functional-only* simulator or *oracle*. The *functional-only* simulator provides the implementation of the instructions not available in the full simulator, and allows verification by comparing values with the full simulator. The main advantage of *timing-first* simulators is to remove the constraint of simulation exactness and completeness. Hence, it is not necessary to implement all the instructions in the full simulator from the beginning because the *functional-only* simulator can handle them. *Timing-first* simulators can improve accuracy compared to *functional-only* simulators for the instructions that are executed by the full simulator. Furthermore, timing-first simulators can only deal with a ISA supported by the *functional-only* simulator. In summary, timing-first simulators have low speed, high potential accuracy and refinability, medium generality, and the development time depends on how accurate one wants to make the simulator.

An example of timing-first simulator is GEMS [67]. GEMS uses SimICs [66] as functional (full-system) simulator. SimICs avoids implementing rare but effort-consuming instructions in the timing simulator. Timing modules interact with SimICs to determine when it should execute an instruction. GEMS also provide independent timing models for the memory system (Ruby) and the superscalar core (Opal). This allows to configure the simulation with different levels of detail. Ruby is a timing simulator of a multiprocessor memory system that models: caches, cache controllers, system interconnect, memory controllers, and banks of main memory. Opal also known as TFSim [69] is a detailed TM that runs ahead of Simics' functional simulation by fetching, decoding, predicting branches, dynamically scheduling, executing instructions, and speculatively accessing the memory hierarchy.

2.2.2.4 Timing-directed

Timing-directed simulators also split the work in functional modeling and time modeling. However, compared to *functional-first* or *timing-first* simulators, the coupling between the TM and the FM is higher. Every TM state has an equivalent FM state that is called at the correct time and in the correct order. The architectural state lives in the FM to simplify the TM. *Execute-in-execute* is a special case of *timing-directed* simulators [74]. In an *execute-in-execute* simulator, an instruction is executed in the FM when it is executed by the TM.

The FM in a *timing-directed* simulator is very target dependent, i.e., the FM is partitioned exactly like the TM and only supports what the target supports. On the other side, the TM has no notion of values; instead, it gets the effective addresses from the FM, which it uses to determine cache hits and misses, access the branch predictor, etc.

Implementing a *timing-directed* simulator requires a minimum level of accuracy because neither TM nor FM can operate on its own. In summary, *timing-directed* simulators are slow (TM is the main bottleneck) and require a lot of development effort. Timing-directed simulators are difficult to parallelize across simulator boundaries. Moreover, they exhibit good refinability and generality .

2.2.3 Improving simulators performance

The main problem of computer architecture simulation is the simulation speed. Accurate simulators are slow. Industrial simulator, for example, may be from 10000 to 1 million times slower than native execution [24]. Besides, computer complexity grows faster than its speed, thus simulators become relatively slower with each new processor generation. Multicore processors aggravate the problem. There is at least a linear slowdown when simulating parallel cores on a sequential host. Moreover, the accuracy becomes more important due to the complexity of the parallel interactions.

There are two approaches to improve performance: (1) reducing the amount of work, either increasing efficiency or eliminating unnecessary work; (2) Exploit parallelism with multicore/multiprocessor host, FPGAs, or a combination of both.

2.2.4 Approximate simulators

Several approximate microarchitecture simulation methods have been proposed [20, 11, 52, 72, 61, 84, 102] (the list is not exhaustive). In general, these methods trade accuracy for simulation speed. They are usually advocated for design space exploration and, more generally, for situations where the slowness of cycle-accurate simulators limits their usefulness.

Trace-driven simulation is a classical way to implement approximate processor models. Trace-driven simulation does not model exactly (and very often ignores) the impact of instructions fetched on mispredicted paths and it cannot simulate certain data misspeculation effects. The primary goal of these approximations is not to speed up simulations but to decrease the simulator development time. A trace-driven simulator can be more or less detailed : the more detailed, the slower.

2.2.4.1 Analytical models

What we call in this work *analytical model* is a mathematical equation used to estimate the performance of a microarchitecture as a function of microarchitectural parameters. Naturally, *analytical models* are less accurate than cycle-accurate simulators. However, they are of great interest because once a model is build, it gives very good simulation performance, simply evaluating an equation; and also because they provide more fundamental insights, apparent from the formula. Three methods have been used to build *analytical models*: statistical inference, neural networks and regression models.

The main goal of linear regression is to understand the effect of microarchitectural parameters and their interaction in the overall processor performance. Joseph et al. [49] use linear regression to create analytical models that estimate the overall processor performance. The selection of microarchitectural parameters involved in the model have a direct effect on the accuracy and the number of cycle-accurate simulations required. Insignificant parameters included in the model do not contribute to accuracy and increase the model building time. Therefore, a relevant parameter not considered leads to inaccurate models [24].

In several cases the assumption of linearity is too restrictive and the model requires to deal with non linear behavior. A common approach is to perform a transformation to the input and/or output variables and then use a linear regression method. Typical transformations are square root, logarithm, power, etc. The transformation is applied to the entire range of the variable. Hence, the transformation may work well in one range and bad in another [24]. Spline functions offer a way to deal with non-linearity without the drawbacks of variable transformations. A spline function is partitioned into intervals, each interval having its own fitting polynomials. In [57], Lee and Brooks use spline regression models to build multiprocessor performance models. Neural networks are an alternative approach for handling non linearity [45, 22, 50]. The accuracy of neural networks has been shown to be as good as spline-based regression models [58]. However, compared to neural networks, spline-based regression models provide more insight. Whereas, neural networks ease the automation of the building process.

2.2.4.2 Structural core models

Structural models speed up superscalar processor simulation by modeling only “first order" parameters, i.e., the parameters that are supposed to have the greatest performance impact in general. Structural models can be more or less accurate depending on how many parameters are modeled. Hence there is a tradeoff between accuracy and simulation speedup.

Loh described a time-stamping method [63] that processes dynamic instructions one by one instead of simulating cycle by cycle as in cycle-accurate performance models. A form of time-stamping had already been implemented in the DirectRSIM multiprocessor simulator [23, 90]. Loh's time-stamping method uses scoreboards to model the impact of certain limited resources (e.g., ALUs). The main approximation is that the execution time for an instruction depends only on instructions preceding it in sequential order. This assumption is generally not exact in modern processors.

Fields et al. used a *dependence graph* model of superscalar processor performance to analyze quickly the microarchitecture performance bottlenecks [35]. Each node in the graph represents a dynamic instruction in a particular state, e.g., the fact that the instruction is ready to execute. Directed edges between nodes represent dependences, e.g., the fact that an instruction cannot enter the reorder buffer (ROB) until the instruction that is ROB-size instructions ahead is retired.

Karkhanis and Smith described a “first-order" performance model [53], which was later refined [33, 12, 32]. Instructions are (quickly) processed one by one to obtain certain statistics, like the CPI (average number of cycles per instruction) in the absence of miss events, the number of branch mispredictions, the number of non-overlapped long data cache misses, and so on. Eventually, these statistics are combined in a simple mathematical formula that gives an approximate global CPI. The model assumes that limited resources, like the issue width, either are large enough to not impact performance or are completely saturated (in a balanced microarchitecture, this assumption is generally not true [75]). Nevertheless, this model provides interesting insights. Recently, a method called *interval simulation* was introduced for building core models based on the first-order performance model [39, 84]. Interval simulation permits building a core model relatively quickly from scratch.

Another recently proposed structural core model, called *In-N-Out*, achieves simulation speedup by simulating only first-order parameters, like interval simulation, but also by storing in a trace some preprocessed microarchitecture-independent information (e.g., longest dependence chains lengths), considering that the time to generate the trace is paid only once and is amortized over several simulations [60].

2.2.4.3 Behavioral core models

Kanaujia et al. proposed a behavioral core model for accelerating the simulation of multicore processors running homogeneous multi-programmed workloads [52] : one core is simulated with a cycle-accurate model, and the others cores mimic the cycle-accurate core approximately.

Li et al. used a behavioral core model to study multicores running heterogeneous multi-programmed workloads [62]. Their behavioral model simulates not only performance but also power consumption and temperature. The core model consists of a trace of level-2 (L2) cache accesses annotated with access times and power values. This per-application trace is generated from a cycle-accurate simulation of a given application, in isolation and assuming a fixed L2 cache size. Then, this trace is used for fast multicore simulations. The model is not accurate because the recorded access times are different from the real ones. Therefore the authors do several multicore simulations to refine the model progressively, the L2 access times for the next simulation being corrected progressively based on statistics from the previous simulation. In the context of their study, the authors found that 3 multicore simulations were enough to reach a good accuracy.

The ASPEN behavioral core model was briefly described by Moses et al. [72]. This model consists of a trace containing load and store misses annotated with timestamps [72]. Based on the timestamps, they determine whether a memory access is blocking or non-blocking.

Lee et al. proposed and studied several behavioral core models [15, 61]. These models consist of a trace of L2 accesses annotated with some information, in particular

timestamps, like in the ASPEN model. They studied different modeling options and found that, for accuracy, it is important to consider memory-level parallelism. Their most accurate model, Pairwise Dependent Cache Miss (PDCM), simulates the effect of the reorder buffer and takes into account dependences between L2 accesses. We describe in Section 3.3 our implementation of PDCM for the Zesto microarchitecture model.

2.2.4.4 Behavioral core models for multicore simulation

Behavioral core models can be used to investigate various questions concerning the execution of workloads consisting of multiple independent tasks [62, 102]. Once behavioral models have been built for a set of independent tasks, they can be easily combined to simulate a multicore running several tasks simultaneously. This is particularly interesting for studying a large number of combinations, as the time spent building each model is largely amortized.

Simulating accurately the behavior of parallel programs is more difficult. Trace-driven simulation cannot simulate accurately the behavior of non-deterministic parallel programs for which the sequence of instructions executed by a thread may be strongly dependent on the timing of requests to the uncore [40]. Some previous studies have shown that trace-driven simulation could reproduce somewhat accurately the behavior of *certain* parallel programs [40, 39], and it may be possible to implement behavioral core models for such programs [15, 82]. Nevertheless, behavioral core modeling may not be the most appropriate simulation tool for studying the execution of parallel programs.

2.3 Simulation methodologies

2.3.1 Workload design

Workload design consists in selecting from the workload space (all existing applications), a reduced set of workloads that is representative of the whole space. Workload design plays an important role in the the *computer system method*. A poor workload design will probably lead to a suboptimal architecture design, or to misleading conclusions. The meaning of the term workload and its associated workload space can change according to the object of study. In a single-core architecture, a workload is a single program or benchmark; and the workload space is the set of all applications that may run in a single-core architecture. However, for an study on a multicore or SMT architecture, a workload is a set of n programs; and the workload space is the set of all possible combinations of n programs that cant execute on the multicore/SMT architecture. In this work, we define a *multiprogram workload* as the set of n independent programs that run simultaneously in a multicore architecture with n cores.

2.3.1.1 Single-program workloads

When designing a representative workload, the goal is to select the smallest set of workloads that is representative of the workload space. The reference workload is the set of benchmarks that the experimenter believes to be representative of the workload space. In general, the reference workload space is still too big for practical simulation experiments. Hence, a reduced but still representative workload is necessary.

The design of a reference workload is a difficult task. The full workload space is huge and has different applications domains: general purpose, multimedia, scientific computing, bio-informatics, medical applications, commercial applications (databases and servers). As a consequence, it is possible to find several benchmark suits: SPEC-CPU [44, 91], MediaBench [59], PARSEC [7], DaCapo [9], STAMP [71], etc. Not all reference workloads are suited for every kind of study. Using the wrong reference workloads leads to suboptimal designs.

Another complexity on the design of reference workloads is that the workload space change on time. This is known as workload drift. Hence, we design future computer systems using yesterday's benchmarks.

The third challenges is that the process of including benchmarks in a benchmark suite is subjective. Citron et al. [16] survey current practices in benchmark subsetting. They found that a common practice is to do subsetting based on program language, portability and simulation infrastructure. This practice leads to misleading performance numbers.

There are two main methodologies to create a reduced but representative benchmark suite: Principal components analysis (PCA) [27], and Placket and Burman design of experiment (PBE) [99]. The target of these techniques is to reduce the amount of work required for performance studies. Hence, the techniques want to discard redundant benchmarks, i.e. benchmarks with similar behavior or that stress the same aspects of the design. We also want to omit benchmarks that not provide any insight in the context of the target design.

PCA is a well known statistical data analysis technique [48]. The objective of this technique is to transform a large set of correlated variables into a smaller set of uncorrelated variables. PCA presents a lower dimensional picture that yet captures the essence of the full set, but that is easier to analyze and understand.

In [27, 26] Eeckhout et al. present a methodology to use PCA on the analysis of workload behavior. With this methodology, workloads are characterized by a p-dimensional space of p-important metrics: instruction mix, ILP, branch prediction, code footprint, memory working set, memory access patterns, etc. Due to the complexity of current systems, the number of p-variables is too large. Moreover, they may be correlated making difficult to visualize and/or reason about the workload space. Hence, PCA is used to transform the p-dimensional workload space into a q-dimensional space, where $q \ll p$. The main hypothesis is that benchmarks close in the q-space have similar behavior. Then, *Cluster Analysis* [30] is used on the q-space to determine a reduced

but representative workload set.

Yi et al. in [99] use a Placket and Burman experiment [80] to understand how workload performance is affected by microarchitectural parameters. A PBE captures the effect of every microarchitectural parameter without simulating all possible combinations of them. In particular, a PBE requires $2c$ cycle accurate simulations for c microarchitectural parameters. The outcome of a PBE is a ranking of the most significant microarchitecture performance bottlenecks. This ranking is a unique signature for a benchmark. Hence, comparing the rankings across benchmarks allows to discern how different benchmarks are. I.e., if for two benchmarks, the top N most significant parameters are the same and have the same order, then one can conclude that the benchmarks have similar behavior.

2.3.1.2 Multiprogram workloads

Only a few papers have explored the problem of defining representative multiprogram workloads. The most obvious systematic method for defining multiprogram workloads is random selection. The advantage of random workload selection is that it is simple and less susceptible to bias. Indeed, if the author of a study has a very good understanding of a problem, he/she can identify "important" workloads. However, the behavior of modern superscalar processors is sometimes quite complex, and accurate simulators are needed to capture unintuitive interactions. This is why research in computer architecture is mostly based on simulation. Defining multiprogram workloads a priori, based on one's understanding of the studied problem, may inadvertently bias the conclusions of the study. Though random selection of workloads is a simple and obvious method, it is not clear how many workloads must be considered. Van Craeynest and Eeckhout have shown in a recent study [20] that using only a few tens of random workloads, as seen in some studies, does not permit evaluating accurately a throughput metric like weighted speedup [89] or harmonic mean of speedups [65]. In their experiments, about 150 random 4-thread workloads are necessary to be able to compute throughput with reasonable accuracy out of 29 individual SPEC CPU2006 benchmarks [91]. That is, random selection requires a sample of workloads larger than what is used in most studies. That may be a reason why most authors use a class-based selection method instead. In a class-based selection method, the experimenter classify benchmarks into classes and define workloads from these classes. In the vast majority of cases, the classes are defined "manually", based on the experimenters' understanding of the problem under study. Among the studies using class-based workload selection, very few are fully automatic. In a recent study, Vandierendonck and Seznec use cluster analysis to define 4 classes among the SPEC CPU2000 benchmarks [96]. Van Biesbrouck et al. [94] described a fully automatic method to define workloads using microarchitecture-independent profiling data. Instead of classifying benchmarks, they apply cluster analysis directly on points representing workloads.

2.3.2 Sampling simulation

Sampling is an established method for representing a data set using a fraction of the data. In the simulation context, a sample is a contiguous interval of dynamic instructions during program execution. Because simulating a benchmark to completion is too long, people generally simulate samples through the program's execution. There are two main approaches in sampling simulation: statistical sampling [42, 97] and representative sampling [41, 79]. Statistical sampling takes either random or periodic samples of instructions without special consideration of the sample location. Representative sampling carefully identifies phases in a program's execution and then uses those phases to select the sample location. In general, *functional-only* simulation is used to go from one sample to the next. The *functional-only* simulation is much faster than the cycle-accurate simulation mode.

2.3.2.1 Statistical sampling

Statistical sampling has a rigorous mathematical foundation in the field of inferential statistics, which offers well-defined procedures to quantify and to ensure the quality of sample derived estimates. Computer architecture researchers have proposed several different sampling techniques to estimate a program's behavior. Laha et al. in [56] propose a simulation method based on statistical techniques. The main target of the method was to reduce measurements in very large traces, and predict the mean miss rate and miss rate distribution of cache memory systems. They compared the sampled mean's accuracy and examined the distribution of random sampling to show that it matched that of the real trace and using just 7 % of the information.

In [18], Thomas Conte uses statistical sampling of address traces to evaluate cache systems improving the performance and trace size of traditional cache simulation. In more recent work [19], Conte et al. provided a framework that took random samples from the execution. They computed the samples' statistical metrics such as standard deviation, probabilistic error, and confidence bounds to predict the estimated results' accuracy, and statistically analyzed the metric of interest such as instructions per cycle [98]. Conte and colleagues specified two sources of error in their sampling technique: non-sampling bias and sampling bias. Non-sampling bias or cold-start effect results from improperly warming up the processor. Sampling bias, on the other hand, is fundamental to the samples, since it quantifies how accurately the sample average represents the overall average. Two major parameters influence this error, the number of samples and the size of each sample in instructions [98].

The smaller the sample size, the faster the simulation. But this comes at the cost of increased overhead and complexity because of the need for accurate sample warm-up. To determine the amount of samples to take, the user determines a particular accuracy level for estimating the metric of interest. The benchmark is then simulated and N samples are collected, N being an initial value for the number of samples. Error and

confidence bounds are computed for the samples, and, if they satisfy the accuracy limit, we are done. Otherwise, more samples ($> N$) must be collected, and the error and confidence bounds must be recomputed for each collected sample set until the accuracy threshold is satisfied. The SMARTS [97] framework proposes an automated approach for applying this sampling technique.

2.3.2.2 Representative Sampling

Representative sampling contrasts with statistical sampling in that it first analyzes the program's execution to identify and locate representative samples for each unique behavior in the program's execution. The main advantage of this approach is that having fewer samples can reduce simulation time and also allows for a simpler simulation infrastructure.

Representative sampling is based on the identification of phases through the execution of a program. A phase is a series of repeating patterns (loops and procedure calls). The phase behavior benefits simulation, because only a single sample per phase is required to have the general picture of the program execution. Sherwood et al. in [86, 87] present SimPoints, an automatic technique for finding and exploiting the phase behavior of programs independent of the micro/architecture. SimPoints is an infrastructure that chooses a small number of representative samples that, when simulated, represent the program's complete execution.

To accomplish this, SimPoints breaks a program's execution into fixed-length intervals of execution, for example, 100 million instructions. A basic-block vector (BBV) is defined for each interval with the occurrences of each basic-block during the interval. The basic assumption here is that the program behavior is related to the code it is executing. Hence, the number of times that each basic block executes in a time interval is a fingerprint of the program execution. Different intervals give different fingerprints. SimPoints then compares two vectors by computing their distance (euclidean or manhatan) from each other. Vectors close to each other are grouped into the same phase, or cluster, using the k-means algorithm from machine learning.

Only one interval is chosen from a cluster for detailed simulation because intervals with similar code signatures have similar architectural metrics. Simulating each of the representative samples together, one from each cluster, creates a complete and accurate representation of the program's execution in minutes.

SimPoints requires two runs, one functional run to collect BBVs, and one run for the sampled performance simulation itself. Therefore, upon each software modification, the full functional run must be done again, which is not practical. This issue has been addressed with on-line SimPoints [41], which only requires a single run and finds clusters of BBVs on the fly.

2.3.3 Statistical simulation

The main purpose of statistical simulation is to reduce the amount of time expended in cycle accurate simulation [98]. Carl et al. introduce statistical simulation as an alternative to cycle accurate simulation and analytical models [10]. Statistical simulator collects several statistical profiles from the program execution. Instruction mixes and dependence relationships profiles are collected during *functional-only* simulation. Cache miss rates and branch miss prediction profiles are collected with execution/trace-driven simulation. Statistical profiles are used to generate a synthetic trace that has the same characteristics, but is significantly shorter than the original program. Statistical profiles are also used to generate statistical models of caches and predictors. Synthetic traces typically contains 100,000 to 1,000,000 instructions [98]. Finally, the synthetic trace is simulated with a very simple core model.

The simulation model required for statistical simulation is simpler because instructions are synthesized into a small number of types. Moreover, statistical models for caches and predictors are also simpler than their detailed counterparts. Coupled with the very short traces, the simulation times for this kind of simulator are several orders of magnitude lower than cycle-accurate simulators.

The accuracy of statistical simulation have improved in recent research thanks to the inclusion of additional levels of correlation among program characteristics. Eeckhout et al. in [25] improve the statistical modeling accuracy extending Carl's work with a memory dependence profile and guaranteeing the syntactical correctness of the synthetic traces. Nussbaum et al. in [76] propose enhanced instruction models to generate synthetic traces. Hence, instead of a global instruction mix profile, Nussbaum et al. propose new statistical profiles of instruction mixes correlated to the abstraction of basic blocks. The proposed profiles are: basic block branch distance profile, basic block size profile, and combinations of these two with a global mix profile. The most accurate statistical simulation frameworks known to date include statistical flow graphs to model paths of execution [28]; as well as accurate memory data flow models for delayed hits, load forwarding and cache miss correlation [38].

Statistical simulation has been also proposed for simulation of symmetric multicore architectures. Nussbaum et al. in [77] collect statistics about barrier distribution, lock accesses and critical section mixes to extend statistical simulation to symmetric multiprocessor systems. They reach speedups of two orders of magnitude with average errors of 10 %. Genbrugge et al. in [37] studied statistical simulation as a fast simulation technique for chip multiprocessor running multiprogram workloads. For this purpose, they collect additional statistical profiles of per-set cache accesses and LRU-Stacks.

Statistical simulation has several applications. The most obvious is uniprocessor power and performance design. Experiments show that statistical simulation achieves excellent relative accuracy, making it extremely useful for early design stage exploration. Joshi et al. evaluate the efficacy of statistical simulation as a design space exploration tool[51]. They apply a Plackett & Burman experiment [80] to measure the

representativeness of synthetic traces with respect to real applications, and found that the first 10 bottlenecks identified by the experiment are shared by both synthetic and real applications. Given that a statistical profile reflects the key properties of the program's execution behavior, statistical simulation can accurately estimate performance and power[98]. This, combined with the simulation performance make it a perfect tool for design space exploration. Other applications of synthetic simulation include workload characterization, program analysis, and system-level exploration/design studies.

2.4 Performance metrics

Performance metrics are the foundation of experimental research and development for evaluating new ideas or design features. In this section, we present the most relevant metrics for computing performance of computer systems with single-thread, multithread and multiprogram workloads.

2.4.1 Single-thread workloads

For single-thread workloads the performance metric is very well defined: the total execution time T. In fact, Patterson and Hennessy in [43] sustain that the only consistent and reliable measure of performance is the execution time of real programs, and that all proposed alternatives have eventually led to misleading claims or even mistakes in computer design. Equation 2.1 presents the Iron Law of Performance:

$$T = N * CPI * \frac{1}{f} \tag{2.1}$$

where N is the number of instructions, CPI is the average number of cycles per instruction, and f is the frequency. The equation relates the three sources of performance: Instructions Set Architecture (N), the microarchitecture (CPI), and the technology (f). If N and f stay constant, then the CPI expresses the performance. The CPI is a *lower is better* performance metric. Some studies present the performance with CPI stacks, which show the number of cycles lost due to different characteristics of the system, like the cache hierarchy or branch predictor, and lead to a better understanding of each component's effect on total system performance [11, 24].

Another important single-thread performance metric is the average number of instructions per cycle or IPC. Where the IPC is the inverse of the CPI, i.e. $IPC = \frac{1}{CPI}$. Note also that the IPC is a *higher is better* metric. The IPC is very popular among computer architects, because it better characterizes a single-thread benchmark's behavior than the total execution time [70]. Computer architects are usually more interested in the benchmark's behavior, which they hope is representative, than in computing exactly the total execution time which generally depends on the program's inputs.

2.4.2 Multi-thread workloads

Performance metrics for multi-thread workloads are similar to single-thread, assuming that no other program is running in parallel. Hence, the most reliable performance metric is again the total execution time of the program. In the context of multi-thread workloads, the IPC is considered a not reliable measure of performance [24]. The number of instructions can change from one run to the next due to spin-lock loop instructions, which do not contribute to performance. Moreover, a higher IPC not necessarily mean more performance, and the other way around. This effect is more pronounced with an increasing number of processors or when applications spend a significant amount of time in OS-mode [1].

The user-mode IPC (*U-IPC*) is used in [42] as an alternative to the IPC, where only user-mode instructions are used to compute the IPC. This metric does not capture the performance of the system code. Emer and Clarck in [29] exclude the VMSNull process from the per-instructions statistics to address the spin-lock loop problem.

2.4.3 Multiprogram workloads

Simultaneous multi-threading processors and multicore processor have become mainstream. This has created a need for performance metrics for multiprogram workloads. A processor running a multiprogram workload executes multiple independent programs concurrently. The independent co-executing programs affect each other's performance due to shared resources. As a result, the programs compete for resources in the last level cache, interconnection network, off-chip bandwidth to memory, and the memory itself. Several different metrics have been proposed for quantifying the throughput of multicore processors. There is no clear consensus about which metric should be used. Some studies even use several throughput metrics [70].

2.4.3.1 Prevalent metrics

Several throughput metrics based on the IPC are commonly used in SMT and multiprogram studies. However, the fact that the metrics are based on IPC limits their applicability to single-thread benchmarks. The most frequently used ones are the IPC throughput, the weighted speedup, and the harmonic mean of speedups.

IPC throughput. In this work, we define the *IPC throughput* (IPCT) as the average of the IPCs of the co-executing programs. Equation 2.2 presents the IPCT metric in terms of individual program IPCs.

$$IPCT = \frac{1}{n}\sum_{i=1}^{n} IPC_i \tag{2.2}$$

Where, IPC_i is the IPC of the co-executing program i and n is the number of co-executing programs. Alternatively, some authors define the IPCT as the sum of IPC_i

[24, 95]. Eeckhout in [24] sustain that the IPCT must not be used for multiprogram due its lack of meaning in terms of user or system perspective.

Weighted speedup. Snavelly and Tullsen [89] propose the *weighted speedup* (WSU) metric. The meaning of WSU relates to the number of jobs completed by unit of time [24]. Equation 2.3 presents the WSU metric in terms of individual program IPCs.

$$WSU = \frac{1}{n}\sum_{i=1}^{n}\frac{IPC_i^{MP}}{IPC_i^{SP}} \tag{2.3}$$

Where, IPC_i^{MP} is the IPC of the program i during multiprogram execution, and IPC_i^{SP} is the IPC of the program i executing in single-program mode. If WSU is less than $1/n$, then the co-execution of the programs take longer in the shared system than it will take in a back-to-back execution. Note that WSU is a *higher is better* metric.

Harmonic mean. The *harmonic mean of speedups* (HSU) has been proposed as a metric to balance fairness and throughput [65]. Equation 2.3 presents the HSU metric in terms of individual program IPCs.

$$HSU = \frac{n}{\sum_{i=1}^{n}\frac{IPC_i^{SP}}{IPC_i^{MP}}} \tag{2.4}$$

Where, IPC_i^{MP} is the IPC of the program i during multiprogram execution, and IPC_i^{SP} is the IPC of the program i executing in single-program mode. HSU is a *higher is better* metric.

2.4.3.2 Other metrics

Additionally to the prevalent performance metrics, there are other metrics that have been proposed to deal with fairness and consistency.

Performance metrics can be analyzed from three different point of view: system perspective, user perspective and fairness. Eyerman and Eeckhout in [31] propose performance metrics for both user and system perspective: system throughput and average normalized turnaround time respectively. Vandierendonck and Seznec in [95] compare different fairness metrics.

2.4.4 Average performance

In general, the performance of a computer system is not a single number. For instance, when evaluating a single-core architecture one may have multiple benchmarks and at the same time multiple performance values for different executions of each benchmark. In the same way, evaluating the performance of multicore architectures requires evaluating multiple workload combinations. As a result, in order to compute the global

performance of an architecture, it is necessary to do an average. The selection of the proper mean has been a long time debate with two points of view: mathematics and statistics. The maths perspective favoring arithmetic and harmonic means [88, 21, 47], meanwhile the statistic point of view favoring the geometric mean [36, 68]. Others as Hennessy and Patterson [43] have shown the strengths and weaknesses of each mean.

The mathematical perspective starts from understanding the physical meaning of the performance metric, and then derives the average in a way that makes sense [24]. In this approach there are no assumptions about the distribution of the performance values, neither about the chosen workloads.

The *harmonic mean* (H-mean) must be used when the metric of interest is a ratio A/B and A is exactly the same for all benchmarks. For example, if the metric is IPC and the simulation of all benchmarks correspond to a fixed number of instructions. Equation 2.5 presents the definition of the H-mean.

$$\text{H-mean} = \frac{n}{\sum_{i=1}^{n} \frac{1}{PM_i}} \tag{2.5}$$

where PM_i is a performance metric.

The *arithmetic mean* must be used when the metric is a ratio A/B and B is weighted equally among benchmarks. For example, if the metric is MIPS and all benchmarks are simulated during the same amount of time. Equation 2.6 presents the definition of A-mean.

$$\text{A-mean} = \frac{1}{n} \sum_{i=1}^{n} PM_i \tag{2.6}$$

The statistical perspective makes several assumptions to prove the suitability of the *geometric mean* (G-mean). First, it assumes that benchmarks are selected randomly from a broader workload space. And second, it assumes that the speedups follow a log-normal distribution. This last assumption implies that some benchmarks experience much larger speedups than others. Equation 2.7 presents the definition of the G-mean.

$$\text{G-mean} = \sqrt[n]{\prod_{i=1}^{n} PM_i} \tag{2.7}$$

An interesting property is that the G-mean allows computing average speedups between two machines by dividing the average speedups for these to machines relative to some reference machine.

We can summarize the prevalent throughput metrics (see section 2.4.3.1) and the way to compute global performance as follows: The per-workload throughput for workload w is

$$t(w) = \underset{k \in [1,K]}{\text{X-mean}} \frac{IPC_{wk}}{IPC_{ref}[b_{wk}]} \tag{2.8}$$

where X-mean is the A-mean, H-mean or G-mean; IPC_{wk} is the IPC of the thread running on core k, $b_{wk} \in [1, B]$ is the benchmark on core k, and $IPC_{ref}[b]$ is the IPC for benchmark b running on a reference machine. The sample throughput is computed from the W per-workload throughput numbers:

$$T = \underset{w \in [1,W]}{\text{X-mean}} \; t(w) \tag{2.9}$$

A metric equivalent to the IPCT can be obtained by setting X-mean to A-mean and $IPC_{ref}[b]$ to 1. WSU and HSU are obtained by setting X-mean to A-mean and H-mean respectively; and for $IPC_{ref}[b]$ we use the IPC for the benchmark running alone on the reference machine (*single-thread IPC*).

Chapter 3

Behavioral Core Models

3.1 Introduction

Modern high-performance processors have a very complex behavior which reflects the complexity of the microarchitecture and of the applications running on it. Models are necessary to understand this behavior and take decisions.

Various sorts of models are used at different stages of the development of a processor, and for different purposes. For instance, analytical models are generally used for gaining insight. Fast performance models are useful in research studies and, in early development stages, for comparing various options. As we take decisions and restrict the exploration to fewer points in the design space, models become more detailed. In general, there is a tradeoff between accuracy and simplicity. A “heavy" model, e.g., a RTL description, gives accurate performance numbers, but requires a lot of work and is not appropriate for research and design space exploration. A “light" model, e.g., a trace-driven performance simulator, can be used for research and exploration but provides approximate numbers. Moreover, it is possible to use different levels of detail for different parts of the microarchitecture, depending on where we focus our attention.

In this study, what we call an application-dependent core model, or *core model* for short, is an *approximate* model of a superscalar core (including the level-1 caches) that *can* be connected to a detailed *uncore* model, where the *uncore* is everything that is not in the superscalar core (memory hierarchy including the L2 cache and beyond, communication network between cores in a multicore chip, etc.) It must be emphasized that a core model is not a complete processor model [49, 57, 62, 45]. A complete processor model provides a global performance number, while a core model emits requests to the *uncore* (e.g., level-1 cache miss requests) and receives responses to its requests from the *uncore*. The request latency may impact the emission time of future requests. The primary goal of a core model is to allow reasonably fast simulations for studies where the focus is not on the core itself, in particular studies concerning the *uncore*.

Core models may be divided in two categories: structural models and behavioral

models. Structural core models try to emulate the *internal* behavior of the core microarchitecture. Simulation speedups in this case come from not modeling all the internal mechanisms but only the ones that are supposed to most impact performance.

Behavioral core models try to emulate the *external* behavior of the core, which is mostly viewed as a black box. Unlike structural models, behavioral models are derived from detailed simulations, which is a disadvantage in some cases. But in situations where model building time can be amortized, behavioral core models are potentially faster and more accurate than structural models. Yet, behavioral core models have received little attention so far.

To the best of our knowledge, the work by Lee et al. is the only previous study that has focused specifically on behavioral superscalar core modeling [61]. They found that behavioral core models could bring important simulation speedups with a reasonably good accuracy. However the detailed simulator that they used, SimpleScalar *sim-outorder* [3], does not model precisely all the mechanisms of a modern superscalar processor. We present in this chapter an evaluation of Lee et al.'s Pairwise Dependent Cache Miss (PDCM) core modeling method using the Zesto detailed simulator, a detailed model of a modern superscalar microarchitecture [64]. We implemented a core model based on the PDCM approach with a reasonably good accuracy. Still, we identified some opportunities to improve the accuracy.

This has led us to propose a new method for behavioral application-dependent superscalar core modeling, BADCO, inspired by but different from PDCM. Unlike PDCM, which uses a single detailed simulation to build the core model, BADCO uses two detailed simulations. The first detailed simulation, identical to the one performed for PDCM, provides timing information for μops when all level-1 (L1) miss requests have a null latency. For the second information, we force a long latency on all L1 miss requests. Unlike PDCM, which uses a structural approach to find the dependences between requests, BADCO infers dependences from the timing information provided by the second detailed simulation.

The accuracy of BADCO is on average better than that of PDCM on all the configurations we have tested. We have studied not only the ability of BADCO to predict raw performance but also its ability to predict how performance changes when we change the *uncore*. Our experiments demonstrate a good qualitative accuracy of BADCO, which is important for design space exploration. The simulation speedups we obtained for PDCM and BADCO are in the same ranges, typically between one and two orders of magnitude.

This Chapter is organized as follows. Section 3.2 illustrates the limits of approximate core modeling. Section 3.3 briefly describes PDCM and the extension we introduce into the model to improve its accuracy for detailed superscalar cores. We introduce a new behavioral core model, BADCO, in Section 3.4. Section 3.5 presents an experimental evaluation of the accuracy and simulation speed of PDCM and BADCO. Finally, Section 3.6 evaluates BADCO's accuracy and simulation speed for multiprogram workloads.

3.2 The limits of approximate microarchitecture modeling

The curves on Figure 3.1 demonstrate the complex behavior of an out-of-order (OoO) superscalar core. These curves, one for *h264ref* and one for *libquantum*, were obtained with the Zesto simulator [64] and show the normalized execution time as a function of the L1 miss latency, assuming that the miss latency is uniform and constant. One would expect these curves to be monotonically increasing and convex.

Let us assume that the execution of a program by a superscalar processor can be modeled as a graph, where nodes represents certain events and edges represent dependences between events [34, 35]. Each edge is annotated with a latency. Let us assume that requests to the *uncore* are a subset of the graph edges, and that all the requests have the same latency X. We enumerate all the possible paths (i.e., dependence chains) in the graph and denote N_k the number of requests on path k. The length of path k is

$$T_k(X) = L_k + N_k X$$

and the total execution time is the length of the longest path

$$T(X) = \max_k T_k(X) = T_{p(X)}(X)$$

where $p(X)$ is the longest path. $N_{p(X)}$ is the slope of $T(X)$ at X. Let us consider $X < Y$. We have

$$T_{p(Y)}(X) \leq T(X)$$
$$T_{p(X)}(Y) \leq T(Y)$$

This implies $(N_{p(Y)} - N_{p(X)})X \leq (N_{p(Y)} - N_{p(X)})Y$, which is possible only if $N_{p(Y)} \geq N_{p(X)}$. The slope of $T(X)$ increases with X, hence $T(X)$ is convex.

As the miss latency is increased, there should be more and more misses on the critical path (the chain of dependent events that determines the overall execution time [34]). The curve for *h264ref* is nearly convex, as are the curves for a majority of our benchmarks. However, some benchmarks like *libquantum* have a clearly non-convex curve. This shows that the critical path, though a convenient conceptual tool, does not reflect completely what happens in a OoO microarchitecture. This illustrates the inherent difficulty of defining approximate microarchitecture performance models. The behavior of an OoO core depends on many mechanisms interacting in a complex way and impacting performance. Such complex behavior is difficult to reproduce with a simplified model, be it structural or behavioral. With this limitation in mind, the aim of approximate microarchitecture modeling is to find a good trade-off between simulation accuracy and simulation speed.

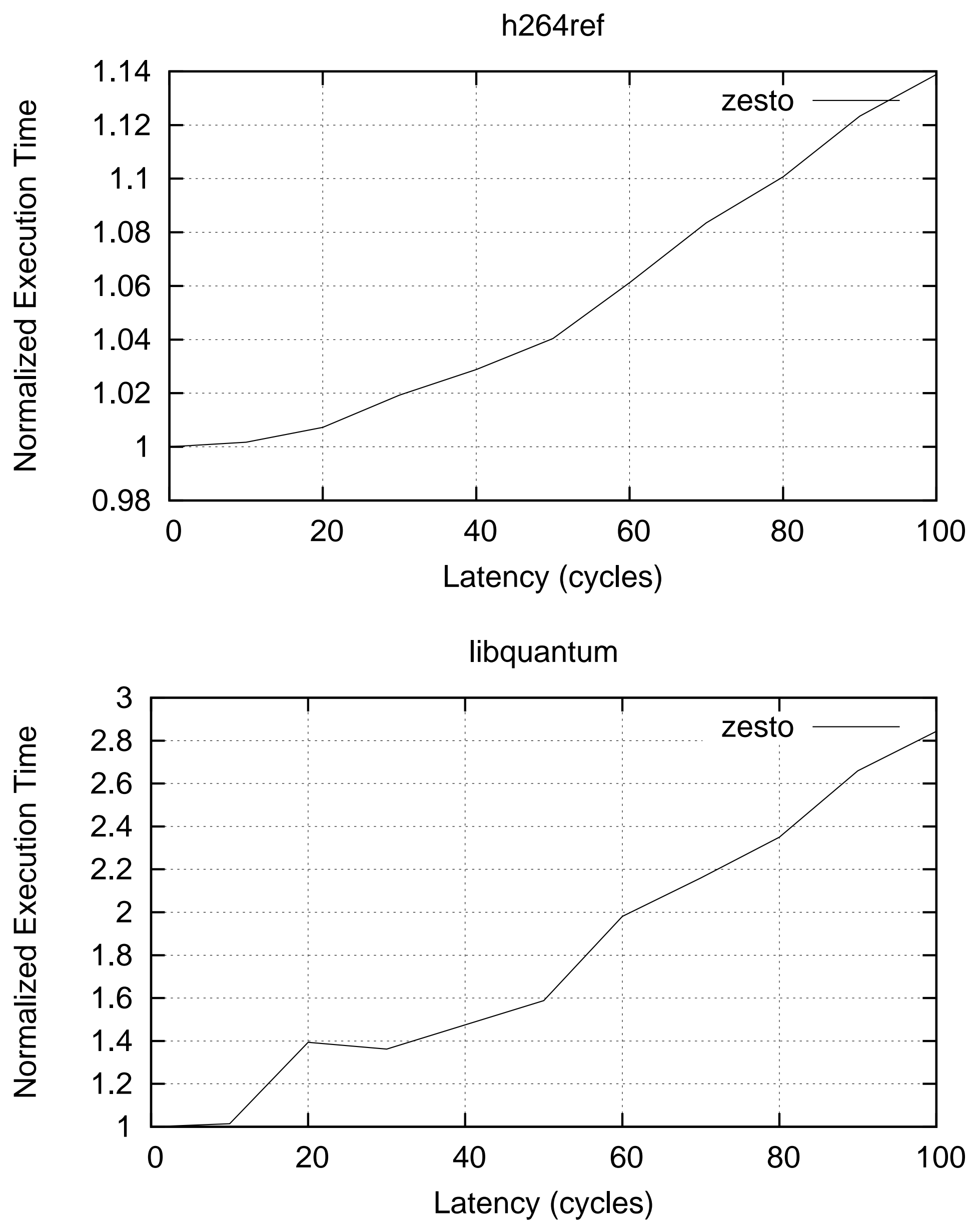


Figure 3.1: Normalized execution time for *h264ref* and *libquantum* as a function of the L1 miss latency, assuming a constant and uniform miss latency, using the Zesto simulator.

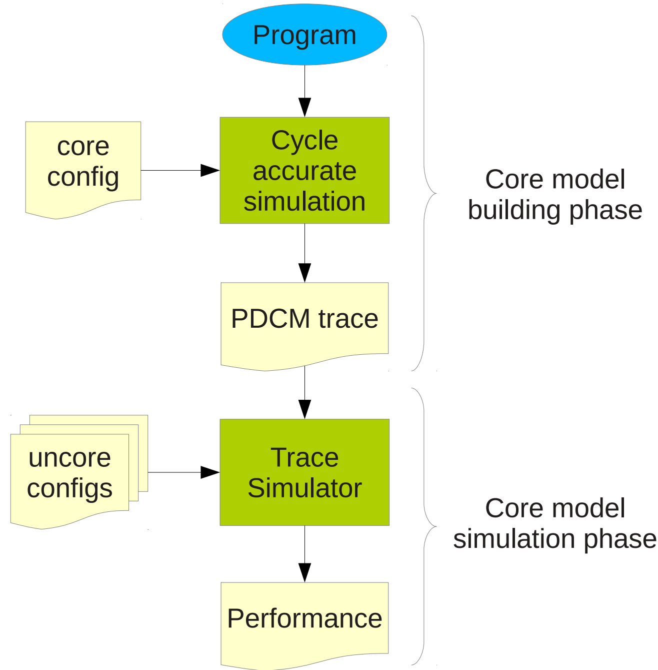


Figure 3.2: Simulation flow for PDCM behavioral core model.

3.3 The PDCM behavioral core model

Lee et al. present in [61] three different behavioral models: isolated cache miss, independent cache miss, and pairwise dependent cache miss. The models use traces of L2 accesses annotated with additional information to approximate the behavior of an OoO core. The isolated cache miss model is a pessimistic approach, where all trace-items are processed in a sequential way, that leads to an overestimation of the execution time. The independent cache miss model uses the ROB to control the number of items that can access memory. This model assumes total independence among trace-items. The performance results highly underestimate the cycle count for benchmarks with many dependencies between memory accesses. Finally, the pairwise dependent cache miss model (PDCM) improves over the independent cache miss model by considering dependencies between items. Lee et al. show that PDCM is accurate for an idealistic OoO core with perfect branch prediction and no hardware prefetch. In this section, we study the PDCM model and how it performs with a detailed OoO core model. We also propose some improvements to increase PDCM's accuracy.

3.3.1 PDCM simulation flow

The simulation flow of PDCM behavioral core model has two phases: model building and trace simulation. Figure 3.2 presents the simulation flow for PDCM. During the model building phase, a per-application trace is generated from a detailed microarchitecture simulator [1], assuming an ideal L2 cache, i.e., forcing an L2 cache hit on each L1 cache miss. Each trace item represents an instruction with an L1 miss. The trace item information contains **(1)** the request type (read, write, instruction, etc.); **(2)** the *instruction delta*, i.e., the number of instructions between this L1 miss and the next L1 miss; **(3)** the *time delta*, i.e., the number of cycles elapsed between this L1 miss and the next L1 miss; and **(4)** a *data dependence*, i.e., on which previous L1 miss this L1 miss depends, directly or indirectly. This data dependence is found by analyzing register and memory dependences during trace generation, taking into account the indirect dependences caused by delayed L1 hits [2].

During the trace-driven simulation, the time deltas and the dependences are used to compute the issue time of each L1 miss. Dependences include both data dependences and structural dependences induced by the limited reorder buffer (ROB). In particular, the instruction deltas are used to simulate the effect of the limited ROB and determine whether or not independent L1 misses can overlap.

3.3.2 Adapting PDCM for detailed OoO core

The original PDCM was tested with SimpleScalar sim-outorder microarchitecture model assuming 100% correct branch predictions [61]. Zesto is more detailed than sim-outorder, and we had to spend substantial effort adapting PDCM for Zesto in order to improve the accuracy until similar level to that described in [61]. Figure 3.3 illustrates our efforts. The first bar (leftmost) shows the accuracy obtained with our initial implementation of PDCM, based on what is explicitly described in the original PDCM work, taking into account the limited MSHRs and assuming a perfect branch prediction. The second bar shows the impact of having a realistic branch predictor and activating hardware prefetchers: unsurprisingly, the accuracy degrades. Then we improved the accuracy, keeping the general principles of the PDCM approach: we have introduced in the model TLB misses (third bar), write backs (fourth bar), wrong-path L1 misses, L1 prefetch requests (sixth bar), and more precise modeling of delayed hits (last bar). The numbers shown for PDCM in the remaining of this chapter were obtained with our optimized version. Here, we present a fully-detailed description of our improvements to PDCM.

[1] Lee et al. used Simplescalar sim-outorder [3] for their experiments.

[2] If an L1 miss Y is data-dependent on a delayed L1 hit which is waiting for a cache line requested by a previous L1 miss X, then Y is considered data-dependent on X [12].

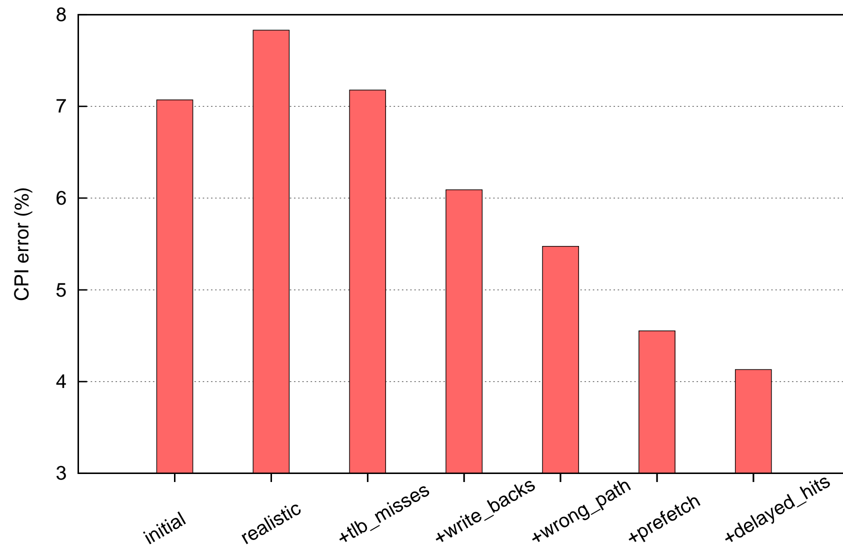


Figure 3.3: Our efforts to adapt the PDCM method to the Zesto microarchitecture model and decrease the average CPI error.

3.3.2.1 TLB misses and inter-request dependencies

The original PDCM considers only three types of L1 misses: instruction, load and store misses. Computer systems that support virtual memory use translation lookahead buffers (TLBs) to translate virtual addresses into physical addresses. TLBs are small caches with the only function of helping in the translation of virtual addresses. Modern processors include at least two TLBs: one for instructions and one for data. TLB misses introduce an extra delay on memory accesses. Additionally, if a memory access generates both TLB and L1 cache misses, then the TLB miss must be resolve before the L1 miss start being processed. A similar dependency exists between instructions misses and data misses. It occurs when the instruction misses during fetch, and later the instruction itself generates a memory access that also misses. An extreme case happens when the same instruction misses during fetch in both instruction TLB and instruction L1 cache, and misses again in both data TLB and data L1 cache. In this case, the latency of the four misses is serialized. In this context, a PDCM model must be able to reproduce this behavior during trace simulation. Hence, we have extended trace items to contain multiple requests, which may have sequential dependencies among them. During trace simulation, we reproduce the dependencies between requests associated with the same trace item.

3.3.2.2 Write-backs

Another type of memory request not considered originally by PDCM are the write-backs (WB). A WB occurs when a dirty cache-block is selected as victim to be replaced by another incoming block. The evicted block is inserted into the WB-buffer and waits there until it can be written to the next level in the memory hierarchy. A WB generally does not have an immediate impact on the program performance, but it consumes bandwidth and increases the latency of other L1 misses when the WB-buffer is full. To include WBs in PDCM we must associate this type of memory request with trace items. Hence, during trace generation we attribute a WB to the trace item whose L1 data miss causes the cache-block eviction. During trace simulation WB are issued to the *uncore* after the associated L1 data miss completes.

3.3.2.3 Branch miss predictions

Every modern superscalar core uses branch prediction to reduce the penalty that branch instructions cause, when executed in long pipelines. If a branch misprediction occurs, then the core must roll back to the mispredicted branch and flush all the instructions in the wrong path. However, during the execution of the wrong path, the core may initiate all kind of *uncore* requests [3]. The impact on performance of wrong-path requests has already been studied [73, 85]. Some mispredicted request behave as prefetch requests, and bring blocks to the cache that will be used in the future, contributing in this way to improve performance. However, wrong-path requests consume bandwidth through the memory hierarchy, and they may pollute the caches. Besides, wrong-path requests may also initiate additional WBs. An extra complexity is the fact that the number of wrong-path requests may change if the outcome of a branch depends on a long latency request.

Our experimental results show that omitting wrong-path requests lead to an underestimation of the cycle count. In order to improve the accuracy, we must capture and trace wrong-path requests. Hence, a mispredicted branch generates a trace item to which all the requests on the wrong path are attributed. During trace simulation, a mispredicted-branch item will stall the fetch of new items until it completes execution and all the wrong-path requests have been issued to the *uncore* [4].

3.3.2.4 Prefetching

The purpose of a hardware prefetcher is to fetch cache blocks before they are needed by the program. The prefetcher uses the stream of memory access/misses to predict which blocks will be needed in the future. If the predicted block is not already present

[3]The only exception is L1 store miss, because store request are processed after the commit pipeline stage

[4]The fetch stall models the pipeline flush on real architectures.

in cache, and the cache and bus are not too busy [5], then a prefetch request is issued to the cache. Not all prefetch requests hide completely the latency of memory accesses. Hence, it may happen that demand misses become delayed hits on a prefetch. Prefetch requests account for an important percentage of the traffic through the memory hierarchy. Moreover, prefetch requests may pollute the cache, and they also initiate additional WB requests.

Considering the effect of prefetch requests on a core model such as PDCM is complex. On one side, a prefetch request depends on the stream of access/misses on a L1 cache. On the other side, a prefetch request also depends on the performance of the *uncore*. Furthermore, delayed hits pending on a prefetch requests impact performance. Ignoring the impact of prefetch on performance request may lead to misestimate the cycle count.

Experimentally, we have observed that the total number of read requests (demand misses + prefetch) to the *uncore* does not change significantly from one *uncore* configuration to another. What we have is an exchange of L1 misses for prefetch requests and vice versa. During trace generation, we record all prefetch requests, and thus we guarantee that during trace-driven simulation, PDCM issues a similar number of read requests to the *uncore*. However, a request that for PDCM trace-driven simulation is a prefetch may be a demand miss on a corresponding detailed simulation, or the other way around. The impact on performance in any cases is different and thus a source of inaccuracy.

During trace generation, prefetch requests are attributed to the instruction that triggered the prefetch. In particular, we have configured Zesto to generate L1 prefetch requests on a miss. Therefore, a prefetch request is attributed to the same item as the L1 demand miss. During trace simulation, prefetch requests are issued to the *uncore* simultaneously with demand misses. However, a trace item does not need to wait for the prefetch request to return to be completed.

3.3.2.5 Delayed hits

A delayed hit is a memory reference to a cache block for which a request has already been initiated by another instruction but has not yet completed, i.e., the requested block is still on its way from memory [12]. Delayed hits were considered by Lee et al. in the original PDCM model as an instrument to account for indirect dependences caused by delayed L1 hits. The same problem has been addressed by Chen et al. in [12]. In the context of a limited number of overlapping long latency data cache misses due to finite MSHR resources, delayed hits have an additional impact on performance that must be addressed. In particular, Zesto models the MSHR in such a way that each L1 cache miss occupies an MSHR entry. As a result, delayed hits also occupy MSHR entries, and thus they can limit the effective number of outstanding requests that can be processed

[5] The MSHR occupancy and the bus utilization, for example, can be monitored by the prefetch controller to decide whether or not to issue prefetch requests.

simultaneously. We have found this problem to be extremely important when modeling the performance of benchmarks such as libquantum and hmmer.

One limitation of PDCM is that the trace is generated assuming an ideal L2 cache, thus it does not capture all delayed hits that may be present when long latencies are simulated. In order to overcome this problem, during trace generation, we search for additional load or store instructions, that in the case of a long latency access, would have been delayed hits. During trace simulation, the information about delayed hits is used in conjunction with the number of requests already in flight and the total number of MSHR entries to limit the number of outstanding requests.

3.3.3 PDCM limitations

It should be noted that PDCM is a behavioral model as the time deltas are obtained from a detailed microarchitecture simulation. Because the time deltas correspond to an ideal L2 cache, PDCM is very accurate when L2 misses are few. However, PDCM uses a structural approach to model the impact of L2 misses: it is assumed that modeling the effect of the ROB and data dependences is sufficient to reproduce accurately the performance impact of L2 misses. Yet, core resources other than the ROB may impact performance significantly, for instance the limited number of ALUs, L1 cache ports, reservation stations, etc. Even considering an unlimited ROB, the time deltas between consecutive and data-independent L1 misses may depend on the miss latency, e.g., because of resource conflicts happening differently (the miss latency may impact the order in which instructions are executed and how many times instructions are rescheduled), or because a mispredicted branch is data-dependent on an L1 miss. In this context, Section 3.4 present a new behavioral core model that is inspired from the PDCM model but tries to overcome its limitations.

3.4 BADCO: a new behavioral core model

The new behavioral model we propose, BADCO, is inspired from PDCM. However BADCO uses a behavioral method to find dependences between requests to the *uncore*, unlike in PDCM where an explicit data-dependence analysis is performed. Unlike PDCM which uses a single detailed simulation to build the core model, BADCO uses two detailed simulations.

For the first detailed simulation, we force the latency of each request to zero. This simulation is identical to the one done for PDCM. From this first simulation, we obtain a trace T0. Then we perform a second simulation by giving a long latency to each request. We set the request latency to a value greater than or equal to L, where L is typically greater than the greatest latency that may be experienced when using the core model, e.g., $L = 1000$ cycles. We give to certain requests a latency greater than L: we set the latencies so as to force the completion times of successive data requests to be

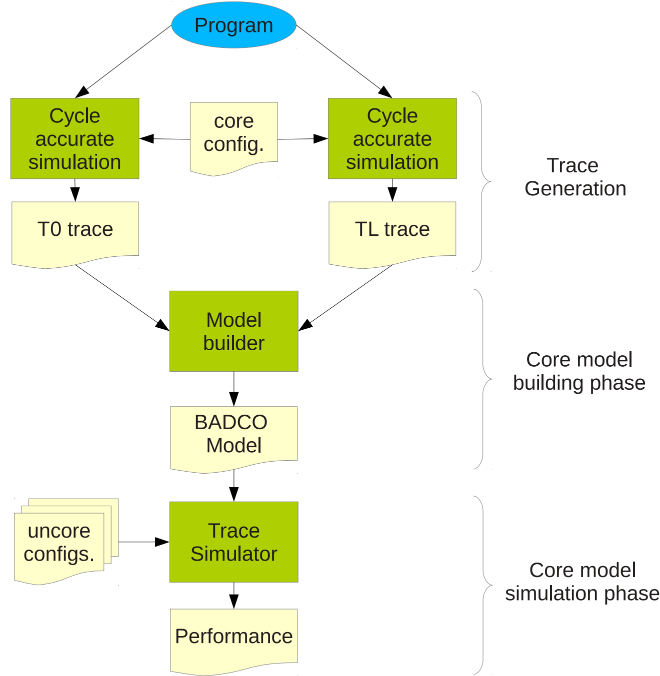


Figure 3.4: Simulation flow for BADCO model.

separated by L cycles or more. We obtain from this second simulation a trace TL. Both T0 and TL contain some timing information for each retired μop.

A BADCO model is then built from the information contained in T0 and TL. The information in TL is used to find (direct and indirect) dependences between requests. Dependences include not only data dependences, but also branch mispredictions, limited resources (reservation stations, MSHRs, ...), etc. We do not perform any detailed analysis of these dependences during trace generation. Instead, dependences are found indirectly by analyzing the timing information in TL. We use the fact that, if a request R2 is issued before a previous request R1 is completed, R2 does not depend on R1. If R2 depends only on R1, R2 is often issued a few cycles after R1 completes. That is basically how we detect dependences. Forcing successive requests in TL to occur at intervals no less than 1000 cycles is for disambiguation: R1 is the request whose completion time is closest to the issue time of R2. Of course, this method is not 100 % reliable, but it works well in practice.

3.4.1 The BADCO machine

A BADCO machine is an abstract core that fetches and executes *nodes*. A node N_i represents a certain number S_i of retired μops (not necessarily contiguous in sequential order). S_i is the node *size*. The sum of all nodes sizes, $\sum_i S_i$, is equal to the total

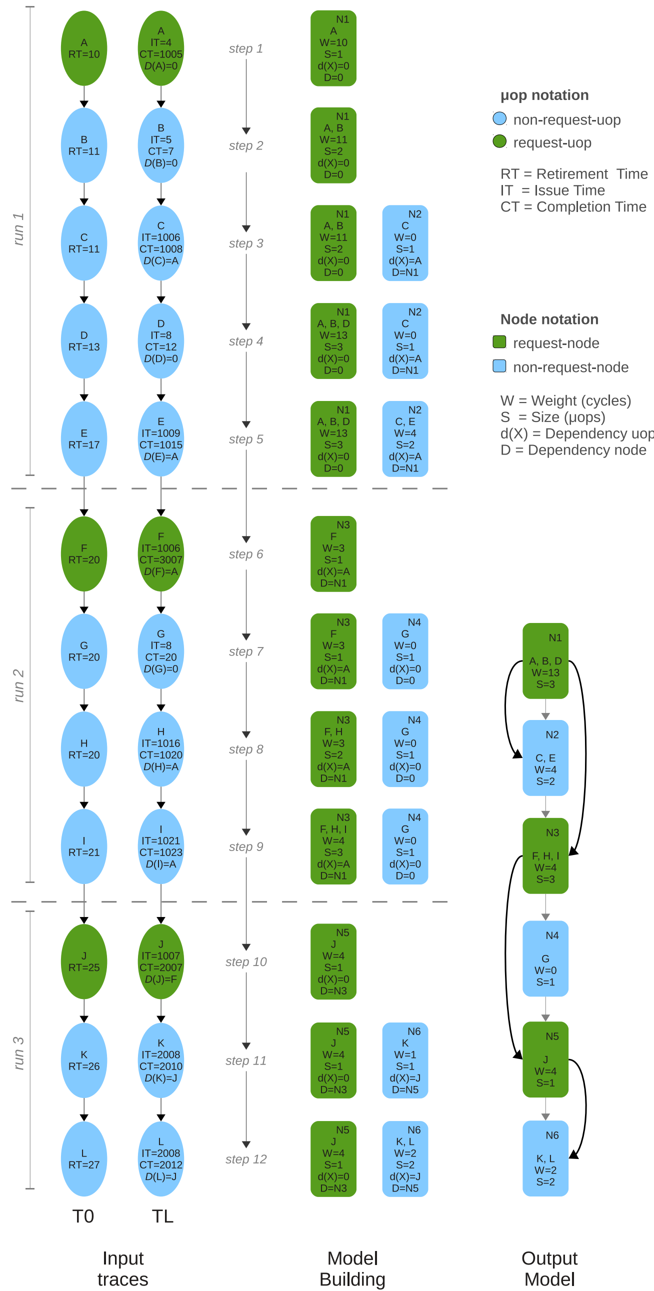


Figure 3.5: Example of BADCO model building: Input traces T0 and TL containing the same 12 dynamic μops in sequential order at the left, μop-by-μop processing of the traces at the center, and the final BADCO model featuring 6 nodes at the right.

number of μops executed. As the BADCO machine works on nodes instead of μops, the bigger the nodes, the greater the expected simulation speedup. The next section explains how we build the nodes. A node N_i also has a certain latency in clock cycles, called the node *weight* W_i.

Some nodes, called *request nodes*, carry one or several requests to the *uncore*. There are three sorts of request nodes: I-nodes, L-nodes and S-nodes. An I-node may carry three sorts of requests: IL1 miss, ITLB miss or instruction prefetch requests. An L-node (or S-node) carries the requests attached to one load (or store) μop (DL1 miss, DTLB miss, write-back, DL1 prefetch [6]). An L-node or S-node can also be an I-node. In the BADCO model, a node may be dependent on one older request node, called the *dependency node*.

During the trace-driven simulation, the BADCO machine fetches nodes and inserts them in the BADCO *window* in sequential order. I-nodes send their requests to the *uncore* at fetch time. Node fetching imitates what the real core does [7]. The BADCO window emulates the real core reorder buffer (ROB). When the sum of nodes sizes inside the window does not exceed the ROB size, the next node can be fetched. Otherwise node fetching is stalled. Once in the window, nodes can start executing. An L-node may send its requests as soon as its dependency node is completed. An L-node is considered completed when all its requests are finished. Other nodes are considered completed when their dependency node is completed. Nodes are retired from the window in the order they were fetched. A node is ready for retirement when it is completed and it is the oldest node in the window. The retirement of a node N_i from the window actually happens exactly W_i cycles after the node is ready for retirement. After being retired from the window, an S-node is sent to a post-retirement store queue, imitating what the real core does with stores. The requests carried by an S-node are issued to the *uncore* after retirement. The BADCO machine models the occupancy of the MSHRs inside the core. It imitates, to the extent possible, how the real core manages the MSHR [8]. In particular, a request requiring an MSHR entry must wait until there is a free MSHR entry before being sent to the *uncore*.

3.4.2 BADCO model building

The BADCO model building phase consists in grouping μops with the same dependencies to form nodes, and in defining dependencies between nodes. Traces T0 and TL provide the information for this process.

[6] We attach a DL1 miss request to the first μop (load or store) accessing that cache line. We attach a DL1 prefetch to the μop triggering the prefetch. We attach a write-back request to the same μop to which the request causing the write-back is attached.

[7] The Zesto model implements next-line prefetching for the instructions, but does not pipeline the instruction misses. Node fetching mimics this behavior.

[8] For instance, the Zesto simulator allocates an MSHR entry for each delayed hit (i.e., hits on a pending miss). Our BADCO machine does the same in our experiments. This is why we simulate an unlimited MSHR for generating trace TL, so as to capture all potential delayed hits in the trace.

Both traces T0 and TL in the left part of Figure 3.5 represent the same sequence of dynamic μops in program order. The μops in T0 are annotated with their retirement time *"RT"*. The μops in TL are annotated with their issue time *"IT"* and completion time *"CT"*. Some μops carry one or several requests, they are called *request* μops[9]. All other μops are called *non-request* μops. Figure 3.5 uses dark-gray/green circles for request μops and light-gray/blue circles for non-request μops. A request μop and the non-request μops following it until the next request μop form a *run*.

For each μop X, we define its *dependency* μop D(X) as follows: D(X) is the request μop before X and closest to X whose CT is less than the IT of X. [10]. For example, μop H in Figure 3.5 has $IT = 1016$, the closest request μop with $CT < 1016$ is μop A with $CT = 1005$, then $D(H) = A$.

We process traces T0 and TL simultaneously and μop by μop, in lockstep fashion. For each μop, we determine if the μop starts a new node or if it is attributed to an existing node. Every request μop X starting a *run* creates a new node N_j to which it is attributed. The dependency node $D(N_j)$ of N_j is the node to which D(X) has been attributed. *All subsequent μops attributed to the same node must have the same dependency μop.* In particular, all the μops in the *run* with the same dependency X are attributed to node N_j. If a non-request μop cannot be attributed to any of the nodes already created for that *run*, we create a new node for the μop.

Attributing a μop to a node N_i means incrementing the node size S_i and adding to the node weight W_i the difference between the retirement time of the μop in T0 and that of the previous μop. By doing so, the sum of all nodes weights, $\sum_i W_i$, equals the total execution time when all the requests to the *uncore* have a null latency.

The central part of Figure 3.5 presents step by step the building process of nodes. Step 1 processes μop A; A is a request μop and starts the node N1 with $W = 10$, $S = 1$, and $D(N1) = 0$. Step 2 processes μop B; B is a non-request μop with $D(B) = 0$, and as consequence, it is attributed to N1 with $D(N1) = 0$. The properties of N1 are updated, the size S is incremented, and 1 cycle is added to the weight W because $RT_B - RT_A = 1$. In Step 3, we start a new node N2 for the non-request μop C with $W = RT_C - RT_B = 0$, $S = 1$ and $D(N2) = N1$ (A attributed to N1). The μop C cannot be attributed to the node N1 because all μops in N1 have a null dependency and C depends on A. Steps 5 and 6 attribute μops D and E to nodes N1 and N2 respectively. Step 6 processes the request μop F and starts the processing of the second *run* of μops. We create a new node N3 with $W = RT_F - RT_E = 3$, $S = 1$ and $D(N3) = N1$ (A attributed to N1). Step 7 processes the non-request μop G; G starts a new node N4 because $D(G) = 0$ and cannot be attributed to N3. Note that G cannot be attributed to N1 either because N1 belongs to the previous *run*. The building process continues in a similar fashion for the subsequents μops. The right part of Figure 3.5 presents the final BADCO model.

[9] Each request to the *uncore* is attached to a single μop.

[10] D(X) is null or 0 when there is not request μop whose CT is less than the IT of X. This just happen at the beginning of the trace.

core type	**small**	**medium**	**big**
decode/issue/commit	3/4/3	3/5/3	4/6/4
RS/LDQ/STQ/ROB	12/12/8/32	18/18/12/64	36/36/24/128
DL1/DTLB MSHRs	4/2	8/4	16/8

clock	3 GHz
IL1 cache	2 cycles, 32 kB, 4-way, 64-byte line, LRU, next-line prefetcher
ITLB	2 cycles, 128-entry, 4-way, LRU, 4 kB page
DL1 cache	2 cycles, 32 kB, 8-way, 64-byte line, LRU, write-back, IP-based stride + next line prefetchers
DTLB	2 cycles, 512-entry, 4-way, LRU, 4 kB page
Branch predictor	TAGE 4 kB, BTAC 7.5 kB, indirect branch predictor 2 kB, RAS 16 entries

Table 3.1: Core configurations. The default configuration is the “big" core.

3.5 Experimental evaluation

The detailed simulator used for this experiment is Zesto [64]. Some of the characteristics of the core and *uncore* configurations we consider are given in tables 3.1 and 3.2 respectively. We consider 3 different core configurations: “small”, “medium" and “big". The L2, LLC and memory bus each can have a low or high value. This defines up to 8 different *uncore* configurations. For instance, the configuration denoted “010" has a small L2, a big LLC, and a narrow memory bus. The “big" core is the default core configuration. The default *uncore* configuration is “001". We will not present results for configurations “100" and “101" since they are not realistic.

For generating traces T0 and TL, we skip the first 40 billions instructions of each benchmark, and the trace represents the next 100 millions instructions (no cache warming was performed). We assume that simulations are reproducible, so that T0 and TL represent exactly the same sequence of dynamic μops. We used SimpleScalar EIO tracing feature [3], which is included in the Zesto simulation package. We present results for the SPEC CPU2006 benchmarks that we are able to run with Zesto. We have also included two SPEC CPU2000 benchmarks, *vortex* and *crafty*. We have chosen these two benchmarks because they experience a relatively high number of instruction misses and branch mispredictions, which is interesting for testing the models. All the benchmarks were compiled with gcc-3.4 using the “-O3" optimization flag.

3.5.1 Metrics

The primary goal of behavioral core modeling is to allow fast simulations for studies where the focus is not on the core itself, in particular studies concerning the *uncore*.

	low (“0")	high (“1")
L2 size/latency	256 kB / 6 cycles	1 MB / 8 cycles
LLC size/latency	2 MB / 18 cycles	16 MB / 24 cycles
FSB width	2 bytes	8 bytes
DL1 write buffer	8 entries	
L2	64-byte line, 8-way, LRU, write-back, 8-entry write buffer, 16 MSHRs, IP-based stride + next line prefetchers	
LLC	64-byte line, 16-way, LRU, write-back, 8-entry write buffer, 16 MSHRs, IP-based stride + stream prefetchers	
FSB clock	800 MHz	
DRAM latency	200 cycles	

Table 3.2: Uncore configurations. The L2, LLC and memory bus each can have a low or high value, which defines up to 8 different configurations. For instance, the configuration denoted “010" has a small L2, a big LLC and a narrow memory bus. The default configuration is “001".

Ideally, a core model should strive for *quantitative* accuracy. That is, it should give *absolute* performance numbers as close as possible to the performance numbers obtained with detailed simulations. Nevertheless, perfect quantitative accuracy is difficult, if not impossible to achieve in general with a simple model.

Yet, *qualitative* accuracy is often sufficient for many purposes. Qualitative accuracy means that if we change a parameter in the *uncore* (i.e., memory latency), the model will predict accurately the *relative* change of performance. Indeed, if we use behavioral core modeling in a design space exploration for example, more important than being accurate in the final cycle count is being able to estimate relative changes in performance among the different configurations in the design space. Therefore we use several metrics to evaluate the PDCM and BADCO core models. The *CPI error* for a benchmark is defined as

$$\text{CPI error} = \frac{CPI_{ref} - CPI_{model}}{CPI_{ref}}$$

where CPI_{ref} is the CPI (cycles per instruction) for the detailed simulator Zesto, and CPI_{model} is the CPI for the behavioral core model (PDCM or BADCO). The CPI error may be positive or negative. The smaller the absolute value of the CPI error, the more *quantitatively* accurate the behavioral core model. The *average CPI error* is the arithmetic mean of the *absolute value* of the CPI error on our benchmark set.

For a fixed core, we define the relative performance variation **RPV** of an *uncore*

	“small”	**“medium”**	**“big”**
PDCM	3.8 %	4.0 %	4.7 %
BADCO	3.3 %	2.4 %	2.8 %

Table 3.3: Average CPI error of PDCM and BADCO respect to Zesto.

xyz as

$$RPV = \frac{CPI_{001} - CPI_{xyz}}{CPI_{001}}$$

where CPI_{001} is the CPI of the *uncore* configuration “001" and CPI_{xyz} is the CPI of *uncore* configuration xyz (see Table 3.2). The model *variation error* is defined as

$$\text{Variation error} = |RPV_{ref} - RPV_{model}|$$

where RPV_{ref} is the RPV as measured with the detailed core model and RPV_{model} is the RPV obtained with the behavioral core model (PDCM or BADCO). The smaller the variation error, the more *qualitatively* accurate the behavioral core model. When the variation error is null, it means that the behavioral core model predicts for *uncore* xyz the exact same performance variation relative to the reference *uncore* as the detailed core model. The *average variation error* is the arithmetic mean of the variation error on our benchmark set.

3.5.2 Quantitative accuracy

Figure 3.6 shows for each benchmark the CPI error of PDCM and BADCO for the “small”, “medium” and “big” cores, with the *uncore* configuration “001”. The maximum error is on *libquantum*, both for PDCM and BADCO and for the three core configurations. This is consistent with the non-convex curve of *libquantum* shown in Section 3.2, indicating an inherent modeling difficulty. Table 3.3 gives the average CPI error of PDCM and BADCO. BADCO is on average more accurate than PDCM for each of the three core configurations.

3.5.3 Qualitative accuracy

Figure 3.7 shows the Relative Performance Variation (RPV) of Zesto, PDCM and BADCO for the six *uncore* configurations “000”, “010”,“011”, “110” and “111” (see Table 3.2), assuming a “big” core. The baseline *uncore* is “001”.

Both PDCM and BADCO exhibit a reasonably good qualitative accuracy, i.e., they predict approximately how performance changes when we change the *uncore*. Neither PDCM nor BADCO are very good at predicting tiny performance changes (RPV of a few percents), but they are relatively good at predicting important performance changes. This makes PDCM and BADCO suitable for design space exploration, e.g., for selecting

(a) “big” core

(b) “medium” core

(c) “small” core

Figure 3.6: CPI error of PDCM and BADCO for the “small", “medium" and “big" cores, with the *uncore* configuration “001".

	"000"	**"010"**	**"011"**	**"110"**	**"111"**
PDCM	4.6 %	4.0 %	1.3 %	4.1 %	1.2 %
BADCO	2.6 %	2.2 %	0.7 %	2.5 %	0.8 %

Table 3.4: Average variation error using as reference the configuration "001".

	Zesto	**PDCM**	**BADCO**
with Zesto *uncore*	0.17	2.91	2.52
core alone	0.19	13.04	8.82

Table 3.5: Single core simulation speed in MIPS

some "interesting" *uncore* configuration for which more detailed simulations will be done. Table 3.4 gives the average variation error of PDCM and BADCO. BADCO is on average more accurate than PDCM for each of the 5 *uncore* configurations.

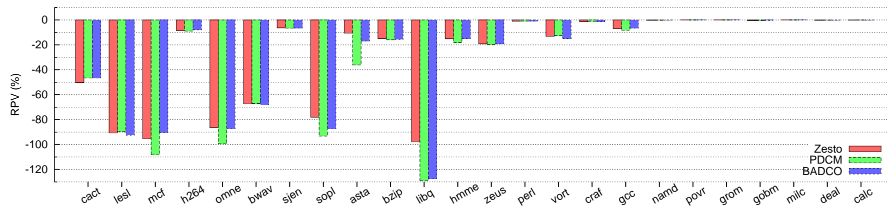
3.5.4 Simulation speed

We did all the simulation speed measurements on the same machine, which features an Intel Xeon W3550 (Nehalem microarchitecture, 8 MB L3 cache, 3.06 GHz) with Turbo Boost disabled and 6 GB of memory. All the simulation input files, including the traces for PDCM and BADCO, were stored on the local disk of that machine. Zesto, PDCM and BADCO were compiled with gcc-4.1 using the "-O3" optimization flag. We simulated the "big" core configuration and two different *uncore* configurations: one is the Zesto *uncore* configuration "001", the other is a simplistic *uncore* forcing all requests latencies to a null value. With the simplistic *uncore*, what we measure is essentially the simulation time for the core alone. Figure 3.8 shows the simulation time in millions of instructions simulated per second for Zesto, PDCM and BADCO.

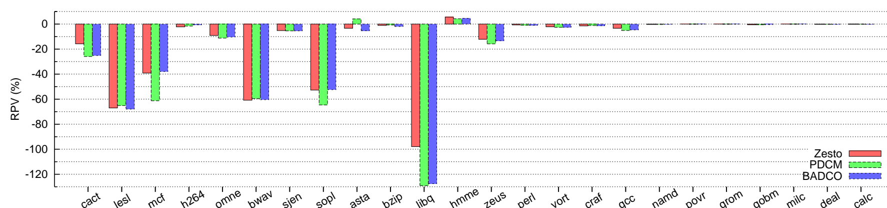
The simulation speedup achieved with PDCM or BADCO, in comparison with Zesto, is typically between one and two orders of magnitude. Benchmarks with the greatest speedups are the ones with the fewest L1 misses. The Table 3.5 gives the harmonic mean on our benchmarks of the simulation speed in millions of instructions simulated per second (MIPS).

PDCM is generally faster than BADCO because a BADCO nodes represents about 50 μops on average (harmonic mean on our benchmarks), whereas a PDCM trace item represents on average 90 μops. Hence PDCM works at a larger granularity.

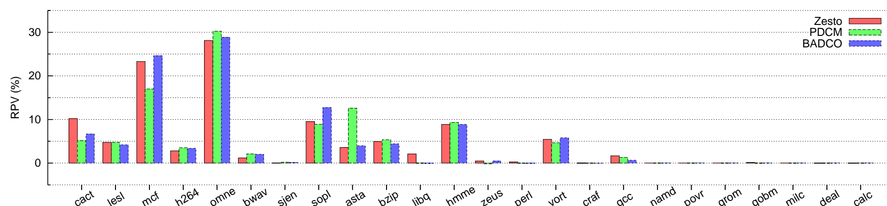
The PDCM and BADCO models we have implemented can be connected to a detailed *uncore* model. This means that the core does not know the request latency when it sends a request to the *uncore*. Hence the core model inspects each clock cycle in case an event occurs, which limits the simulation speedup.



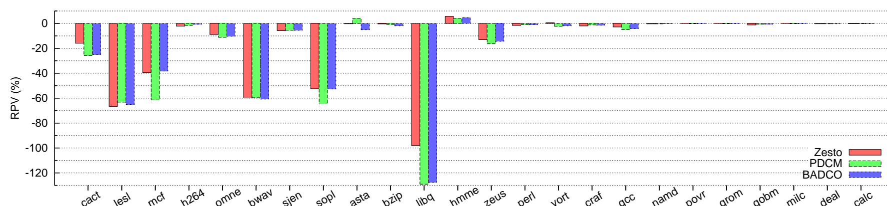
(a) *uncore* “000”



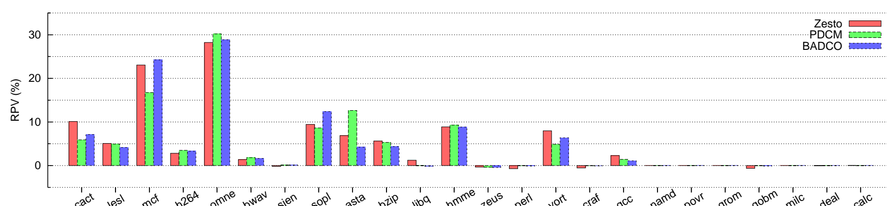
(b) *uncore* “010”



(c) *uncore* “011”

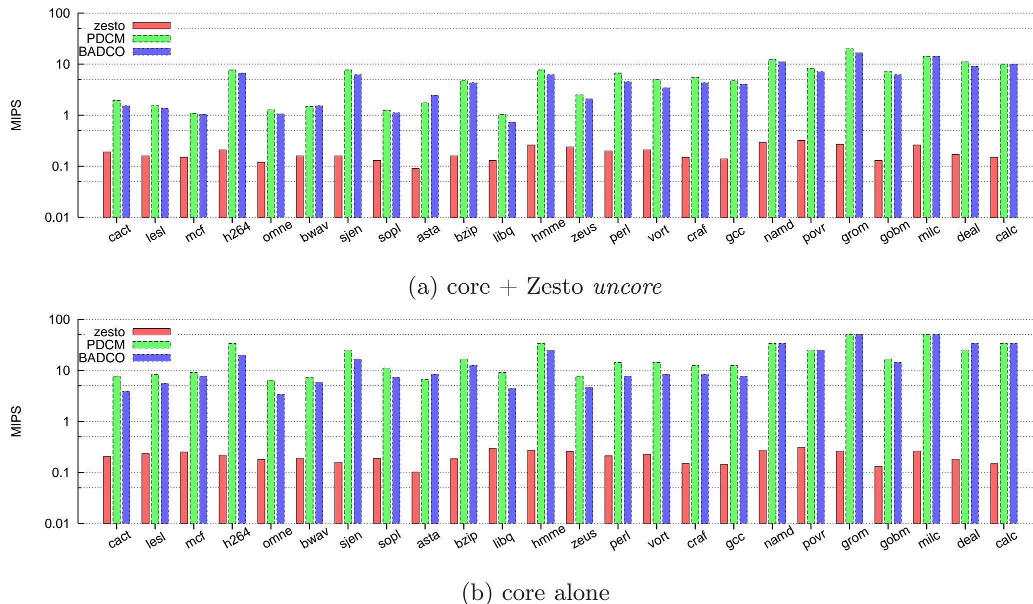


(d) *uncore* “110”



(e) *uncore* “111”

Figure 3.7: Relative performance variation (RPV) of Zesto, PDCM and BADCO for the *uncore* configurations “000”, “010”,“011”, “110" and “111”, assuming a “big" core. The baseline *uncore* is “001”.



(a) core + Zesto *uncore*

(b) core alone

Figure 3.8: Simulation speed in millions of instructions simulated per second (MIPS) with and without considering the impact of the Zesto *uncore* (logarithmic scale).

3.6 Modeling multicore architectures with BADCO

In recent years, research in microarchitecture has shifted from single-core to multicore processors. Cycle-accurate models for many-core processors featuring hundreds or even thousands of cores are out of reach for the simulation of realistic workloads. Approximate simulation methodologies that trade accuracy for simulation speed are necessary for conducting certain research, in particular for studying the impact of resource sharing between cores, where the shared resource can be caches, on-chip network, memory bus, power, temperature, etc.

Behavioral core models are one option to trade accuracy for simulation speed in situations where the focus of the study is not the core itself but what is outside the core, i.e., the *uncore*. In sections 3.3 and 3.4, we presented two behavioral core models: PDCM, a previously proposed core model that we have extended to model detailed superscalar processors; and BADCO, a new behavioral core model that is more accurate than PDCM. Both core models enable fast simulation of multicore architectures when the design target is the *uncore*. In this section we evaluate the speed and accuracy of BADCO when simulating multiprogram workloads for processor configuration of 2, 4 and 8 cores.

Extending BADCO to execute multiprogram workloads is straightforward. Once BADCO core models have been built for a set of single-thread benchmarks, the core models can be easily combined to simulate a multi-core running several independent

decode/issue/commit	4/6/4
RS/LDQ/STQ/ROB	36/36/24/128
DL1/DTLB MSHR entries	16/8
Clock	3 GHz
IL1 cache	2 cycles, 32 kB, 4-way, 64-byte line, LRU, next-line prefetcher
ITLB	2 cycles, 128-entry, 4-way, LRU, 4 kB page
DL1 cache	2 cycles, 32 kB, 8-way, 64-byte line, LRU, write-back, IP-based stride + next line prefetchers
DTLB	2 cycles, 512-entry, 4-way, LRU, 4 kB page
Branch predictor	TAGE 4 kB, BTAC 7.5 kB, indirect branch predictor 2 kB, RAS 16 entries

Table 3.6: Core configuration.

threads simultaneously. We connect several BADCO machines, one per core, to a detailed simulator of the *uncore*[11]. A BADCO machine communicates with the *uncore* by sending requests and receiving the acknowledge of the completion of those requests. BADCO machines send read and write requests to the *uncore*. A request indicates the type of transaction and the virtual memory address. The *uncore* simulator informs the BADCO machine when its requests have completed.

There is a round robin arbitration to decide which BADCO machine can access the *uncore*. When the *uncore* receives a request, it translates the virtual address to a physical address. If a page miss occurs, BADCO allocates a new physical page. Once this is done, the *uncore* processes the request. The *uncore* notifies the BADCO machine about the completion of one of its request through a call-back. A request completes when it has fully completed the processing through the memory hierarchy.

Analogously to Zesto, BADCO does not model physical page conflicts. In both Zesto and BADCO, main memory is assumed infinite. That means that every time that a page miss occurs, BADCO allocates a new physical page. The assignation of physical pages to virtual pages is made in a sequential fashion. During trace generation, BADCO traces save the request's virtual addresses to ease multicore simulation.

3.6.1 Experimental setup

Our experiments analyze the performance of multicore processors with 2, 4 and 8 identical cores. Table 3.6 presents a summary of cores characteristics. A case study with five *uncore* design points is evaluated, each design point corresponding to a different replacement policy in the shared last-level cache: LRU, RANDOM (RND), FIFO, DIP and DRRIP. Table 3.7 gives the *uncore* characteristics.

We build 250 random workloads from 22 of the 29 SPEC CPU2006 benchmarks

[11] The *uncore* simulator was extracted from Zesto.

Number of cores	**2**	**4**	**8**
LLC size/latency	1MB/5cyc.	2MB/6cyc.	4MB/7cyc.
DL1 write buffer	8 entries		
LLC	64-byte line, 16-way, write-back, 8-entry write buffer, 16 MSHR entries, IP-based stride + stream prefetchers		
FSB clock	800 MHz		
FSB width	8 bytes		
DRAM latency	200 cycles		

Table 3.7: Uncore configurations.

(the 22 benchmarks that we were able to simulate with Zesto). We perform detailed simulation with Zesto and trace-simulation with BADCO for every design point and every workload in the sample. Then, we compare BADCO's accuracy in terms of CPI error and speedup error using Zesto as reference. Finally, we measure the average simulation speed of both BADCO and Zesto.

All the benchmarks were compiled with gcc-3.4 using the "-O3" optimization flag. For generating BADCO traces, we skip the first 40 billions instructions of each benchmark, and the trace represents the next 100 millions instructions (no cache warming is performed). We assume that simulations are reproducible, so that traces represent exactly the same sequence of dynamic μops. We used SimpleScalar EIO tracing feature [3], which is included in the Zesto simulation package.

During multiprogram execution, each core runs a separate threads. When a thread has finished executing its 100 million instructions earlier than the other threads, it is restarted. This is done as many times as necessary until all the threads in the workloads have executed at least 100 million instructions. Performance is measured only for the first 100 million committed instructions of each thread.

3.6.2 Experimental results

Figure 3.9 reports the measured and the estimated CPIs for Zesto and BADCO respectively. Each dot in the graph represents the CPI performance of individual benchmarks in the 250 workloads and the five design points. A perfect estimation would imply that all the dots lie on the bisector. In this case, we observe that most of the points are over the bisector. This indicates that BADCO tends to slightly underestimate the CPI.

Table 3.8 presents the average of the *absolute CPI error* for 2, 4 and 8 cores and each design point. The global average of the *absolute CPI error* is 4.59 %, 3.98 % and 4.09 % for 2, 4 and 8 cores respectively. The maximum error is in all cases less than 25 %. Moreover, for approximate simulators, more important than predicting CPIs accurately is predicting speedups accurately. We compared the speedups predicted by BADCO and Zesto for replacement policies FIFO, RANDOM, DIP and DRRIP using LRU as

rep. policy	**2 cores**	**4 cores**	**8 cores**
LRU	4.66	3.83	3.90
RANDOM	4.63	4.19	4.46
FIFO	4.79	4.10	4.33
DIP	4.54	4.01	3.99
DRRIP	4.35	3.75	3.77

Table 3.8: Average of absolute CPI error in percentage for 2, 4 and 8 cores.

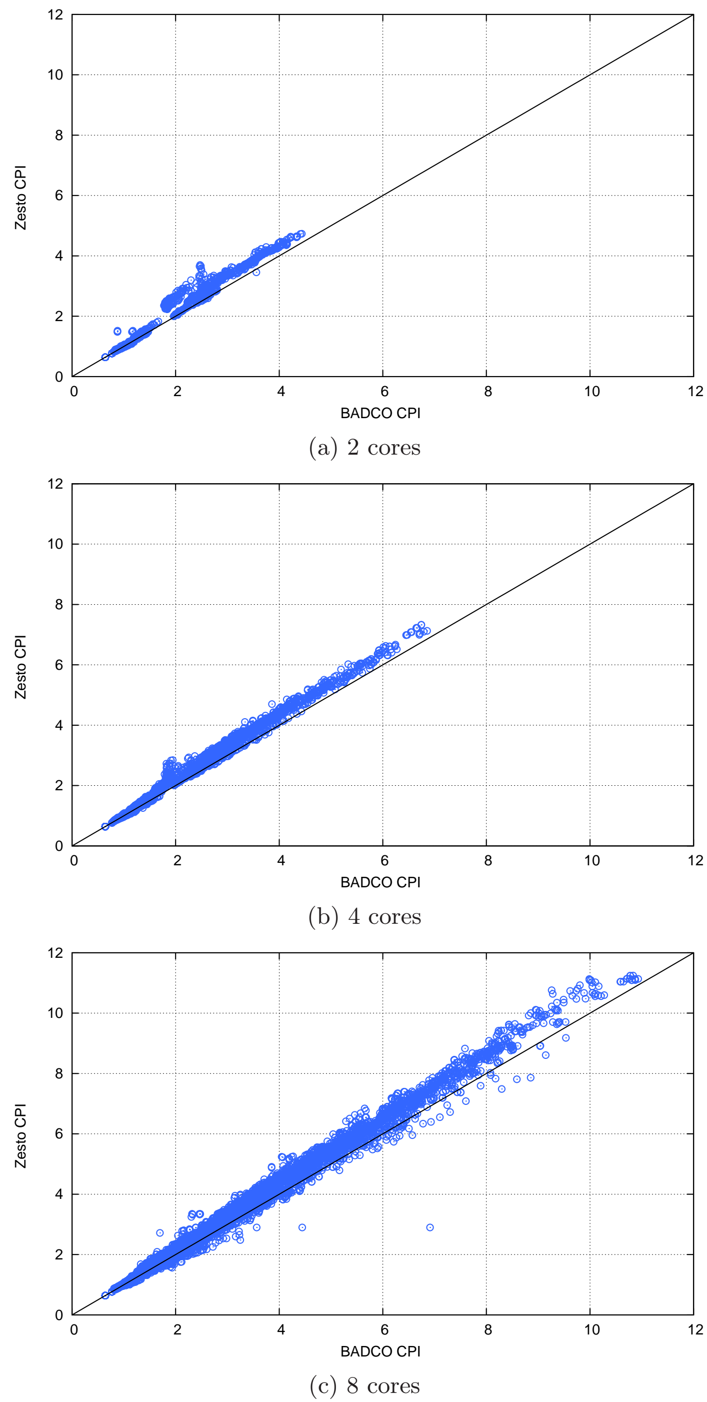
rep. policy	**2 cores**	**4 cores**	**8 cores**
RANDOM/LRU	0.89	0.76	1.34
FIFO/LRU	0.56	0.65	1.01
DIP/LRU	0.49	0.54	1.63
DRRIP/LRU	0.67	0.52	1.77

Table 3.9: Average of absolute *speedup error* in percentage for 2, 4 and 8 cores and LRU as reference.

reference. We found that, on average, the global speedup error is 0.66 % 0.61 % and 1.43 % for 2, 4 and 8 cores respectively. Table 3.9 presents the individual speedup errors for four design pairs that use LRU as reference. Results show that BADCO is notably better in predicting speedups than raw CPIs.

3.6.3 Multicore simulation speed

Table 3.10 reports the *simulator* performance of Zesto and BADCO. BADCO is clearly faster than the detailed simulator Zesto, with simulation speedups going from 15x to 68x when going from 1 to 8 cores. Zesto's decreases faster than BADCO's speed mainly because of memory management. Zesto must manage a memory space for each application. Such space reach typical sizes of 1GB or even more. When simulating many cores, Zesto does not have any other option but paging in order to keep running. BADCO does not have the same problem, and the decrease in simulation performance is mainly because of more conflicts and work in the *uncore* simulation. These simulation times do not include the time spent generating BADCO models. Nevertheless, a benchmark can be integrated in many workload and many different simulations of the *uncore* and thus the one time cost for build a BADCO model is rapidly compensated by BADCO's speedup. It should be noted that BADCO still uses a detailed *uncore* simulator whose simulation speed may not be optimal and thus limiting the potential speedup that BADCO can provide.



(a) 2 cores

(b) 4 cores

(c) 8 cores

Figure 3.9: CPI measured with Zesto on the vertical axis versus estimated CPI with BADCO on the horizontal axis.

Number of cores	1	2	4	8
MIPS - Zesto	0.170	0.096	0.049	0.017
MIPS - BADCO	2.52	2.41	1.89	1.19
Speedup	14.8	25.19	38.88	68.1

Table 3.10: BADCO average speedup for 1, 2, 4 and 8 cores.

3.7 Summary

We introduced BADCO, a new behavioral application-dependent model of superscalar cores. A behavioral core model is like a black box emitting requests to the *uncore* at certain times. A behavioral core model can be connected to a detailed *uncore* model for studies where the focus is not the core itself, e.g., design space exploration of the *uncore* or study of multiprogrammed workloads. We have extended PDCM, a previously proposed core model, in order to model more accurately detailed superscalar processors. We also propose BADCO, a new behavioral core model. A BADCO model is built from two detailed simulations. Once the time to build the model is amortized, important simulation speedups can be obtained. We have compared the accuracy of BADCO with that of PDCM. From our experiments, we conclude that BADCO is on average more accurate than PDCM, essentially because it is based on two detailed simulations instead of a single one for PDCM. With BADCO, the average of the absolute CPI error is less than 4 % for all configurations and benchmarks we have tested. We have also evaluated the accuracy of BADCO for simulating multiprogram workloads, the average of the absolute CPI error is less than 5 % for 2, 4 and 8 cores, and all evaluated configurations. Moreover, we have demonstrated that BADCO offers a good qualitative accuracy, being able to predict how performance varies when we change the *uncore* configuration in both single and multicore execution. So far, the simulation speedups we have obtained with BADCO are typically between one and two orders of magnitude compared with Zesto.

Chapter 4

Multiprogram Workload Design

4.1 Introduction

The performance of an application executing on a multicore processor can be strongly impacted by applications running simultaneously on the other cores, mainly because of resource sharing (last-level cache, memory bandwidth, chip power...). This impact is not obvious, and quantifying it often requires detailed simulations.

The study of multicore performance on multiprogram workloads, i.e., sets of independent threads running simultaneously, is still a very active research area. The most widely used method for such study is to use a set of single thread benchmarks, to define a fixed set of multiprogram workloads from these benchmarks, to simulate these workloads and to quantify performance with a throughput metric.

The population of possible benchmark combinations may be very large. Hence most studies use a relatively small sample of a few tens, sometimes a few hundreds of workloads. In general, all the benchmarks in a suite are assumed to be equally important. Therefore we would like the sample to be representative of the whole population of possible workloads. Yet, there is no standard method in the computer architecture community for defining multiprogram workloads. There are some common practices, but not really a common method. More important, authors rarely demonstrate the representativeness of their workload samples. Indeed, it is difficult to assess the representativeness of a workload sample without simulating a larger number of workloads, which is precisely what we want to avoid. Approximate microarchitecture simulation methods that trade accuracy for simulation speed offer a way to solve this dilemma.

Approximate simulation is usually advocated for design-space exploration. We show in this chapter that fast approximate simulation can also help select representative multiprogram workloads in situations requiring detailed simulations (e.g., for estimating power consumption).

We investigate several sampling methods, using as a case study a comparison of several multicore last-level cache replacement policies. We performed simulations with

Zesto, a detailed microarchitecture simulator [64], and with BADCO, a fast approximate simulator presented in Chapter 3.

We show that, unless we know a priori that the microarchitecture being assessed *significantly* outperforms (or underperforms) the baseline microarchitecture, it is not safe to simulate only a few tens of random workloads, as frequently done in many studies. Hence it is necessary to simulate a large workload sample, which is possible with a fast approximate simulator. We propose a method for determining, from a representative subset of all possible workloads, what should be the size of a random workload sample. We propose an improved sampling method, balanced random sampling, that defines workloads in such a way that all the benchmarks are equally weighted in the sample. Sometimes, random sampling requires more than a few tens of workloads. We evaluate an alternative method, benchmark stratification, that defines workloads by first defining benchmark classes. However, this method is not significantly better than random sampling. Finally, we propose a new method, *workload stratification*, that is very effective at reducing the sample size when random sampling would require too large a sample.

This Chapter is organized as follows. Section 4.2 introduces the problem of multiprogram workload design. In Section 4.3, we propose a method for obtaining the size of a representative workload sample under random sampling. Section 4.4 describes our experimental setup. We evaluate experimentally our random sampling method in Section 4.5. Then Section 4.6 introduces and evaluates experimentally three alternative sampling methods. Section 4.7 gives a practical guideline. Finally, Section 4.8 summarizes the main chapter's contribution.

4.2 The problem of multiprogram workload design

Simulation objectives for a computer architect are generally to compare two or more multicore microarchitectures under some criterion such as execution time, multiprogram throughput, power consumption, fairness, etc. Generally one wants also to quantify the differences between microarchitectures. In this study we consider the problem of evaluating multiprogram throughput, i.e., the quantity of work done by the machine in a fixed time when executing simultaneously several independent threads. The usual procedure for evaluating multiprogram throughput is to take a set of benchmarks (e.g., the SPEC CPU benchmarks) and define some combinations of threads executing concurrently, on which the microarchitectures are evaluated.

We call *workload* a combination of K benchmarks, K being the number of logical cores[1]. The number of workloads out of B benchmarks is generally very large. If the cores are identical and interchangeable, and assuming that the same benchmark can be replicated several times, there is a population of $\binom{B+K-1}{K}$ possible workloads. Because detailed microarchitecture simulators are very slow, computer architects generally consider a *sample* of W workloads where W is typically only a few tens. The microarchi-

[1]Physical cores may be SMT

tectures being compared are simulated on all W workloads. For each microarchitecture, we obtain a total of $W \times K$ IPC (instructions per cycle) values, denoted IPC_{wk}, where $w \in [1, W]$ is the workload and $k \in [1, K]$ is the core. The $W \times K$ IPC values are then reduced to a single throughput value via a throughput metric. The microarchitecture whose throughput value on the W workloads is the highest is deemed to be the one offering the highest throughput on the full workload population.

The workload sample is generally much smaller than the full population, but there is no standard method for defining a representative sample, although there are some common practices. Yet, the method used for selecting the sample may change the conclusions of a study dramatically.

We did a survey of the papers published in three major computer architecture conferences, ISCA, MICRO and HPCA, from 2007 to march 2012. We identified 75 papers that have used fixed multiprogram workloads[2]. The vast majority of these 75 papers use a small subset of all possible workloads, ranging from a few workloads to a few hundreds. Many papers use a few tens of workloads and compute an average performance on them. Of the 75 papers, only 9 use a completely random selection of workloads. The 66 other papers classify benchmarks into classes and define workloads from these classes. In the vast majority of cases, the classes are defined "manually", based on the authors' understanding of the problem under study. Then, some workload types are defined. For instance, if there are two benchmark classes A and B and two identical cores, 3 types of workloads may be defined: AA, BB and AB. Then a certain number of workloads are defined for each workload type. The number of workloads and the method for defining them is more or less arbitrary. The practices here are very diverse depending on the author and on the problem studied. For instance, some authors choose to give more weight to certain workload types, sometimes without any reason. Some authors select benchmarks randomly under the constraint of the workload type. Some others choose a single benchmark to be representative of its class.

4.3 Random sampling

As noted in Section 4.2, random sampling is not the most popular method in the computer architecture community. Many authors prefer to work with a relatively small sample that they try to define (more or less carefully) so that it is representative. Yet, random sampling is a safe way to avoid biases, provided the sample is large enough. Moreover, random sampling lends itself to analytical modeling. We present in the remainder of this section a model for estimating the probability of drawing correct conclusions under random workload selection.

For a fixed W, the sample throughput defined by formula (2.9) can be viewed as a random variable, the sample space for that variable being all the possible subsets of

[2] We do not count the studies using a number of benchmarks small enough for allowing to simulate all the possible workloads.

W workloads out of a full population of $N = \binom{B+K-1}{K}$ workloads. The problem of comparing two microarchitectures X and Y can be stated as follows. We want to know whether or not Y yields a greater throughput than X. Let T_X and T_Y be the sample throughput of microarchitectures X and Y respectively. T_X and T_Y are two random variables. For the IPCT and WSU metrics, we define random variable D:

$$D \quad = \quad T_Y - T_X = \underset{w\in[1,W]}{\text{A-mean}}\ d(w) \tag{4.1}$$

where the random variable $d(w)$ is defined as

$$d(w) = t_Y(w) - t_X(w) \tag{4.2}$$

In words, $d(w)$ and D are respectively the per-workload and average throughput difference. If we have some information about the distribution of D, we may be able to compute the probability that D is positive. Because the W workloads are chosen randomly and independently from each other, the Central Limit Theorem (CLT) applies, and D can be approximated by a normal distribution [17].

Let μ and σ^2 be respectively the mean and variance of $d(w)$. The mean of D is also equal to μ and its variance is σ^2/W, assuming $W \ll N$. The **degree of confidence** that Y is better than X is equal to the probability that D is positive:

$$Pr(D \geq 0) \quad = \quad \frac{1}{2}\Big[1 + \text{erf}\Big(\frac{1}{c_v}\sqrt{\frac{W}{2}}\Big)\Big] \tag{4.3}$$

where $\text{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$ is the error function and $c_v = \sigma/\mu$ is the coefficient of variation of $d(w)$.

For the HSU metric, a H-mean is used in formulas (2.8) and (2.9), and it is the inverse of the HSU on which the CLT applies[3]. Thus for the HSU we define the random variable D as

$$D \quad = \quad \frac{1}{T_X} - \frac{1}{T_Y} = \underset{w\in[1,W]}{\text{A-mean}}\ d(w) \tag{4.4}$$

with the random variable $d(w)$ defined as

$$d(w) = \frac{1}{t_X(w)} - \frac{1}{t_Y(w)} \tag{4.5}$$

that is, $d(w)$ and D are respectively the per-workload and average reciprocal throughput difference. The coefficient of variation c_v of $d(w)$ is used in equation (4.3).

[3]Our goal is not to discuss which throughput metric should be used or not. The CLT applies to any throughput metric that can be expressed as a sum of per-workload terms. For instance, if one prefers to quantify throughput as a geometric mean of speedups [70], i.e., if a geometric mean is used in formulas (2.8) and (2.9), the CLT applies to the logarithm of throughput, which leads to define the random variables $D = \log T_Y - \log T_X$ and $d(w) = \log t_y(w) - \log t_x(w)$.

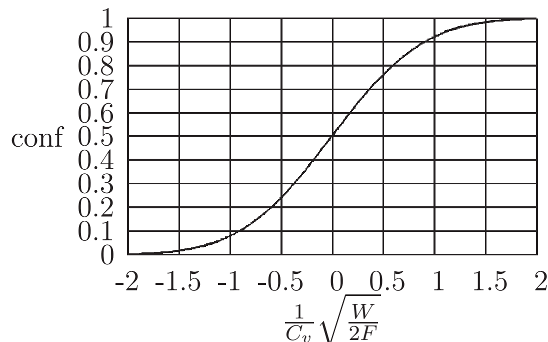


Figure 4.1: Degree of confidence as a function of $\frac{1}{c_v}\sqrt{\frac{W}{2}}$ (equation (4.3).

Figure 4.1 shows the degree of confidence as a function of $\frac{1}{c_v}\sqrt{\frac{W}{2}}$ (equation (4.3)). A degree of confidence close to zero means that it is very likely that Y is *not* better than X. The degree of confidence becomes very close to 0 or 1 for

$$\left|\frac{1}{c_v}\sqrt{\frac{W}{2}}\right| = 2$$

Solving this equation for W, we obtain the required sample size:

$$W = 8c_v^2 \tag{4.6}$$

The only parameter needed in this model is the coefficient of variation c_v, which is estimated from experiments. We present an experimental validation of the model in Section 4.5.1.

4.4 Experimental evaluation

4.4.1 Simulation setup

Our experiments analyze the performance of symmetric multicore with 2, 4 and 8 identical cores. Table 3.6 summarizes the cores characteristics. As a case study, we consider five *uncore* microarchitectures, each *uncore* corresponding to a different shared last-level cache replacement policy: LRU, RANDOM (RND), FIFO, DIP[81] and DRRIP [46]. Table 3.7 presents the *uncore* characteristics.

We build the workloads from 22 of the 29 SPEC CPU2006 benchmarks (the 22 benchmarks that we were able to simulate with Zesto). We simulate every *uncore* using BADCO for the full population of workloads whenever possible (253 workloads for 2 cores, 12650 workloads for 4 cores), or for a large sample when the number of possible combinations is huge (we consider 10000 workloads for 8 cores). We also perform Zesto

simulations for 250 randomly selected workloads for 2, 4 and 8 cores, and for every *uncore*. We compiled all the benchmarks with gcc-3.4 using the "-O3" optimization flag. For generating BADCO traces, we skip the first 40 billion instructions of each benchmark, and the trace represents the next 100 million instructions (no cache warming is done). We assume that simulations are reproducible, so that traces represent exactly the same sequence of dynamic μops. We used SimpleScalar EIO tracing feature [3], which is included in the Zesto simulation package.

During multiprogram execution, each core runs a separate threads. When a thread has finished executing its 100 million instructions earlier than the other threads, it is restarted[4]. This is done as many times as necessary until all the threads in the workloads have executed at least 100 million instructions. The IPC is measured only for the first 100 million committed instructions of each thread.

4.5 Experimental results for random sampling

4.5.1 Random sampling model validation

We experimentally validated formula (4.3) for the 10 pairs of replacement policies, for the 3 metrics (IPCT, WSU and HSU) and for 2, 4, and 8 cores. We measured the *experimental degree of confidence* that policy Y outperforms policy X for a given sample

[4]More rigorous multiprogram simulation methods could be used, such as the co-phase matrix method [93]. The problem of defining representative benchmark combinations is orthogonal and concerns the co-phase matrix method as well.

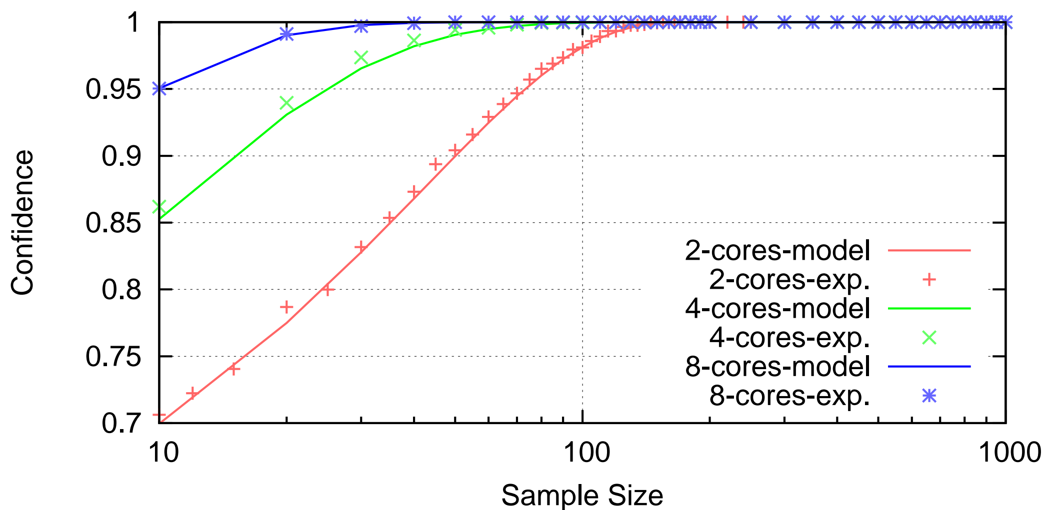


Figure 4.2: Confidence degree that *“DRRIP outperforms DIP”* as function of the sample size. Experimental result vs. analytical model. Throughput metric WSU.

size by generating 1000 random samples: the experimental degree of confidence is the fraction of samples for which the sample throughput of Y is greater than that of X. Figure 4.2 shows the result of this experimental validation for one pair of policies and one metric: the model curve matches the experimental points quite well, even for small samples. Although not shown, the other metrics and policy pairs exhibit similarly good matching between the model and the experiment.

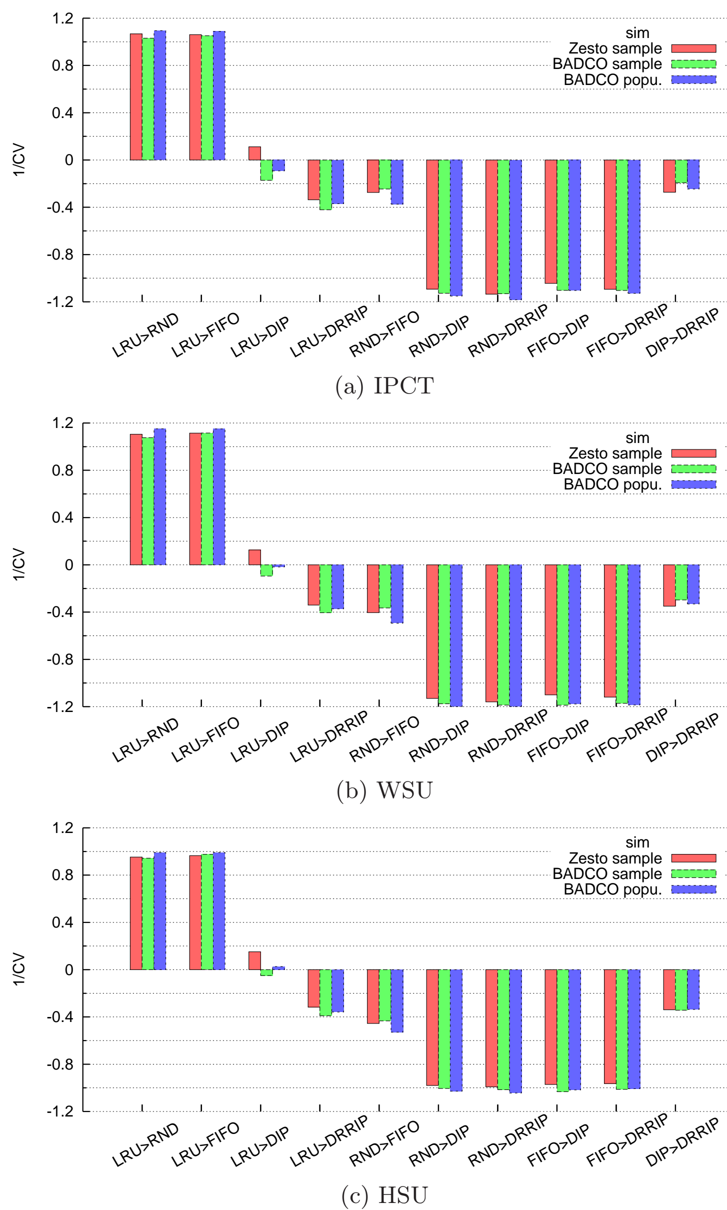
4.5.2 Performance difference impacts the sample size

Random workload selection requires knowing the appropriate sample size, i.e., the number of workloads that we must consider for drawing conclusions consistent with the full workload population with a reasonably high probability. As explained in Section 4.3, the coefficient of variation c_v of the random variable $d(w)$ is the only parameter needed to decide the sample size.

Figure 4.3 shows the inverse of the coefficient of variation ($1/c_v = \mu/\sigma$) for each pair of replacement policies, assuming a 4 core processor. The sign of $1/c_v$ indicates which policy in a pair performs best. The magnitude $|1/c_v|$ gives an indication of the performance difference between the two policies.

When the performance difference between two policies is significant, $|1/c_v|$ is relatively large. For instance, LRU significantly outperforms FIFO on all 3 metrics, and $c_v \approx 1$. From formula (4.6), about 8 randomly chosen workloads are sufficient to compare LRU and FIFO. In accordance with intuition, the larger the performance difference between two microarchitectures the fewer workloads are necessary to identify the best of the two.

However, when two policies have very close performance, such as LRU and DIP, $|1/c_v|$ is much smaller than 1. In such situation, a reasonable conclusion is that the two policies perform almost equally. However, we need a very large sample even for drawing this conclusion. For instance, the value of $|1/c_v|$ for LRU vs. DIP is smaller when computed on the full population than on the 250-workload sample. We cannot be certain that the value of c_v estimated on a sample is accurate unless we know a priori that one microarchitecture significantly outperforms the other. Two microarchitectures may have the same average performance on the full population of workloads, yet one microarchitecture may seem to outperform the other on a sample. In other words, if we have no a priori reason to believe that one microarchitecture significantly outperforms the other, we must consider a workload sample as large as possible. A fast qualitatively-accurate simulator such as BADCO allows to consider a large workload sample. If the sample is large enough, we can use it to estimate the coefficient of variation c_v. If c_v is greater than 10, we must conclude that the two policies perform equally on average.

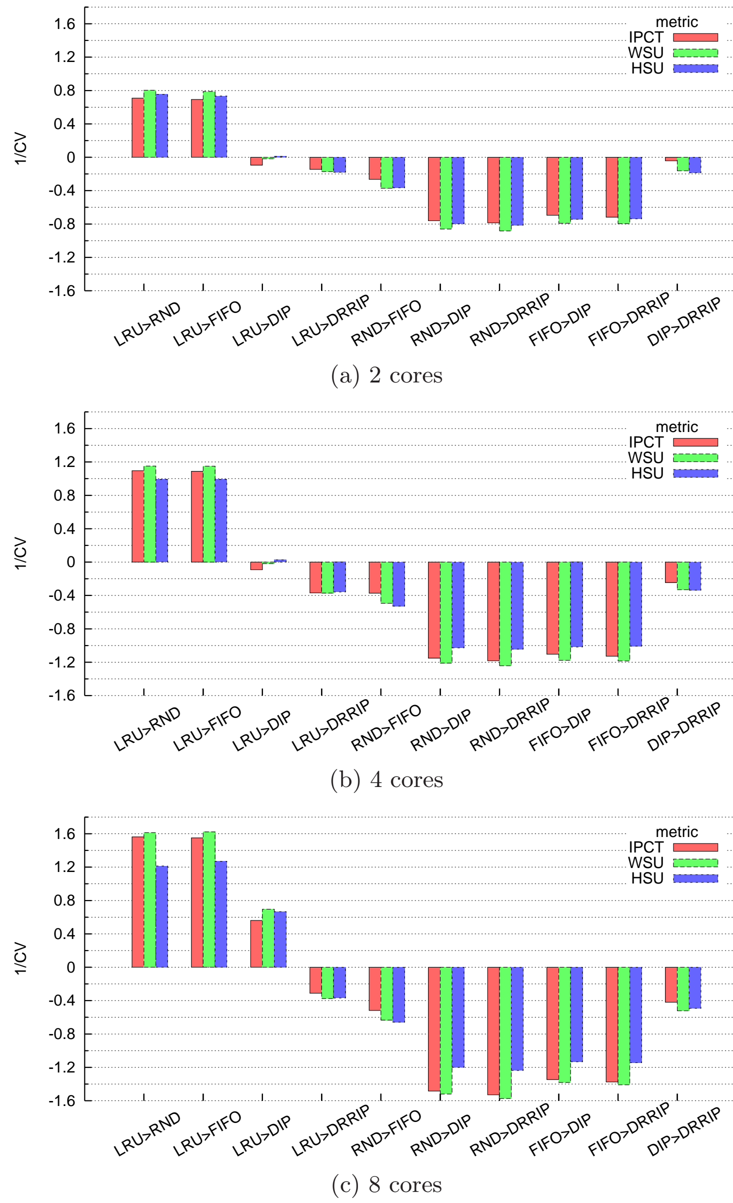


(a) IPCT

(b) WSU

(c) HSU

Figure 4.3: Inverse of the coefficient of variation, $1/c_v$, assuming 4 cores processor. The 3 graphs corresponds to the 3 throughput metrics: IPCT, WSU and HSU. Each group of 3 bars corresponds to a pair of replacement policies being compared. The first bar gives $1/c_v$ measured with the Zesto on a 250-workload sample. The second bar gives $1/c_v$ measured with BADCO on the same 250-workload sample. The third bar gives $1/c_v$ measured with BADCO on the full 12650-workload population.



(a) 2 cores

(b) 4 cores

(c) 8 cores

Figure 4.4: Inverse of the coefficient of variation, $1/c_v$, measured with BADCO on the full population of 12650 workloads. The 3 graphs corresponds to 2, 4 and 8 cores configurations. Each group of 3 bars corresponds to a pair of replacement policies being compared. The first bar gives $1/c_v$ measured for metric IPCT. The second bar gives $1/c_v$ measured for metric WSU. The third bar gives $1/c_v$ measured for metric HSU.

4.5.3 Different metrics may require different sample sizes

The physical meaning of a throughput metric depends on some assumptions regarding benchmarks and what they represent. Different metrics rely on different assumptions [70]. Computer architecture studies sometimes use several different throughput metrics to show that the conclusions are robust. Figure 4.4 shows the inverse of the coefficient of variation ($1/c_v$) for different pairs of policies on 4 cores and for the 3 throughput metrics. On this particular example, the sign of c_v does not depend on the throughput metric. That is, all 3 metrics rank replacement policies identically on a large enough workload sample. However, the magnitude of $|c_v|$ is not the same for all metrics. It means that some metrics permit using fewer workloads.

For example, when comparing RND and FIFO, $|1/c_v| \approx 0.4$ using the IPCT and $|1/c_v| \approx 0.5$ using the HSU. It means that a random sample of $8c_v^2 = 32$ workloads is sufficient with the HSU, but it would not be sufficient for the IPCT, which requires a random sample of $8c_v^2 = 50$ workloads. If one wants to use simultaneously several different throughput metrics on a fixed random workload sample, the required sample size must be determined for each metric, and the selected sample must be large enough for all metrics.

4.6 Alternative sampling methods

4.6.1 Balanced random sampling

If we consider the full population of workloads and count how many times a given benchmark occurs overall, we find that all the benchmarks occur the same number of times. This is consistent with the implicit assumption that all the benchmarks are equally important.

Random sampling, that we have considered so far, assumes that all the workloads have the same probability of being selected and that the same workload might be selected multiple times (though unlikely in a small sample). However, there is no guarantee that all the benchmarks occur exactly the same number of times in such random sample.

We propose another form of random sampling, *Balanced Random Sampling*. Balanced random sampling guarantees that every benchmark has the same number of occurrences in the whole sample. Hence, after picking a workload, all the workloads in the population may not have the same probability of being selected.

We have no mathematical model for this kind of sampling. Instead we have drawn 10000 balanced random samples and have computed experimentally the degree of confidence. Figure 4.5 shows the degree of confidence estimated with BADCO for several different sampling methods, including random sampling and balanced random sampling (the other methods are introduced afterwards).

Compared to simple random sampling, balanced random sampling is a more effective method, providing higher confidence for a given sample size. Balanced random sampling

is also, on average, the second most effective sampling method. However, there are still some situations such as the one in Figure 4.5a where the required sample size is very large.

4.6.2 Stratified random sampling

The workload population is generally not homogeneous. For example, let us assume that microarchitecture Y consistently outperforms microarchitecture X on 80 % of the workload population, while X consistently outperforms Y on the remaining 20 %. The knowledge of these subsets allows us to define a more representative sample. Instead of taking a single sample of W random workloads, we could take $0.8 \times W$ samples randomly from the first subset and $0.2 \times W$ workloads randomly from the second subset. This is a well-known method in statistics, called *stratified sampling* [17]. The method generalizes as follows.

The full population of N workloads is divided into L subsets $S_1, S_2, ..., S_L$ of $N_1, N_2, ..., N_L$ workloads respectively. The subsets, called *strata*, are non overlapping, and each workload in the population belongs to one stratum, so we have

$$N_1 + N_2 + ... + N_L = N$$

Once strata are defined, a random sample of W_h workloads is drawn independently from each stratum $S_h, h \in [1, L]$. The total sample size W is

$$W = W_1 + W_2 + ... + W_L$$

Global throughput is no longer computed with formula (2.9) but with a weighted arithmetic mean (WA-mean) or a weighted harmonic mean (WH-mean) depending on the throughput metric:

$$T = \underset{h \in [1,L]}{\text{WX-mean}} \; \underset{w \in S_h}{\text{X-mean}} \; t(w) \tag{4.7}$$

where WX-mean stands for WA-mean or WH-mean and where the weight for stratum S_h is N_h/N. If the strata are well defined, it is possible to divide a very heterogeneous workload population into strata that are internally homogeneous, so that the coefficient of variation of each stratum is small. As a result, a precise estimate of throughput for a stratum can be obtained from a small sample in that stratum. There are many different ways to define strata. Ideally, we would like to have the minimum number of strata with minimum W_h that produce maximum precision. It is important to note that stratified sampling requires to draw samples from each stratum. Hence W cannot be less than the number of strata. In the remainder of section, we compare two different ways to define strata: *benchmark stratification* and *workload stratification*.

MPKI Classe	Benchmarks
Low	povray, gromacs, milc, calculix, namd, dealII, perlbench, gobmk, h264ref, hmmer, sjeng
Medium	bzip2, gcc, astar, zeusmp, cactusADM
High	libquantum, omentpp, leslie3d, bwaves, mcf, soplex

Table 4.1: Classification of SPEC benchmarks according to memory intensity: Low ($MPKI < 1$), Medium ($MPKI < 5$), and High ($MPKI \geq 5$).

4.6.2.1 Benchmark stratification

It is common in computer architecture studies to define multiprogram workloads by first defining benchmark classes (cf. Section 4.2). The main assumption is that benchmarks in the same class exhibit similar behavior. Benchmark classes by themselves do not constitute strata but allow to build workload strata. We can construct strata according to the number of occurrences of each benchmark class in a workload. For example, the workloads composed of benchmarks all belonging to a given class constitute a stratum. Assuming there are M benchmark classes $C_1, C_2, ..., C_M$, we can represent a stratum with an n-tuple $(c_1, c_2, ..., c_M)$ where c_i is the number of occurrences of class C_i in a workload, with the constraint $\sum_{i=1}^{M} c_i = K$, the number of cores. That is, workloads with the same number of occurrences per class belong to the same stratum. This method defines $L = \binom{M+K-1}{K}$ distinct strata. The size of a stratum is

$$N_h = \prod_{i=1}^{M} \binom{b_i + c_i - 1}{c_i}$$

where b_i is the number of benchmarks in class C_i. Table 4.1 shows a classification of the SPEC CPU2006 benchmarks according to the memory intensity measured in misses per kilo-instruction (MPKI). For a 4 core processor, this classification generates 15 strata, hence $(c_{low}, c_{med}, c_{high})$ = (004, 013, 022, 031, 040, 103, 112, 121, 130, 202, 211, 220, 301, 310, 400). Using this stratification, we have drawn 10000 stratified samples and have estimated experimentally the degree of confidence for policy pairs comparisons. Figure 4.5 shows the degree of confidence with benchmark stratification. For almost all sample sizes, benchmark stratification increases the degree of confidence to some extent, but does not reduce dramatically the sample size required to reach a high degree of confidence.

It should be noted that the benchmark stratification method described here is an attempt to formalize some common practices that are diverse and not always explicit. The studies we are aware of that define multiprogram workloads by first defining benchmark classes neither use stratified sampling nor formula (4.7). Note also that classifying benchmarks according to the MPKI is probably not the best classification for studying

replacement policies. Nevertheless, the effectiveness of benchmark stratification strongly depends on the authors' intuition.

4.6.2.2 Workload stratification

Fast approximate simulators such as BADCO allow to estimate the throughput on large samples of thousands of workloads. Once approximate throughput values have been obtained for all workloads in the large sample, defining strata directly from these values is straightforward. As we seek to compare two microarchitectures according to a certain throughput metric, we can define strata based on the distribution of $d(w)$ for that pair of microarchitectures (see section 4.3). The proposed method is as follows:

1. Measure $d(w)$ for every workload in the large sample.
2. Sort the workloads according to $d(w)$.
3. Process the workloads in ascending order of $d(w)$, putting workloads in the same stratum
4. When the stratum has reached a minimum size W_T and when the standard deviation of the stratum exceeds a certain threshold T_{SD}, create a new stratum and repeat the previous step.

Parameters T_{SD} and W_T allow to control the number of strata. There is a tradeoff between the number of strata and the gain in precision we can obtain from *workload stratification*.

The degree of confidence obtained with *workload stratification* is shown in Figure 4.5 for a 4-core processor using the IPCT metric, $T_{SD} = 0.001$ and $W_T = 50$. It is very important to define strata separately and independently for each pair of microarchitectures and for each metric. For the pair FIFO-RND and a sample as small as 10 workloads, the degree of confidence with *workload stratification* is approximately 100 % while simple random sampling requires about 80 workloads to reach the same confidence. The pair DIP-LRU requires 50 workloads with *workload stratification* while random sampling requires 800 workloads to reach an equivalent confidence. The performance difference of DRRIP vs. FIFO is large enough for all sampling methods to bring nearly 100 % of confidence with just 10 workloads.

4.6.3 Actual degree of confidence

The degrees of confidence presented in Figure 4.5 were estimated with BADCO in order to isolate the error coming from workload sampling from the error due to approximate simulation, i.e., as if BADCO were 100% accurate. However in practice the approximate simulator is also a source of inaccuracy. Figure 4.6 shows the experimental degree of

confidence for DIP vs. LRU for small sample sizes and for the different sampling methods. Here the degree of confidence is measured with Zesto, but *workload stratification* is done with BADCO.

We did the experiment as follows. For 2 cores, we have simulated with Zesto the full population of 253 workloads. For 4 cores and 8 cores, we have simulated 250 workloads. For a given sample size and for each sampling method[5], we take 100 samples, each sample consisting of workloads that we have simulated with Zesto. We compute the per-sample throughput metric (here, the IPCT) for each of the 100 samples and for DIP and LRU. The experimental degree of confidence is the fraction of samples on which DIP outperforms LRU.

The results in Figure 4.6 confirm that the degree of confidence of samples selected with *workload stratification* outperform the degree of confidence of those selected with random, balanced random and benchmark stratification sampling methods. However, the degree of confidence measured with Zesto for workload stratification and 4 cores seems to be less than the degree of confidence estimated with BADCO for 4 cores on the pair LRU-DIP in Figure 4.5.

4.7 Practical guidelines in multiprogram workload selection

The method we propose relies on qualitatively accurate approximate simulation. It is not intended for design space exploration, but for studying incremental modifications of a microarchitecture, i.e., for comparing a baseline microarchitecture and a new microarchitecture. Moreover it is most useful when it is not obvious a priori whether the new microarchitecture outperforms the baseline. Detailed microarchitecture simulation is used to obtain information that the approximate simulator does not provide, such as power consumption (e.g., to find if the extra hardware complexity is worth the performance gain). In this situation, the two machines differ only in some parts of the microarchitecture that the approximate simulator should model precisely. The parts of the microarchitecture that are identical in both machines can be abstracted for simulation speed. For example, if one wants to compare two branch predictors for an SMT core, the approximate simulator should model the branch predictors precisely, but the other core mechanisms can be approximated.

Developing an ad-hoc approximate simulator requires some effort. Approximate simulators are commonly used in the industry for design space exploration, hence for some studies it may be sufficient to reuse and modify an already available approximate simulator. Publicly available approximate simulators include Sniper [11], recently de-

[5]We did not apply balanced random sampling for 4 cores and 8 cores because the method we used for automatically defining a balanced sample works with the full workload population. In real situations this would not be a problem because detailed simulations are normally done *after* the workload sample is defined.

veloped at the University of Ghent, which can be used for various studies, e.g., *uncore* studies or branch prediction studies. If one wants to compare different *uncore* microarchitectures, an approximate simulation method such as BADCO is also possible. It took us roughly one person-month of work to implement the BADCO core models for this study.

Once we have a fast approximate simulator, we simulate a large workload sample for the two microarchitectures (balanced random sampling should be used so that all the benchmarks have the same weight, cf. Section 4.6.1). The required size for such sample does not depend on the full population size but on the actual coefficient of variation c_v (formula (4.6)). However, the actual c_v cannot be estimated with certainty from a workload sample. Nevertheless, the larger the sample, the more likely it is representative. For instance if $c_v < 10$, i.e., if the two microarchitectures are not equivalent throughput-wise, 800 random workloads are sufficient.

Then, assuming the large sample is representative of the full population, we estimate the coefficient of variation c_v on this sample. If c_v is greater than 10, we declare that the two machines offer the same average throughput. If c_v is less than 2, random sampling may be sufficient, as a few tens of workloads ensures a high confidence (cf. formula (4.6)). Nevertheless, for such small sample, balanced random sampling should be preferred over random sampling. It is when c_v is in the $[2, 10]$ range that we recommend using *workload stratification*. However, one must keep in mind that the workload sample thus defined is valid only for a pair of microarchitectures and for a throughput metric.

4.7.1 Simulation overhead: example

As an example, let us consider the top graph of Figure 4.5 (DIP vs. LRU) and the speedup numbers provided in Table 3.10, and let us assume that we simulate 100 million instructions per thread, i.e., 400 million instructions per workload. With balanced random sampling, 30 workloads yields a confidence of 75% and necessitate roughly $30 \times (400/0.049)/3600$ cpu*hours of Zesto simulation for each replacement policy, that is, 136 cpu*hours in total. To reach a confidence of 90% under balanced random sampling, we need 120 workloads, which requires $2 \times 120 \times (400/0.049)/3600 \approx 544$ cpu*hours. of Zesto simulation. That is, to increase the degree of confidence from 75% to 90%, we need 300% extra simulation. With workload stratification, 30 workloads are sufficient to obtain 99% of confidence, which takes 136 cpu*hours of Zesto simulation. In order to identify 30 "good" workloads, we first generate a BADCO model for each benchmark, which takes $22 \times 2 \times (100/0.17)/3600 = 7$ cpu*hours (22 benchmarks, 2 traces per benchmark, 100 million instructions, Zesto single-core simulation speed). Then we simulate 800 random workloads with BADCO for each policy (notice on this example that 600 random workloads are sufficient to reach 99% of confidence). This takes $2 \times 800 \times (400/1.89)/3600 = 94$ cpu*hours. Increasing the degree of confidence from 75% to 99% requires $(7 + 94)/136 \approx 74\%$ extra simulation with *workload stratification*. On this example, *workload stratification* yields more confidence than random sampling for

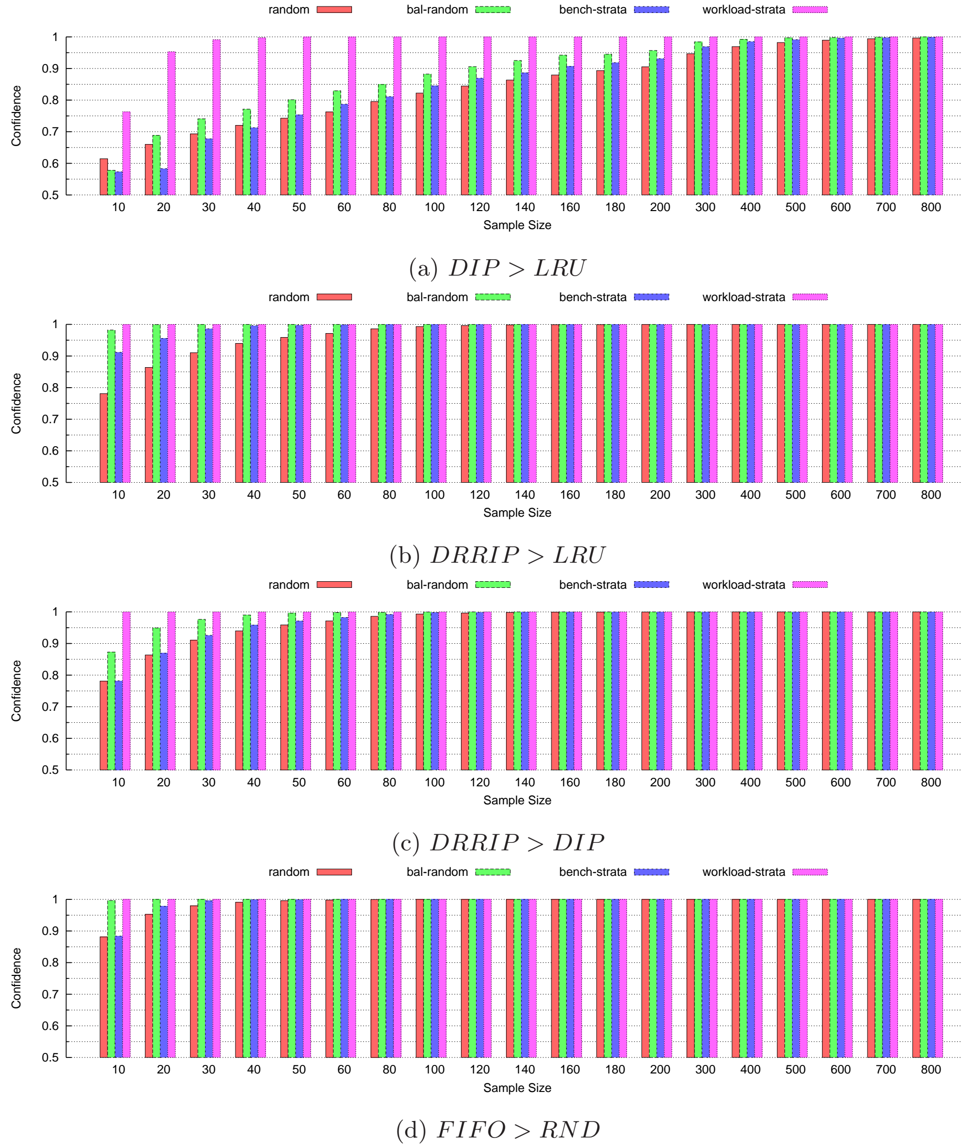
a simulation overhead that is 4 times smaller.

4.8 Summary

The multiprogram workloads used in computer architecture studies are often defined without any clear method and with no guarantee that the chosen sample is representative of the workload population. Indeed, it is difficult to assess the representativeness of a workload sample without simulating a much larger number of workloads, which is precisely what we want to avoid by using sampling. We propose to solve this dilemma with approximate simulations that trade accuracy for simulation speed. Approximate simulation is generally used for design-space exploration. We have shown in this study that approximate simulation can also help selecting multiprogram workloads in situations requiring detailed microarchitecture simulations.

We have investigated several methods for defining multiprogram workloads. As a case study, we compared several multicore last-level cache replacement policies. We have shown that, unless we know a priori that the microarchitecture under study *significantly* outperforms (or underperforms) the baseline, it is not safe to simulate only a few tens of randomly chosen workloads. An approximate yet qualitatively accurate simulator, because it runs faster, allows to consider a much large number of workloads. We have proposed a method for defining, from a large workload sample, a smaller sample to simulate with a detailed simulator.

We have considered in this study the problem of defining workload samples that tell if a microarchitecture outperforms another, consistently with the full workload population. To our knowledge, the problem of defining workload samples that provide accurate speedups with high probability is still open.



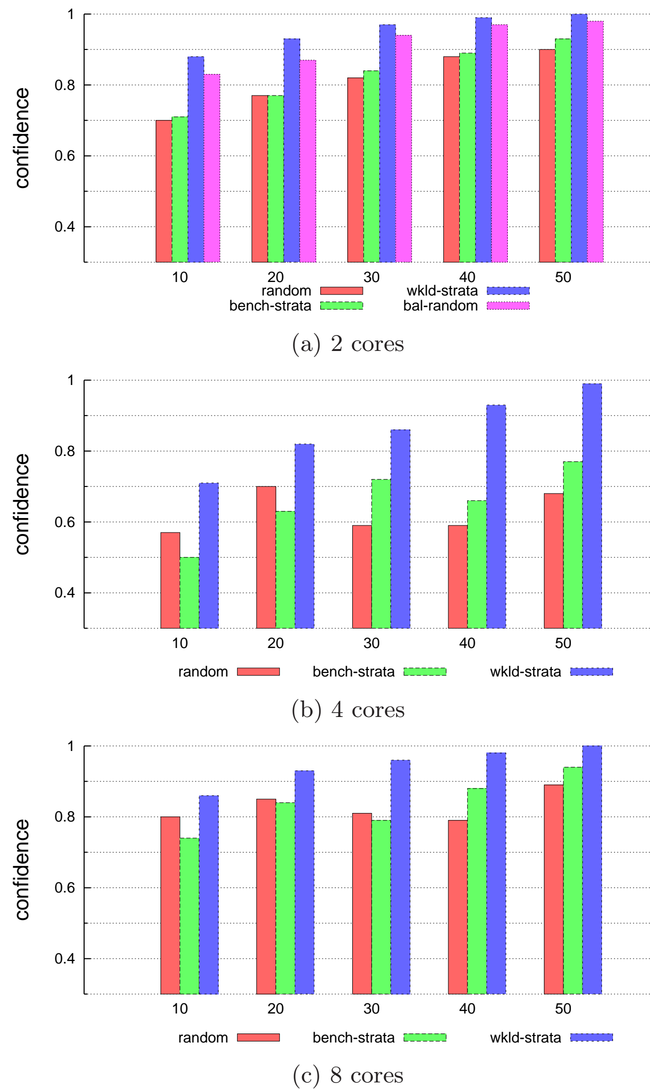
(a) $DIP > LRU$

(b) $DRRIP > LRU$

(c) $DRRIP > DIP$

(d) $FIFO > RND$

Figure 4.5: Alternative sampling methods. Confidence degree as function of the sample size for (a) $DIP > LRU$, (b) $DRRIP > LRU$, (c) $DRRIP > DIP$, and (c) $FIFO > RND$. Throughput metric $IPCT$ for 4 core configuration. Each group of 4 bars corresponds to samples size. The first bar gives the model computed confidence for simple random sampling. The second bar gives the experimental confidence for balanced random sampling. The third bar gives the experimental confidence for benchmark stratification. The fourth bar present the experimental confidence for *workload stratification*



(a) 2 cores

(b) 4 cores

(c) 8 cores

Figure 4.6: Experimental degree of confidence measured with Zesto as a function of the sample size for $DIP > LRU$, 4 sampling methods (simple random sampling, balanced random sampling, benchmark stratification and *workload stratification*), using the IPCT metric.

Chapter 5

Conclusion

In this thesis, we have shown that among all simulation tools available to a computer architect, behavioral core modeling is a competitive option for multicore studies where the research focus is in the *uncore* microarchitecture and considering independent tasks. We demonstrated that behavioral core models can bring speedups between one and two orders of magnitude with average CPI errors of less than 5 %. We have also demonstrated that behavioral core models can help in the selection of multiprogram workloads. This thesis makes contributions in the context of fast simulation tools for multicore systems, and in the context of methodologies for the evaluation of multicore research ideas.

Contributions

In the context of fast simulation tools for multicore processors, we have proposed behavioral core models as an alternative simulation tool when the research target is specifically the *uncore*. We showed that a behavioral core model such as PDCM, a previously proposed model, can model a superscalar core with an average error of less than 5 %. We achieved this accuracy by modeling the impact of wrong-path requests, prefetch requests, delayed hits, etc. These additional requests were not considered in the original PDCM model. We also proposed BADCO, a new behavioral application-dependent superscalar core model. BADCO exhibits an average error of less than 3.5 % for single core processors and all the evaluated configurations. We achieved this level of accuracy in BADCO by considering an extra detailed simulation that helps to expose additional dependencies and requests that have an effect upon long latency misses. We have also evaluated the accuracy of BADCO for simulating multiprogram workloads, and we obtained average errors of less than 5 % for 2, 4, and 8 cores in all the evaluated configurations. We also demonstrated that BADCO predicts correctly the changes in performance in both single and multicore execution. Finally, using Zesto as reference, we have obtained average speedups between one and two orders of magnitude. In particular, we have observed an increase of the speedup with the number of cores, passing

from 14.8x to 68.1x for simulations with 1 and 8 cores respectively.

In the context of new methodologies for multicore systems, we studied the problem of selecting workload samples for multicore architectures. We established the difficulty of assessing the representativeness of a workload sample without simulating a much larger number of workloads. We tackled this problem using approximate simulation. We have shown in this thesis that approximate simulation can help to select multiprogram workloads in situations requiring detailed microarchitecture simulations.

We proposed an alternative method that estimates the degree of confidence of a random sample as the probability of drawing correct conclusions when comparing two microarchitectures. The method can be used either to compute the confidence of a sample or the required sample size provided that we can estimate the coefficient of variation. We also showed that an approximate simulator can help estimate the coefficient of variation.

We proposed and compared different sampling methods for defining multiprogram workloads. We evaluated their effectiveness on a case study that compares several multicore last-level cache replacement policies. We showed that random sampling, the simplest method, is robust enough to define a representative sample of workloads, provided the sample is big enough. We proposed a new method, *workload stratification*, which is very effective at reducing the sample size in situations where random sampling would require a large sample. Workload stratification uses approximate simulation for defining the sample.

Open problems and perspectives

There are a number of perspectives for further exploiting the advantages of core modeling. In particular, there is potential in BADCO models for simulating multi-thread applications, heterogeneous multicore architectures, and as a power/energy estimation tool. With respect to multiprogram workload selection, there are some questions that deserve further study. For instance, Can we define the degree of confidence of a sample in some other alternative ways? Can cluster analysis help define strata in *workload stratification*? How can we take into account the effect of program phases on the selection of representative multiprogram workloads?

Behavioral core models

In this thesis, we have presented BADCO models for simulating single-core applications and multiprogram workloads. However, we did not explore the possibility of using BADCO for simulating multi-thread workloads. Trace-driven simulation cannot simulate accurately the behavior of non-deterministic parallel programs for which the sequence of instructions executed by a thread may be strongly dependent on the timing of requests to the *uncore* [40]. However, some previous studies have shown that trace-driven simulation could reproduce somewhat accurately the behavior of *certain* parallel

programs [40, 39], and it may be possible to implement behavioral core models for such programs [15, 82].

Heterogeneous chip multiprocessors [54, 4, 55] present unique opportunities for improving system throughput, reducing processor power, and mitigating Amdahl's law. On-chip heterogeneity allows the processor to better match execution resources to each application's needs. BADCO models can be a effective tool for studying heterogeneous chip multiprocessors. One may build BADCO models for different microarchitectures and then combine them in all kinds of ways for exploring the huge space of options that heterogeneous chip multiprocessors bring.

So far we have used behavioral core models and specifically BADCO models to evaluate performance. A possibility not explored is the use of BADCO models for power/energy consumption estimation. In the same way as trace T0 abstracts the dynamic execution of instructions in a superscalar pipeline, it might be possible to abstract the dynamic power consumption of a superscalar core. This is a topic for future work.

Multiprogram workload selection

One important insights that we have obtained is the need for defining analytical methods to asses the representativeness of random samples. The method of confidence intervals is the most common for computing the degree of confidence of random samples. But confidence intervals do not consider relative comparisons between microarchitectures. In this thesis, we have proposed an alternative method for defining the degree of confidence. The method responds to the question: how large must be a sample for comparing two microarchitectures and obtaining the correct conclusion about which one is best. The method guarantees that the sample is large enough for comparing correctly the relative performance, but it does not guarantee anything about the measured speedup. If the goal is to obtain a close estimation of the speedup, a new method for defining the sample's degree of confidence must be found. Our method is also restrictive in the sense that the relative comparison is limited to two microarchitectures. If the goal is to rank the global performance of several microarchitectures with our method, then we must do pairwise comparison among all microarchitectures. It may be possible to define a new method that considers multiple microarchitectures simultaneously.

We have also introduced *workload stratification*, a sampling method alternative to random sampling. We have demonstrated that *workload stratification* is more effective at defining representative multiprogram workloads. To be effective, *workload stratification* must define groups of workloads that have similar behavior. In particular, in this thesis we build strata from the difference in performance between two microarchitectures on every workload. In Section 4.6.2.2, we proposed a very simple algorithm to cluster the workload differences. We believe that more advanced clustering algorithms may increase the effectiveness of *workload stratification*.

In this thesis we did not consider the benchmark's phase behavior. Some methods

such as the co-phase matrix [93] have been proposed to deal with phase behavior. The problem of defining representative multiprogram workloads is orthogonal and concerns the co-phase matrix method as well. However, further studies combining our *workload stratification* method and the co-phase matrix method are needed.

Annexe A

Résumé en français

A.1 Introduction

Au début de l'ère informatique, les architectes d'ordinateurs s'appuyaient sur l'intuition et des modèles analytiques simples pour choisir entre les variantes d'une conception. Actuellement, les processeurs sont trop complexes pour s'appuyer sur l'intuition. Les architectes d'ordinateurs ont besoin d'outils et de méthodologies appropriés pour l'évaluation des performances, qui permettent de maitriser la complexité du processeur et de prendre des décisions de conception appropriées.

Ces dernières années, la recherche en microarchitecture a changé sa focalisation sur les processeurs simple cœur pour maintenant se concentrer sur les processeurs multi-cœurs. Plus précisément l'effort de recherche est passé de la microarchitecture du cœur à la microarchitecture de la hiérarchie mémoire. Les modèles précis au cycle près pour processeurs multi-cœurs avec des centaines ou même des milliers de cœurs ne sont pas pratiques pour simuler des charges multitâches réelles du fait de la lenteur de la simulation. Un grand pourcentage du temps de simulation est consacré à la simulation des différents cœurs, et ce pourcentage augmente linéairement avec chaque génération de processeur. Les modèles approximatifs sacrifient de la précision pour une vitesse de simulation accrue, et sont la seule option pour certains types de recherche. En particulier, ces modèles sont très utiles pour étudier l'impact du partage des ressources entre les cœurs, où les ressources partagées peuvent être : les caches, le réseau sur puce, le bus mémoire, la puissance, la température, etc.

La modélisation comportementale de cœur superscalaire est un moyen d'échanger de la précision contre de la vitesse de simulation dans les situations où l'objet de l'étude n'est pas dans le cœur, mais ce qui se trouve à l'extérieur du cœur, et notamment la hiérarchie mémoire. Cette méthode considère un coeur superscalaire comme une boite noire émettant des requêtes vers le reste du processeur à des instants déterminés. Ces instants de requêtes dépendent non seulement de la configuration de la hiérarchie mémoire mais aussi des interférences sur les ressources partagées dues à l'activité des autres coeurs.

Ainsi, les modèles comportementaux essayent d'imiter la manière dont les instants de requêtes changent. Un ou plusieurs modèles comportementaux peuvent être connectés à un modèle de hiérarchie mémoire précis au cycle près. Ces modèles comportementaux sont construits à partir de simulations détaillées.

En plus d'outils de simulation rapide, les processeurs multi-cœurs exigent également des méthodes de simulation plus rigoureuses. Il existe plusieurs méthodes couramment utilisées pour simuler les architectures simple cœur. De telles méthodes doivent être adaptées ou même repensées pour la simulation des architectures multi-cœurs. Un problème méthodologique qui n'a pas reçu une attention suffisante est le problème de la sélection des charges de travail multiprogrammées pour l'évaluation de performance des architectures multi-cœurs. La population de toutes les charges multiprogrammées possibles est immense. Par conséquent, la plupart des études ont utilisé un échantillon relativement petit de quelques dizaines voire quelques centaines de charges multiprogrammées. En supposant que toutes les charges multiprogrammées sont également importantes, nous voulons que l'échantillon soit représentatif de la population. Toutefois, aucune méthode standard n'existe dans la communauté pour définir les charges multiprogrammées. Bien qu'il existe des pratiques communes, il n'y a pas vraiment de méthode commune.

A.2 Contributions

Les principales contributions de cette thèse sont les suivantes :

Adaptation de PDCM pour la modélisation des architectures simple-cœur réalistes Nous étudions le modèle PDCM, un modèle comportemental proposé précédemment, et nous évaluons sa capacité à modéliser avec précision une architecture superscalaire moderne. Nous avons d'abord identifié les principales sources d'imprécision de PDCM. Ensuite, nous avons proposé et mis en œuvre certaines modifications du modèle PDCM pour modeliser certaines caractéristiques importantes du cœur comme la prédiction de branchements et le préchargement des caches de premier niveau. De cette manière, l'erreur moyenne de 8 % avec la version originale PDCM est réduite à 4 % avec notre version améliorée de PDCM.

BADCO : une nouvelle méthode pour définir des modèles comportementaux Nous décrivons et étudions une nouvelle méthode pour définir des modèles comportementaux pour les cœurs superscalaires modernes. Le modèle comportemental proposé, BADCO (pour son sigle en anglais), prédit le temps d'exécution d'un programme séquentiel avec une erreur moyenne de moins de 3,5 %. Nous montrons également que BADCO est plus précis que PDCM d'un point de vue qualitatif, étant capable de prédire les changements de performance quand nous changeons la configuration de la hiérarchie

mémoire. Les gains en vitesse de simulation par rapport à la simulation détaillée sont typiquement entre un et deux ordres de grandeur.

Un nouveau modèle analytique pour calculer le degré de confiance d'un échantillon de charges multiprogrammées Les intervalles de confiance sont la méthode la plus courante pour calculer le degré de confiance d'un échantillon aléatoire. Nous proposons une nouvelle méthode où le degré de confiance d'un échantillon de charges multiprogrammées est défini comme la probabilité d'atteindre des conclusions correctes quand on compare deux microarchitectures sur cet échantillon. Ce modèle analytique calcule le degré de confiance en fonction de la taille de l'échantillon et du coefficient de variation d'une variable aléatoire définie sur toute la population de charges multiprogrammées. La méthode peut être utilisée à la fois pour calculer le degré de confiance d'un échantillon ou pour calculer la taille d'un échantillon représentatif, en supposant que nous avons une estimation du coefficient de variation. Nous montrons qu'un simulateur approximatif peut aider à estimer le coefficient de variation.

Stratification des charges de travail : une nouvelle méthode pour sélectionner des charges multiprogrammées
Nous proposons et comparons différentes méthodes d'échantillonnage pour définir des charges multiprogrammées pour l'étude d'une architecture multi-cœur. Nous évaluons l'efficacité de ces techniques dans une étude de cas qui compare cinq politiques de remplacement du cache partagé sur des architectures à 2, 4 et 8 cœurs. Nous montrons que l'échantillonnage aléatoire, la méthode la plus simple, est suffisamment robuste pour définir un échantillon représentatif, pourvu que l'échantillon soit suffisamment grand. Nous proposons également une nouvelle méthode d'échantillonnage, la stratification des charges de travail, qui permet de définir un échantillon représentatif relativement petit dans les situations où l'échantillonnage aléatoire nécessiterait un très grand échantillon. La stratification des charges de travail utilise la simulation approximative pour définir un échantillon représentatif.

A.3 Modèles comportementaux

Au cours de la mise au point d'un processeur, il est possible d'utiliser divers types de modèles dans diverses étapes du procédé et à des fins différentes. Par exemple, les modèles analytiques sont utiles pour comprendre un problème. Pendant ce temps, les modèles de performance rapides sont utiles dans les premiers stades du développement pour comparer des options de conception.

Un modèle de coeur est un modèle approximatif de la microarchitecture superscalaire d'un cœur. Un modèle de coeur peut également être connecté à un modèle détaillé de la hiérarchie mémoire. Ainsi, la communication entre les deux modèles se fait à travers des requêtes générées par le coeur et exécutées par la hiérarchie mémoire. L'objectif

principal d'un modèle de coeur est de fournir aux concepteurs un outil de simulation rapide pour les études où l'objet de la recherche n'est pas le cœur lui-même mais la hiérarchie mémoire.

Il existe deux variantes principales de modèle de coeur : les modèles structurels et les modèles comportementaux. Les modèles structurels essayent de reproduire le comportement interne de la microarchitecture du cœur. L'augmentation de la vitesse de simulation est obtenue en ne simulant que les mécanismes qui ont le plus d'impact sur les performances. Les modèles comportementaux essayent d'imiter le comportement externe du cœur, qui est considéré avant tout comme une boîte noire émettant des requêtes vers la hiérarchie mémoire. Contrairement aux modèles structurels, les modèles comportementaux sont dérivés de simulation détaillées, ce qui peut être un inconvénient dans certains cas. Toutefois, dans les situations où le temps de construction du modèle peut être amorti, les modèles comportementaux sont potentiellement plus rapides que les modèles structurels à niveau de précision égal.

A.3.1 Modèle comportemental PDCM

Lee et al. ont présenté dans [61] trois modèles comportementaux différents : le défaut de cache isolé, le défaut de cache indépendant, et le défaut de cache dépendant (PDCM pour son sigle en anglais). Ces modèles utilisent une trace des accès au cache L2 (cache de deuxième niveau), qui sont annotés avec des informations complémentaires permettant de reproduire le comportement d'un coeur superscalaire à exécution dans le désordre. Parmi les trois modèles proposés par Lee et al., PDCM est le plus précis. Pour ce faire, PDCM utilise la taille du ROB (fenêtre d'instructions) pour contrôler le nombre d'éléments qui sont traités simultanément. PDCM considère également les dépendances entre requêtes, de sorte que deux requêtes dépendantes soient exécutées l'une après l'autre. Lee et al. ont montré que PDCM est suffisamment précis pour modéliser des cœurs idéaux ayant un prédicteur de branchements parfait et ne prenant pas en compte le préchargement des caches de premier niveau.

L'implémentation initiale de PDCM était basée sur le modèle de microarchitecture *sim-outorder* fourni avec l'environnement de simulation SimpleScalar. Dans cette thèse, nous évaluons PDCM avec un modèle de microarchitecture plus détaillé, *Zesto*. Notamment, nous avons évalué l'effet des branchements mal prédits et du préchargement dans les caches de premier niveau sur la précision de PDCM.

En améliorant la version initiale de PDCM, nous avons réussi à obtenir un niveau de précision similaire à celui obtenu par Lee et al. sur sim-outorder. La figure A.1 illustre nos efforts. La première barre à gauche montre la précision obtenue avec PDCM dans notre mise en œuvre initiale, qui est basée sur ce qui est explicitement décrit par Lee et al. dans [61]. Cette mise en œuvre prend en compte le nombre limité de MSHRs (requêtes en cours) et suppose une prédiction de branchements parfaite. La deuxième barre montre l'impact que la prédiction de branchement réelle et le préchargement dans les caches de premier niveau ont sur la précision de PDCM. Comme attendu, la

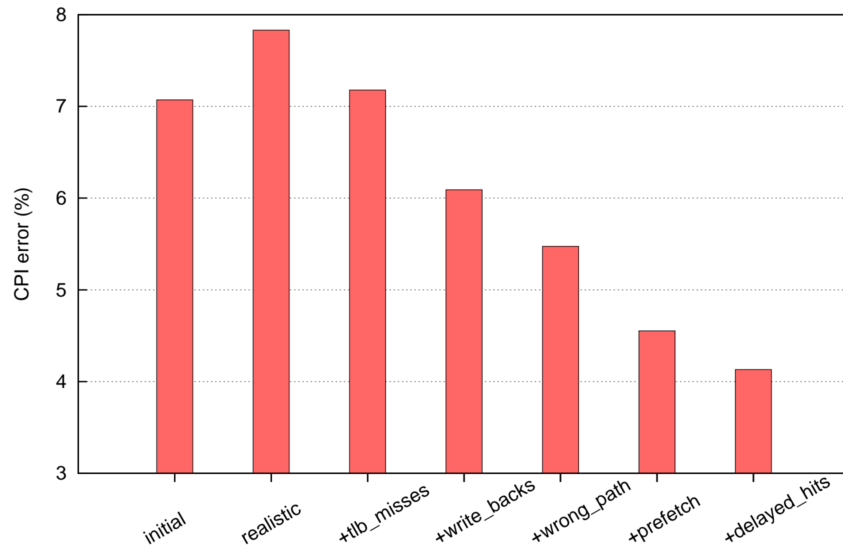


FIGURE A.1 – Nous efforts pour améliorer la précision du modèle PDCM.

précision de PDCM est dégradée. À partir de ce point nous avons commencé à améliorer la précision de PDCM tout en gardant les principes généraux de PDCM : la troisième barre inclut l'effet des défauts de TLB, la barre suivante considère l'effet des write-backs, la cinquième considère les requêtes sur les mauvais chemins des branchements mal prédits, la sixième prend en compte les requêtes de préchargement dans les caches de premier niveau, et enfin la dernière barre montre l'effet d'une modélisation plus précise des *delayed hits* (accès aux lignes de cache pour lesquelles un défaut de cache est déjà en cours).

De cette manière, l'erreur moyenne de 8 % avec la version originale PDCM est réduite à 4 % avec notre version améliorée de PDCM. Les nombres indiqués pour PDCM dans le reste de ce résumé concernent la version optimisée de PDCM.

PDCM est un modèle comportemental car il utilise une simulation détaillée pour estimer les temps d'exécution des instructions dans le pipeline. Parce que ces temps sont obtenus pour un cache L2 idéal, PDCM est généralement très précis lorsqu'il y a peu de défauts de cache L2.

Cependant, PDCM utilise une approche structurelle pour modéliser l'impact des défauts de cache L2 : PDCM suppose que modéliser le ROB et les dépendances de données est suffisant pour reproduire avec précision l'impact sur la performance des défauts de cache L2. Mais d'autres paramètres de la microarchitecture ont un impact sur la performance, telles que le nombre limité d'ALUs, le nombre de ports sur le cache de données de premier niveau, le nombre de MSHRs, la taille des tampons de micro-opérations, etc.

A.3.2 BADCO : un nouveau modèle comportemental

Considérant les limites de PDCM, nous proposons un nouveau modèle comportemental appelé BADCO par ses initiales en anglais. Contrairement à PDCM, qui analyse les dépendances explicitement au cours de la génération de la trace, BADCO utilise une approche comportementale pour trouver les dépendances entre requêtes. BADCO utilise deux simulations détaillées pour construire le modèle de coeur, contrairement à PDCM qui n'en utilise qu'une seule.

Au cours de la première simulation détaillée, la latence de toutes les requêtes au cache L2 est forcée à zéro. De cette première simulation, nous obtenons une trace T0. Ensuite, nous procédons à une deuxième simulation détaillée où nous attribuons une longue latence L à toutes les requêtes au cache L2. L est généralement supérieure à la latence réelle maximum d'une requête dans des conditions normales d'utilisation du modèle, par exemple $L = 1000$ cycles. De cette deuxième simulation, nous obtenons une trace TL. Les traces T0 et TL contiennent des informations temporelles pour chaque μop (micro-opération) retirée du ROB.

L'étape suivante, après l'obtention des traces T0 et TL, est de construire un modèle BADCO à partir de l'information contenue dans ces traces. La construction du modèle consiste à regrouper en *nœuds* les μops dépendant d'une même requête et à définir des dépendances entre nœuds. Les dépendances comprennent non seulement les dépendances de données, mais aussi les dépendances résultant des ressources d'exécution limitées, des branchements mal prédits, etc. Au lieu d'effectuer une analyse détaillée de ces dépendances lors de la génération de trace, nous trouvons les dépendances indirectement en analysant l'information temporelle dans la trace TL.

Les traces T0 et TL sont traitées simultanément, μop par μop, de façon séquentielle. Chaque μop est supposée dépendante d'une seule requête. Cette dépendance (déterminée à partir de la trace TL) et le fait que la μop porte ou non une requête déterminent si la μop commence un nouveau nœud ou si au contraire elle est attribuée à un nœud précédemment créé. Un nœud N_i représente un nombre S_i (taille du nœud) de μops. La somme de la taille de tous les nœuds, $\sum_i S_i$, est égale au nombre total de μops exécutées. Un nœud a aussi un *poids* de w_i cycles. Ce poids représente la somme des temps d'exécution des μops constituant le noeud lorsque les requêtes ont toutes une latence nulle (information provenant de la trace T0).

Une fois qu'un modèle BADCO est créé, il peut être réutilisé plusieurs fois, par exemple pour analyser les performances de différentes configurations de la hiérarchie mémoire, ou pour évaluer la performance d'un multi-coeur sur diverses combinaisons de benchmarks.

Un modèle BADCO est traité par ce que nous appelons une machine BADCO. Une machine BADCO est une machine asbtraite qui exécute des nœuds. Plus les noeuds comportent de μops, plus la vitesse de simulation de la machine BADCO est grande.

La machine BADCO distingue trois types de nœuds : nœud I, nœud L et nœud S. Un nœud I peut porter trois types de requêtes : défaut de cache IL1 (cache d'instructions

de premier niveau), défaut d'ITLB (cache de traduction d'adresse pour les instructions) et requête de préchargement du cache IL1. Un nœud L (ou nœud S) peut porter les requêtes liées à une μop load (ou store) : défaut de cache DL1 (cache de données de premier niveau), défaut de DTLB (cache de traduction d'adresse pour les données), requêtes de write-backs et requêtes de préchargement du cache DL1.

Durant la simulation du modèle, la machine BADCO charge les nœuds et les insère séquentiellement dans sa fenêtre. Les nœuds I émettent leurs requêtes vers la hiérarchie mémoire au moment du chargement, imitant le comportement du coeur réel. La fenêtre de la machine BADCO simule l'effet de la fenêtre d'instructions (ROB) du cœur réel. Ainsi, lorsque la somme de la taille des nœuds à l'intérieur de la fenêtre ne dépasse pas la taille du ROB, le nœud suivant peut être chargé. Sinon, le chargement du nœud est suspendu jusqu'à que la fenêtre de la machine BADCO dispose de l'espace suffisant pour insérer le nœud. Une fois dans la fenêtre, les nœuds peuvent être exécutés. Un nœud L émet ses requêtes à la hiérarchie mémoire lorsque le nœud dont il dépend a terminé son exécution. Un nœud L est considéré comme exécuté quand toutes les requêtes qu'il porte ont été traitées par la hiérarchie mémoire. Les autres types de nœuds sont considérés comme exécutés lorsque le nœud dont ils dépendent est lui-même exécuté. Les nœuds sont retirés de la fenêtre dans l'ordre dans lequel ils ont été insérés. Un nœud N_i est *prêt* à être retiré quand il est le nœud le plus ancien dans la fenêtre et qu'il a terminé son exécution. Le noeud est prêt à être retiré au temps t mais il est *effectivement* retiré w_i cycles après, c'est-à-dire au temps $t+w_i$. Une fois retirés de la fenêtre, les nœuds S sont envoyés dans une file post-retrait, imitant ce que fait un coeur réel. Les requêtes portées par un nœud S sont émises vers la hiérarchie mémoire à partir de la file post-retrait.

La machine BADCO modélise l'occupation des MSHRs associés aux caches de premier niveau. Ainsi, avant qu'une requête soit envoyée à la hiérarchie mémoire, la machine BADCO vérifie qu'il existe au moins une entrée MSHR libre. Dans le cas contraire, l'émission de la requête est différée en attendant qu'une entrée MSHR se libère.

A.3.3 Évaluation expérimentale

A.3.3.1 Précision simple cœur

Nous avons expérimentalement évalué et comparé la précision des modèles comportementaux : PDCM (version optimisée) et BADCO. L'évaluation est également faite qualitativement et quantitativement. Pour l'évaluation quantitative, nous avons utilisé trois configurations de cœur différentes : "petit", "moyen" et "gros" (voir le tableau 3.1). La configuration de la hiérarchie mémoire est la même pour les trois configurations de coeur. La figure A.2 montre l'erreur CPI de BADCO et PDCM sur 22 des 29 benchmarks SPEC CPU2006 et pour la configuration "gros" coeur. L'erreur CPI est calculée en utilisant comme référence le simulateur détaillé Zesto.

Parallèlement, le tableau A.1 présente l'erreur CPI moyenne de BADCO et PDCM pour les trois configurations de cœur. Les résultats montrent que BADCO est quanti-

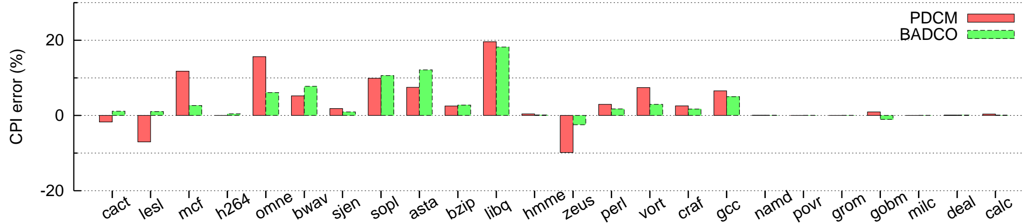


FIGURE A.2 – Erreur CPI de PDCM et BADCO par la configuration "gros".

	"petit"	**"moyen"**	**"gros"**
PDCM	3.8 %	4.0 %	4.7 %
BADCO	3.3 %	2.4 %	2.8 %

TABLE A.1 – Erreur CPI moyenne de PDCM et BADCO sur Zesto.

tativement plus précis que PDCM dans toutes les configurations de cœur évaluées.

Pour évaluer qualitativement les performances de BADCO et PDCM, nous définissons six configurations différentes de la hiérarchie mémoire, dans lesquelles nous faisons varier la taille des mémoires cache (L2 et LLC) et la bande passante vers la mémoire principale (voir le tableau 3.2). Parmi les six configurations, nous choisissons comme référence la configuration "001" et nous calculons la variation relative de performance (VRP) pour BADCO, PDCM et Zesto. Nous définission ensuite l'erreur qualitative comme $|VRP_{modele} - VRP_{zesto}|$. Le tableau A.2 montre l'erreur qualitative moyenne pour cinq configurations de la hiérarchie mémoire. Les résultats montrent que BADCO est également qualitativement plus précis que PDCM.

A.3.3.2 Précision multi-cœur

Utiliser BADCO pour évaluer la performance des architectures multi-cœurs avec des charges de travail multiprogrammées est très simple. La première étape consiste à créer un modèle BADCO pour chaque benchmark. Une fois cela fait, les modèles de cœur peuvent être facilement combinés pour simuler un processeur multi-cœur exécutant des programmes indépendants. L'étape suivante consiste à simuler une machine BADCO

	"000"	**"010"**	**"011"**	**"110"**	**"111"**
PDCM	4.6 %	4.0 %	1.3 %	4.1 %	1.2 %
BADCO	2.6 %	2.2 %	0.7 %	2.5 %	0.8 %

TABLE A.2 – Erreur qualitative moyenne de PDCM et BADCO sur la configuration "001".

pol. rempl.	2 cœurs	4 cœurs	8 cœurs
LRU	4.66 %	3.83 %	3.90 %
RANDOM	4.63 %	4.19 %	4.46 %
FIFO	4.79 %	4.10 %	4.33 %
DIP	4.54 %	4.01 %	3.99 %
DRRIP	4.35 %	3.75 %	3.77 %

Table A.3 – Erreur CPI moyenne absolue pour 2, 4 et 8 cœurs.

pol. rempl.	2 cœurs	4 cœurs	8 cœurs
RANDOM/LRU	0.89 %	0.76 %	1.34 %
FIFO/LRU	0.56 %	0.65 %	1.01 %
DIP/LRU	0.49 %	0.54 %	1.63 %
DRRIP/LRU	0.67 %	0.52 %	1.77 %

Table A.4 – Erreur de speedup moyenne absolue pour 2, 4 et 8 cœurs.

pour chaque cœur dans le processeur et de connecter les machines BADCO à un simulateur détaillé de la hiérarchie mémoire. Ainsi, chaque machine BADCO traite un unique modèle BADCO. Les machines BADCO envoyent des requêtes à la hiérarchie mémoire, qui informe en retour les machines BADCO lorsque les requêtes sont exécutées. Nous utilisons un arbitrage round-robin pour décider quelles machines BADCO accèdent à la hiérarchie mémoire à chaque cycle de simulation.

Afin d'évaluer l'exactitude de BADCO pour simuler des processeurs multi-cœurs (2, 4 et 8 cœurs), nous avons utilisé une étude de cas qui compare la performance de cinq politiques de remplacement du cache partagé : LRU, RANDOM (RND), FIFO, DIP et DRRIP. Les tableaux 3.6 et 3.7 décrivent en détail la configuration des cœurs et de la hiérarchie mémoire respectivement. Le tableau A.3 présente l'erreur CPI moyenne absolue pour chaque politique de remplacement. Parallèlement, le tableau A.4 montre l'erreur de speedup moyenne absolue en utilisant LRU comme configuration de référence. Les résultats montrent que BADCO est capable de quantifier finement les changements de performance et qu'il est raisonnablement précis pour estimer la performance brute.

A.3.3.3 Vitesse de simulation

La figure A.3 compare la vitesse de simulation de Zesto, PDCM et BADCO en millions d'instructions simulées par seconde (MIPS) pour chacun des 22 benchmarks évalués (SPEC CPU2006). L'accélération par rapport à Zesto varie entre un et deux ordres de grandeur. Cependant, PDCM est légèrement plus rapide que BADCO. Cela est principalement dû à la plus grande granularité de PDCM (90 μops en moyenne par élément de trace) par rapport à BADCO (50 uops en moyenne par nœud).

Le tableau A.5 compare la vitesse moyenne de simulation de BADCO et Zesto pour des processeurs de 1, 2, 4 et 8 cœurs. Le tableau A.5 donne également l'accélération

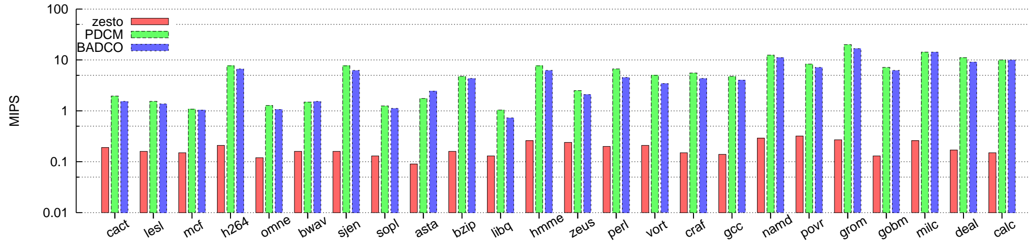


FIGURE A.3 – Vitesse de simulation en MIPS pour Zesto, PDCM et BADCO.

Number of cores	1	2	4	8
MIPS - Zesto	0.170	0.096	0.049	0.017
MIPS - BADCO	2.52	2.41	1.89	1.19
Speedup	14.8	25.19	38.88	68.1

TABLE A.5 – Vitesse de simulation moyenne de BADCO et Zesto pour des processeurs de 1, 2, 4 et 8 cœurs.

relative de BADCO par rapport à Zesto. Les résultats montrent que la vitesse de simulation diminue lorsque le nombre de cœurs augmente. Une des causes de la réduction de la vitesse de simulation est l'augmentation de l'empreinte mémoire du simulateur. Cependant, parce que la vitesse de simulation diminue plus rapidement pour Zesto que pour BADCO, l'accélération relative de BADCO par rapport à Zesto augmente avec le nombre de cœurs simulés.

A.4 Sélection de charges de travail multiprogrammées

Pour évaluer les performances d'une architecture multi-cœur sur des charges multiprogrammées, une procédure habituelle consiste à prendre des benchmarks séquentiels et à simuler des combinaisons de ces benchmarks. La micro-architecture est évaluée sur de telles combinaisons de benchmarks. Nous appelons charge de travail multiprogrammées une combinaison de K benchmarks, où K est le nombre de cœurs logiques. La population totale des charges de travail multiprogrammées qui peuvent être obtenus à partir de B benchmarks est donnée par $\binom{B+K-1}{K}$. La population est dans la plupart des cas trop grande pour être évaluée complètement. Pour cette raison, les architectes de microprocesseurs choisissent en général un échantillon de W charges de travail, où W vaut typiquement quelques dizaines. L'échantillon de charges de travail est généralement beaucoup plus petit que la population totale. Cependant il n'existe pas dans la communauté de la microarchitecture de méthode standard pour définir un échantillon représentatif. Or la méthode utilisée pour sélectionner l'échantillon peut avoir un impact important sur les résultats d'une étude.

Nous avons effectué une enquête sur les articles publiés dans les trois principales conférences en architecture des ordinateurs, ISCA, MICRO et HPCA, de 2007 jusqu'à Mars 2012. L'enquête a cherché à établir quelles sont les pratiques utilisées pour sélectionner des charges de travail multiprogrammées. Nous avons constaté que sur 75 articles concernés, seulement 9 d'entre eux utilisent un échantillonnage aléatoire pour sélectionner les charges de travail. Les 66 autres articles définissent des classes de benchmarks, puis ils définissent des charges de travail à partir de ces classes. Dans la plupart des cas, les classes sont définies manuellement par les auteurs en fonction de leur compréhension du problème étudié. Le nombre de charges de travail et la méthode de sélection sont dans la plupart des cas arbitraires.

Dans cette thèse, nous évaluons l'efficacité de méthodes d'échantillonnage différentes pour produire des échantillons représentatifs.

A.4.1 Méthodes d'échantillonnage

A.4.1.1 Échantillonnage aléatoire simple

L'échantillonnage aléatoire simple exige que toutes les charges de travail aient la même probabilité d'être sélectionnées. Comme nous l'avons indiqué précédemment, ce type d'échantillonnage n'est pas le plus utilisé pour définir des charges de travail. La plupart des auteurs préfèrent utiliser un petit échantillon, qu'ils cherchent à définir de manière à ce qu'il soit représentatif. L'échantillonnage aléatoire est un moyen sûr d'éviter tout biais, pourvu que l'échantillon soit suffisamment grand. En outre, l'échantillonnage aléatoire simple peut être modélisé analytiquement afin d'estimer le degré de confiance de l'échantillon ou pour déterminer la taille d'un échantillon représentatif. Cette thèse présente un modèle analytique pour estimer la probabilité de juger correctement laquelle de deux microarchitectures est la meilleure.

A.4.1.2 Échantillonnage aléatoire équilibré

Si nous considérons la population de charges de travail et si nous comptons le nombre d'occurrences de chaque benchmark, nous constatons que tous les benchmarks ont le même poids. Ceci est cohérent avec l'hypothèse implicite que tous les benchmarks sont également importants. L'échantillonnage aléatoire équilibré garantit que tous les benchmarks ont le même nombre d'occurrences dans l'échantillon. Ainsi, après avoir sélectionné une première charge de travail, les autres charges de travail n'ont plus la même probabilité d'être sélectionnées. Cela rend difficile la modélisation analytique de ce type d'échantillonnage.

A.4.1.3 Échantillonnage aléatoire stratifié

L'échantillonnage aléatoire stratifié exploite le fait que la population des charges de travail est généralement hétérogène. Le but de cette méthode d'échantillonnage est de

diviser la population en strates ayant des caractéristiques plus homogènes que la population globale, et de prendre dans chaque strate un échantillon aléatoire. Par exemple, supposons qu'une microarchitecture Y soit plus performante qu'une microarchitecture X sur 80 % de la population, tandis que X est plus performante que Y sur les 20 % restant. La connaissance de ces sous-ensembles (strates) permet de définir un échantillon représentatif avec moins de charges de travail.

Il y a différentes façons de définir les strates, et de cela dépend l'efficacité de la méthode d'échantillonnage aléatoire stratifié. Dans cette thèse, nous analysons deux stratégies pour définir des strates : *stratification des benchmarks* et *stratification des charges de travail*.

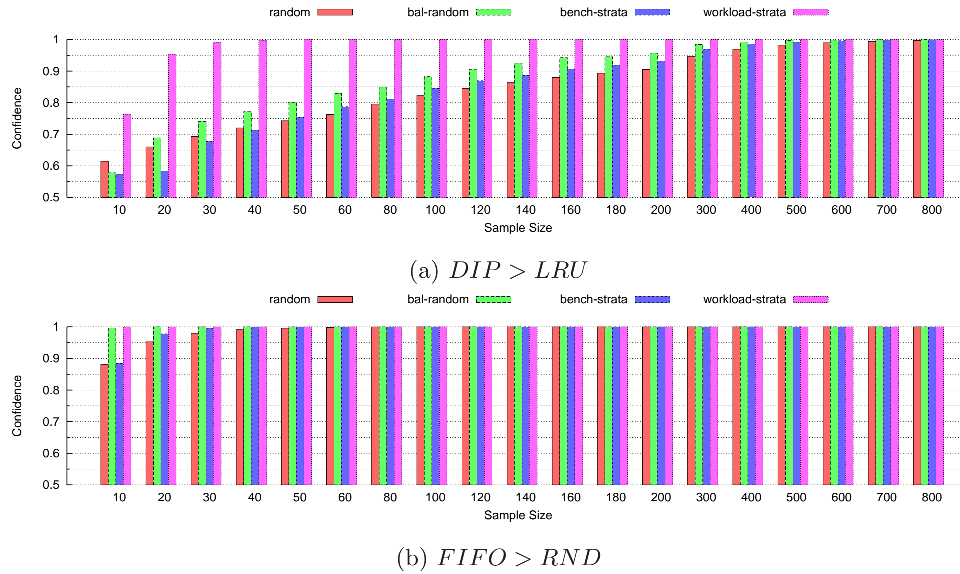
Stratification des benchmarks. Diviser les benchmarks en classes afin de définir des charges de travail multiprogrammées est une pratique courante dans les études en microarchitecture. L'hypothèse principale est que les benchmarks d'une même classe ont un comportement similaire. Les classes des benchmarks à elles seules ne constituent pas des strates, mais il est possible de construire des strates à partir du nombre d'occurrences de chaque classe dans les charges de travail. Par exemple, les charges de travail composées de benchmarks appartenant tous à une classe donnée forment une strate.

Le méthode de stratification des benchmarks consiste à créer des strates à partir des classes de benchmarks définies par le chercheur ou l'ingénieur, puis à sélectionner pour chaque strate un échantillon aléatoire d'une ou plusieurs charges de travail. Il est important de noter que la méthode de stratification des benchmarks décrite dans cette thèse est une tentative de formalisation de pratiques qui sont diverses et pas toujours explicites.

Stratification des charges de travail. La méthode de stratification des charges de travail utilise des simulations approximatives rapides afin d'évaluer un grand échantillon comportant des milliers de charges de travail. Une fois que les charges de travail de ce grand échantillon ont été simulées, il est possible d'utiliser des algorithmes de clustering pour définir des strates. Parce que notre objectif est de définir des échantillons représentatifs permettant de bien évaluer laquelle de deux microarchitectures est la meilleure, la stratification des charges de travail est faite à partir de la distribution sur les charges de travail des différences de performance entre les deux microarchitectures. Une fois les charges de travail divisées en strates, nous prenons dans chaque strate un échantillon aléatoire d'une ou plusieurs charges de travail.

A.4.2 Évaluation expérimentale

La figure A.4 compare l'efficacité des méthodes d'échantillonnage étudiées. Nous avons utilisé une étude de cas qui compare la performance de cinq politiques de remplacement du cache partagé : LRU, RANDOM (RND), FIFO, DIP et DRRIP. En particulier, la figure A.4 montre les résultats pour un processeur à quatre cœurs et en utilisant



(a) $DIP > LRU$

(b) $FIFO > RND$

FIGURE A.4 – Efficacité des méthodes d'échantillonnage. Confiance en fonction de la taille de l'échantillon par $DIP > LRU$ et $FIFO > RND$.

l'IPCT comme métrique de performance. Chaque groupe de quatre barres correspond à une certaine taille d'échantillon. La première barre montre la confiance théorique pour l'échantillonnage aléatoire simple (random). La deuxième barre montre la confiance expérimentale pour l'échantillonnage aléatoire équilibré (bal-random). La troisième barre montre la confiance expérimentale pour l'échantillonnage aléatoire stratifié par benchmark (bench-strata). Enfin, la dernière barre montre la confiance expérimentale pour l'échantillonnage aléatoire stratifié par charge de travail (workload-strata).

Les résultats obtenus indiquent que, parmi les méthodes d'échantillonnage étudiées, la méthode de stratification par charge de travail est la plus effective pour définir des charges de travail multiprogrammées. La seconde méthode la plus effective est l'échantillonnage aléatoire équilibré. On peut noter que lorsque la différence de performance est très petite, comme entre DIP et LRU, toutes les méthodes d'échantillonage nécessitent un grand échantillon, à l'exception de la méthode de stratification des charges de travail.

A.5 Conclusions

Dans cette thèse, nous avons montré que, parmi tous les outils de simulation à la disposition d'un architecte de microprocesseurs, les modèles comportementaux sont intéressants pour étudier la hiérarchie mémoire des processeur multi-coeurs. Nous avons démontré que l'utilisation de modèles comportementaux permet d'accélérer les simula-

tions d'un facteur entre un et deux ordres de grandeur. Dans cette thèse, nous avons évalué et comparé deux modèles comportementaux : PDCM et BADCO. Nous avons montré que notre version optimisée de PDCM peut modéliser un cœur superscalaire avec des erreurs moyennes de moins de 5 %. Dans cette thèse, nous avons proposé BADCO, un nouveau modèle comportemental qui présente une plus grande précision que PDCM, avec des erreurs moyennes de moins de 3,5 % pour les microarchitectures à un seul cœur, et des erreurs de moins de 5 % pour les microarchitectures multi-cœurs étudiées.

Dans le domaine des nouvelles méthodologies de simulation pour systèmes multi-cœurs, nous avons abordé le problème de la sélection des charges de travail multiprogrammées pour évaluer la performance des microarchitectures multi-cœurs. Dans ce travail, nous avons souligné la difficulté d'évaluer la représentativité d'un échantillon sans simuler un plus grand nombre de charges de travail. Nous avons abordé ce problème en utilisant la simulation approximative avec des modèles comportementaux. Grâce à leur excellente combinaison de vitesse et de précision, les modèles comportementaux permettent d'évaluer la performance d'une microarchitecture sur un grand échantillon de plusieurs milliers de charges de travail.

Nous avons proposé une méthode alternative qui définit le degré de confiance d'un échantillon aléatoire comme la probabilité d'obtenir une conclusion correcte lorsqu'on compare deux microarchitectures afin de d'eterminer laquelle est la meilleure. Dans ce travail, nous avons également comparé différentes méthodes d'échantillonnage En particulier, nous avons proposé la méthode de stratification des charges de travail, qui s'avère être très efficace dans la réduction de la taille de l'échantillon dans les situations où l'échantillonnage aléatoire nécessite un grand échantillon. La stratification des charges de travail utilise la simulation approximative rapide pour définir des échantillons représentatifs.

D'autres questions intéressantes se posent pour de futurs travaux. En particulier, nous pensons que les modèles de type BADCO peuvent être étendus pour simuler certaines applications parallèles, pour simuler des systèmes hétérogènes multi-cœurs, et pour estimer la consommation d'énergie.

En ce qui concerne la sélection des charges de travail multiprogrammées, cette étude soulève plusieurs questions qui nécessitent une analyse plus approfondie. Il est nécessaire d'étudier les moyens pour définir la confiance dans un échantillon, par exemple pour obtenir des échantillons qui garantissent que le speedup est dans une certaine marge d'erreur. Ainsi, la question se pose : quelle est la manière la plus appropriée pour mesurer la représentativité d'un échantillon ? D'autres questions restent ouverte : existe-t-il d'autres algorithmes de stratification qui permettraient d'augmenter l'efficacité de la méthode de stratification des charges de travail ? Et comment prendre en compte l'effet des phases d'exécution des programmes dans la sélection d'un échantillon représentatif ?

Bibliography

[1] A. Alameldeen and D. Wood, "Ipc considered harmful for multiprocessor workloads," *Micro, IEEE*, vol. 26, no. 4, pp. 8–17, 2006.

[2] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: infrastructure for full system simulation," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 1, pp. 52–61, 2009.

[3] T. Austin, E. Larson, and D. Ernst, "SimpleScalar : an infrastructure for computer system modeling," *IEEE Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002, http://www.simplescalar.com.

[4] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," in *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2. IEEE Computer Society, 2005, pp. 506–517.

[5] B. Barnes and J. Slice, "Simnow: A fast and functionally accurate amd x86-64 system simulator," in *Tutorial at the IEEE International Workload Characterization Symposium*, 2005.

[6] F. Bellard, "Qemu, a fast and portable dynamic translator." USENIX, 2005.

[7] C. Bienia, S. Kumar, J. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[8] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, "The M5 simulator: Modeling networked systems," *Micro, IEEE*, vol. 26, no. 4, pp. 52–60, 2006.

[9] S. Blackburn, R. Garner, C. Hoffmann, A. Khang, K. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer *et al.*, "The dacapo benchmarks: Java benchmarking development and analysis," in *ACM SIGPLAN Notices*, vol. 41, no. 10. ACM, 2006, pp. 169–190.

[10] R. Carl and J. Smith, “Modeling superscalar processors via statistical simulation,” in *Workshop on Performance Analysis and Its Impact on Design*, 1998.

[11] T. E. Carlson, W. Heirman, and L. Eeckhout, “Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation,” in *Proc. of Supercomputing 2011*, 2011.

[12] X. E. Chen and T. M. Aamodt, “Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs,” in *Proc. of the 41st Int. Symp. on Microarchitecture*, 2008.

[13] D. Chiou, H. Angepat, N. Patil, and D. Sunwoo, “Accurate functional-first multicore simulators,” *Computer Architecture Letters*, vol. 8, no. 2, pp. 64–67, 2009.

[14] D. Chiou, D. Sunwoo, J. Kim, N. Patil, W. Reinhart, D. Johnson, J. Keefe, and H. Angepat, “Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators,” in *Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 249–261.

[15] S. Cho, S. Demetriades, S. Evans, L. Jin, H. Lee, K. Lee, and M. Moeng, “TPTS: A novel framework for very fast manycore processor architecture simulation,” in *Parallel Processing, 2008. ICPP’08. 37th International Conference on*. IEEE, 2008, pp. 446–453.

[16] D. Citron, “Misspeculation: partial and misleading use of spec cpu2000 in computer architecture conferences,” in *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2. ACM, 2003, pp. 52–61.

[17] W. G. Cochran, *Sampling Techniques, 3rd Edition*, 2nd ed. John Wiley, 1977.

[18] T. M. Conte *et al.*, “Systematic computer architecture prototyping,” Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[19] T. Conte, M. Hirsch, and W. Hwu, “Combining trace sampling with single pass methods for efficient cache simulation,” *Computers, IEEE Transactions on*, vol. 47, no. 6, pp. 714–720, 1998.

[20] K. V. Craeynest and L. Eeckhout, “The multi-program performance model : debunking current practice in multi-core simulation,” in *Proc. of the IEEE International Symposium on Workload Characterization*, 2011.

[21] H. Cragon, *Computer architecture and implementation*. Cambridge University Press, 2000.

[22] C. Dubach, T. Jones, and M. O'Boyle, "Microarchitectural design space exploration using an architecture-centric approach," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 2007, pp. 262–271.

[23] M. Durbhakula, V. S. Pai, and S. Adve, "Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors," in *Proc. of the 5th Int. Symp. on High-Performance Computer Architecture*, 1999.

[24] L. Eeckhout, "Computer architecture performance evaluation methods," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–145, 2010.

[25] L. Eeckhout, K. De Bosschere, and H. Neefs, "Performance analysis through synthetic trace generation," in *Performance Analysis of Systems and Software, 2000. ISPASS. 2000 IEEE International Symposium on.* IEEE, 2000, pp. 1–6.

[26] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Designing computer architecture research workloads," *Computer*, vol. 36, no. 2, pp. 65–71, 2003.

[27] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Quantifying the impact of input data sets on program behavior and its applications," *Journal of Instruction-Level Parallelism*, vol. 5, no. 1, pp. 1–33, 2003.

[28] L. Eeckhout, R. H. Bell Jr, B. Stougie, K. De Bosschere, and L. K. John, "Control flow modeling in statistical simulation for accurate and efficient processor design studies," in *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on.* IEEE, 2004, pp. 350–361.

[29] J. Emer and D. Clark, "A characterization of processor performance in the vax-11/780," in *ACM SIGARCH Computer Architecture News*, vol. 12, no. 3. ACM, 1984, pp. 301–310.

[30] B. Everitt, S. Landau, M. Leese, and D. Stahl, *Cluster analysis.* John Wiley & Sons, Ltd, 2011.

[31] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, May 2008.

[32] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 2, p. 3, 2009.

[33] S. Eyerman, J. E. Smith, and L. Eeckhout, "Characterizing the branch misprediction penalty," in *Proc. of the Int. Symp. on Performance Analysis of Systems and Software*, 2011.

[34] B. Fields, S. Rubin, and R. Bodik, "Focusing processor policies via critical-path prediction," in *Proc. of the 28th Int. Symp. on Computer Architecture*, 2001.

[35] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn, "Using interaction costs for microarchitectural bottleneck analysis," in *Proc. of the 36th Int. Symp. on Microarchitecture*, 2003.

[36] P. J. Fleming and J. J. Wallace, "How not to lie with statistics : the correct way to summarize benchmark results," *Communications of the ACM*, vol. 29, no. 3, pp. 218–221, Mar. 1986.

[37] D. Genbrugge and L. Eeckhout, "Statistical simulation of chip multiprocessors running multi-program workloads," in *Computer Design, 2007. ICCD 2007. 25th International Conference on.* IEEE, 2007, pp. 464–471.

[38] D. Genbrugge, L. Eeckhout, and K. De Bosschere, "Accurate memory data flow modeling in statistical simulation," in *Proceedings of the 20th annual international conference on Supercomputing.* ACM, 2006, pp. 87–96.

[39] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on.* IEEE, 2010, pp. 1–12.

[40] S. R. Goldschmidt and J. L. Hennessy, "The accuracy of trace-driven simulations of multiprocessors," in *Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 1993.

[41] R. Gupta, B. Calder, J. Lau, and C. Pereira, "Dynamic phase analysis for cycle-close trace generation," in *Hardware/Software Codesign and System Synthesis, 2005. CODES+ ISSS'05. Third IEEE/ACM/IFIP International Conference on.* IEEE, 2005, pp. 321–326.

[42] N. Hardavellas, S. Somogyi, T. Wenisch, R. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. Hoe, and A. Nowatzyk, "Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, pp. 31–34, 2004.

[43] J. Hennessy and D. Patterson, *Computer architecture: a quantitative approach.* Morgan Kaufmann, 2011.

[44] J. Henning, "Spec cpu2000: Measuring cpu performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[45] E. İpek, S. McKee, R. Caruana, B. de Supinski, and M. Schulz, *Efficiently exploring architectural design spaces via predictive modeling.* ACM, 2006, vol. 40, no. 5.

[46] A. Jaleel, K. Theobald, S. C. Steely Jr., and J. Emer, "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)," in *Proc. of the 37th Annual International Symposium on Computer Architecture*, 2010.

[47] L. K. John, "More on finding a single number to indicate overall performance of a benchmark suite," *ACM SIGARCH Computer Architecture News*, vol. 32, no. 1, Mar. 2004.

[48] R. Johnson and D. Wichern, *Applied multivariate statistical analysis.* Prentice hall Englewood Cliffs, NJ, 1992, vol. 4.

[49] P. Joseph, K. Vaswani, and M. Thazhuthaveetil, "Construction and use of linear regression models for processor performance analysis," in *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on.* IEEE, 2006, pp. 99–108.

[50] ——, "A predictive performance model for superscalar processors," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 2006, pp. 161–170.

[51] A. Joshi, J. Yi, R. Bell Jr, L. Eeckhout, L. John, and D. Lilja, "Evaluating the efficacy of statistical simulation for design space exploration," in *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on.* IEEE, 2006, pp. 70–79.

[52] S. Kanaujia, I. E. Papazian, J. Chamberlain, and J. Baxter, "FastMP : a multi-core simulation methodology," in *Workshop on Modeling, Benchmarking and Simulation*, 2006.

[53] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proc. of the 31st Int. Symp. on Computer Architecture*, 2004.

[54] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on.* IEEE, 2003, pp. 81–92.

[55] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, vol. 38, no. 11, pp. 32–38, 2005.

[56] S. Laha, J. H. Patel, and R. K. Iyer, "Accurate low-cost methods for performance evaluation of cache memory systems," *Computers, IEEE Transactions on*, vol. 37, no. 11, pp. 1325–1336, 1988.

[57] B. Lee and D. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5. ACM, 2006, pp. 185–194.

[58] B. Lee, D. Brooks, B. de Supinski, M. Schulz, K. Singh, and S. McKee, “Methods of inference and learning for performance modeling of parallel applications,” in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming.* ACM, 2007, pp. 249–258.

[59] C. Lee, M. Potkonjak, and W. Mangione-Smith, “Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems,” in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture.* IEEE Computer Society, 1997, pp. 330–335.

[60] K. Lee and S. Cho, “In-N-Out : reproducing out-of-order superscalar processor behavior from reduced in-order traces,” in *Proc. of the IEEE Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2011.

[61] K. Lee, S. Evans, and S. Cho, “Accurately approximating superscalar processor performance from traces,” in *Proc. of the Int. Symp. on Performance Analysis of Systems and Software*, 2009.

[62] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron, “CMP design space exploration subject to physical constraints,” in *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on.* IEEE, 2006, pp. 17–28.

[63] G. Loh, “A time-stamping algorithm for efficient performance estimation of superscalar processors,” in *Proc. of the ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems*, 2001.

[64] G. Loh, S. Subramaniam, and Y. Xie, “Zesto : a cycle-level simulator for highly detailed microarchitecture exploration,” in *Proc. of the Int. Symp. on Performance Analysis of Systems and Software*, 2009.

[65] K. Luo, J. Gummaraju, and M. Franklin, “Balancing throughput and fairness in SMT processors,” in *Proc. of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2001.

[66] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A full system simulation platform,” *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[67] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood, “Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset,” *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[68] J. R. Mashey, “War of the benchmark means : time for a truce,” *ACM SIGARCH Computer Architecture News*, vol. 32, no. 4, Sep. 2004.

[69] C. J. Mauer, M. D. Hill, and D. A. Wood, "Full-system timing-first simulation," in *Proc. of the ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems*, 2002.

[70] P. Michaud, "Demystifying multicore throughput metrics," Aug. 2012.

[71] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on.* IEEE, 2008, pp. 35–46.

[72] J. Moses, R. Illikkal, R. Iyer, R. Huggahalli, and D. Newell, "ASPEN: towards effective simulation of threads & engines in evolving platforms," in *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on.* IEEE, 2004, pp. 51–58.

[73] O. Mutlu, H. Kim, D. Armstrong, and Y. Patt, "Understanding the effects of wrong-path memory references on processor performance," in *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture.* ACM, 2004, pp. 56–64.

[74] D. Nellans, V. Kadaru, and E. Brunvand, "ASIM-An asynchronous architectural level simulator," in *Proceedings of GLSVLSI.* Citeseer, 2004.

[75] D. B. Noonburg and J. P. Shen, "Theoretical modeling of superscalar processor performance," in *Proc. of the 27th Int. Symp. on Microarchitecture*, 1994.

[76] S. Nussbaum and J. Smith, "Modeling superscalar processors via statistical simulation," in *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on.* IEEE, 2001, pp. 15–24.

[77] ——, "Statistical simulation of symmetric multiprocessor systems," in *Simulation Symposium, 2002. Proceedings. 35th Annual.* IEEE, 2002, pp. 89–97.

[78] P. Ortego and P. Sack, "SESC: SuperESCalar simulator," in *17 th Euro micro conference on real time systems (ECRTS'05)*, 2004, pp. 1–4.

[79] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1. ACM, 2003, pp. 318–319.

[80] R. Plackett and J. Burman, "The design of optimum multifactorial experiments," *Biometrika*, pp. 305–325, 1946.

[81] M. Qureshi, A. Jaleel, Y. Patt, S. C. Steely Jr., and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. of the 34th Annual International Symposium on Computer Architecture*, 2007.

[82] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero, "Trace-driven simulation of multithreaded applications," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on.* IEEE, 2011, pp. 87–96.

[83] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation: The simos approach," *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 3, no. 4, pp. 34–43, 1995.

[84] F. Ryckbosch, S. Polfliet, and L. Eeckhout, "Fast, accurate, and validated full-system software simulation on x86 hardware," *IEEE Micro*, vol. 30, no. 6, pp. 46–56, Nov. 2010.

[85] R. Sendag, A. Yilmazer, J. Yi, and A. Uht, "Quantifying and reducing the effects of wrong-path memory references in cache-coherent multiprocessor systems," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International.* IEEE, 2006, pp. 10–pp.

[86] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[87] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2, pp. 336–349, 2003.

[88] J. E. Smith, "Characterizing computer performance with a single number," *Communications of the ACM*, vol. 31, no. 10, pp. 1202–1206, Oct. 1988.

[89] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for simultaneous multithreading processor," in *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[90] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood, "Analytic evaluation of shared-memory systems with ILP processors," in *Proc. of the 25th Int. Symp. on Computer Architecture*, 1998.

[91] C. Spradling, "Spec cpu2006 benchmark tools," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 130–134, 2007.

[92] M. Vachharajani, N. Vachharajani, D. Penry, J. Blome, and D. August, "The liberty simulation environment, version 1.0," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, pp. 19–24, 2004.

[93] M. Van Biesbrouck, L. Eeckhout, and B. Calder, "Considering all starting points for simultaneous multithreading simulation," in *Proc. of the Int. Symp. on Performance Analysis of Systems and Software*, 2006.

[94] ——, "Representative multiprogram workloads for multithreaded processor simulation," in *Proc. of the IEEE International Symposium on Workload Characterization*, 2007.

[95] H. Vandierendonck and A. Seznec, "Fairness metrics for multi-threaded processors," *IEEE Computer Architecture Letters*, vol. 10, no. 1, Jan. 2011.

[96] ——, "Managing SMT Resource Usage through Speculative Instruction Window Weighting," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 3, Oct. 2011.

[97] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on.* IEEE, 2003, pp. 84–95.

[98] J. Yi, L. Eeckhout, D. Lilja, B. Calder, L. John, and J. Smith, "The future of simulation: A field of dreams," *Computer*, vol. 39, no. 11, pp. 22–29, 2006.

[99] J. Yi, D. Lilja, and D. Hawkins, "A statistically rigorous approach for improving simulation methodology," in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on.* IEEE, 2003, pp. 281–291.

[100] M. Yourst, "Ptlsim users guide and reference: The anatomy of an x86-64 out of order microprocessor," Technical report, SUNY Binghamton, Tech. Rep.

[101] ——, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on.* IEEE, 2007, pp. 23–34.

[102] L. Zhao, R. Iyer, J. Moses, R. Illikkal, S. Makineni, and D. Newell, "Exploring large-scale CMP architectures using ManySim," *IEEE Micro*, vol. 27, no. 4, pp. 21–33, Jul. 2007.

List of Figures

2.1 Scientific method versus computer systems method. 10

3.1 Normalized execution time for h264ref and libquantum 32
3.2 Simulation flow for PDCM behavioral core model. 33
3.3 Adapting PDCM for detailed OoO core 35
3.4 Simulation flow for BADCO model. 39
3.5 Example of BADCO model building 40
3.6 CPI error: PDCM vs. BADCO 46
3.7 Relative performance variation: PDCM vs. BADCO 48
3.8 BADCO simulation speed 49
3.9 CPI measured with Zesto vs. estimated CPI with BADCO 53

4.1 Degree of confidence as a function of $\frac{1}{c_v}\sqrt{\frac{W}{2}}$ 59
4.2 Confidence that *"DRRIP outperforms DIP"* 60
4.3 Inv. coef. of variation for 4 cores, IPCT vs. WSU vs. HSU 62
4.4 Inv. coef. of variation measured with BADCO for 2, 4 and 8 cores . . . 63
4.5 Confidence of sampling methods as function of sample size. 71
4.6 Experimental degree of confidence measured with Zesto. 72

A.1 Nous efforts pour améliorer la précision du modèle PDCM. 81
A.2 Erreur CPI de PDCM et BADCO par la configuration "gros". 84
A.3 Vitesse de simulation en MIPS pour Zesto, PDCM et BADCO. 86
A.4 Efficacité des méthodes d'échantillonnage. Confiance en fonction de la taille de l'échantillon par $DIP > LRU$ et $FIFO > RND$. 89

List of Tables

3.1 Core configuration – single core experiment 43
3.2 Uncore configuration – single core experiment 44
3.3 Average CPI error of PDCM and BADCO respect to Zesto. 45
3.4 Average variation error using as reference the configuration “001”. 47
3.5 Single core simulation speed. 47
3.6 Core configuration – multicore experiment. 50
3.7 Uncore configurations – multicore experiment. 51
3.8 Average of absolute CPI error in percentage. 52
3.9 Average of absolute *speedup error* in percentage. 52
3.10 BADCO average speedup for 1, 2, 4 and 8 cores. 54

4.1 Classification of SPEC benchmarks by memory intensity 66

A.1 Erreur CPI moyenne de PDCM et BADCO sur Zesto. 84
A.2 Erreur qualitative moyenne de PDCM et BADCO sur la configuration “001”. 84
A.3 Erreur CPI moyenne absolue pour 2, 4 et 8 cœurs. 85
A.4 Erreur de speedup moyenne absolue pour 2, 4 et 8 cœurs. 85
A.5 Vitesse de simulation moyenne de BADCO et Zesto pour des processeurs de 1, 2, 4 et 8 cœurs. 86

Résumé

Ces dernières années, l'effort de recherche est passé de la microarchitecture du cœur à la microarchitecture de la hiérarchie mémoire. Les modèles précis au cycle près pour processeurs multi-cœurs avec des centaines de cœurs ne sont pas pratiques pour simuler des charges multitâches réelles du fait de la lenteur de la simulation. Un grand pourcentage du temps de simulation est consacré à la simulation des différents cœurs, et ce pourcentage augmente linéairement avec chaque génération de processeur. Les modèles approximatifs sacrifient de la précision pour une vitesse de simulation accrue, et sont la seule option pour certains types de recherche. Les processeurs multi-cœurs exigent également des méthodes de simulation plus rigoureuses. Il existe plusieurs méthodes couramment utilisées pour simuler les architectures simple cœur. De telles méthodes doivent être adaptées ou même repensées pour la simulation des architectures multi-cœurs.

Dans cette thèse, nous avons montré que les modèles comportementaux sont intéressants pour étudier la hiérarchie mémoire des processeurs multi-coeurs. Nous avons démontré que l'utilisation de modèles comportementaux permet d'accélérer les simulations d'un facteur entre un et deux ordres de grandeur avec des erreurs moyennes de moins de 5 %. Nous avons démontré également que des modèles comportementaux peuvent aider dans le problème de la sélection des charges de travail multiprogrammées pour évaluer la performance des microarchitectures multi-cœurs.

Abstract

In recent years, the research focus has moved from core microarchitecture to *uncore* microarchitecture. Cycle-accurate models for many-core processors featuring hundreds or even thousands of cores are out of reach for simulating realistic workloads. A large portion of the simulation time is spend in the cores, and it is this portion that grows linear with every processor generation. Approximate simulation methodologies, which trade off accuracy for simulation speed, are necessary for conducting certain research. Multicore processors also demand for more advanced and rigorous simulation methodologies. Many popular methodologies designed by computer architects for simulation of single core architectures must be adapted or even rethought for simulation of multicore architectures.

In this thesis, we have shown that behavioral core modeling is a competitive option for multicore studies where the research focus is in the *uncore* microarchitecture and considering independent tasks. We demonstrated that behavioral core models can bring speedups between one and two orders of magnitude with average CPI errors of less than 5 %. We have also demonstrated that behavioral core models can help in the problem of selecting multiprogram workloads for the evaluation of multicore throughput.