PERFORMANCE SIMULATION METHODOLOGIES FOR HARDWARE/SOFTWARE CO-DESIGNED PROCESSORS

Aleksandar Brankovic
Departament d'Arquitectura de Computadors
Universitat Politecnica de Catalunya

Advisor:

Kyriakos Stavrou

Co-Advisors:

Enric Gibert Codina

Antonio González Colas

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy
Programa de Doctorat: Arquitectura de Computadors
January, 2015

Acknowledgments

This is to express my gratitude to the UPC committee and the Government of Catalonia for providing me with the necessary resources that allowed me to perform my PhD studies in a highly competitive, but at the same time friendly international atmosphere. A special thank goes to Prof. Antonio Gozalez and ARCO research group for giving me an opportunity to work in the challenging field of HW/SW co-designed processors. I would further like to thank my advisors Kyriakos Stavrou and Enric Gibert. I really appreciate their valuable guidance, constructive criticism and the time and concentration they spent reading all the versions of this thesis. Many thanks go to all my current and past colleagues, with whom I spent last four years (Gaurang, Rakesh, Maria, Demos, Marti, Javi, Jose, Jose Maria, Gem, Xavi, Zoran, Enrique, Shrikanth, Manish, Indu, Abhishek, Marc, Enric). Maybe the best achievement in Barcelona is getting to know them. Furthermore a special thank goes to my flatmates (Milan, Milovan, Nikola, Darko) for encouraging me during my PhD time. I wish to sincerely thank my family (my mother Slavica and my father Momcilo) for supporting me, morally above all, but also financially, and sharing all the useful life experience.

Aleksandar Brankovic

Abstract

Recently the community started looking into Hardware/Software (HW/SW) co-designed processors as potential solutions to move towards the less power consuming and the less complex designs. Unlike other solutions, they reduce the power and the complexity doing so called dynamic binary translation and optimization from a guest ISA to an internal host custom ISA. This thesis tries to answer the question on how to simulate this kind of architectures.

For any kind of processor's architecture, the simulation is the common practice, because it is impossible to build several versions of hardware in order to try all alternatives. The simulation of HW/SW co-designed processors has a big issue in comparison with the simulation of traditional HW-only architectures. First of all, open source tools do not exist. Therefore researchers many times assume that the software layer overhead, which is in charge for dynamic binary translation and optimization, is constant or ignored. In this thesis we show that such an assumption is not valid and that can lead to very inaccurate results. Consequently, including the software layer in the simulation is a must.

On the other side, the simulation is very slow in comparison to native execution, so the community spent a big effort on delivering accurate results in a reasonable amount of time. Therefore it is the common practice for HW-only processors that only parts of application stream, which are called samples, are simulated. Samples usually correspond to different phases in the application stream and usually they are no longer than a few million of instructions. In order to archive accurate starting state of each sample, microarchitectural structures are warmed-up for a few million instructions prior to samples instructions. Unfortunately, such a methodology cannot be directly applied for HW/SW co-designed processors.

The warm-up for HW/SW co-designed processors needs to be 3-4 orders of magnitude longer than the warm-up needed for traditional HW-only processor, because the warm-up of software layer needs to be longer than the warm-up of hardware structures. To overcome such a problem, in this thesis we propose a novel warm-up technique specialized for HW/SW co-designed processors. Our solution reduces the simulation time by at least 65X with an average error of just 0.75%. Such a trend is visible for different software and hardware configurations.

The process used to determine simulation samples cannot be applied to HW/SW co-designed processors as well, because due to the software layer, samples show more dissimilarities than in the case of HW-only processors. Therefore we propose a novel algorithm that needs 3X less number of samples to achieve similar error in comparison with state of the art algorithms. Again, such a trend is visible for different software and hardware configurations.

Summarizing, in this thesis we answered the question on how to simulate HW/SW co-designed processors, potential solutions to move towards the less power consuming and the less complex designs. We proved that the software layer cannot be excluded from the simulation and we proposed efficient warm-up and sampling techniques which are tolerant to different software and hardware configurations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the work done in this thesis. It firstly motivates Hardware/Software (HW/SW) co-designed processors and very briefly explains why the simulation process is so important and so difficult for them. After that it lists our main contributions, which provide fast and accurate simulations for HW/SW co-designed processors. The chapter finishes with the discussion of our published work together with the organization of the thesis.

1.1 Motivation

HW/SW co-designed processors are an effective solution for reducing the complexity, reducing the power consumption and improving performance [67]. Unlike traditional *HW-only* processors, they are equipped with a software layer that dynamically analyzes, profiles, translates and optimizes instructions from a guest ISA to an internal microarchitecture with its own customized host ISA. Such a software layer sits below the OS, and it is totally transparent to the entire software stack (Figure 1.1). This layer is often organized as a *staged optimizer*, in which lower optimization stages consist of an interpreter or a fast translator for individual instructions and code regions. As code regions become hotter, they are promoted to higher and more aggressive optimization stages. Optimized code regions are stored in a *code cache* and in the steady state most of the code is fetched and executed from it. While this software layer has received many different names (*e.g.* Code Morphing Software in Transmeta designs [24, 42]), we refer to it as the *Transparent Optimization Layer* or TOL throughout the rest of this thesis. Splitting the processor design into the two components (hardware and software) aims to achieve better performance, lower power consumption, lower design complexity, better design customization, backward/forward ISA compatibility or a

combination of them [42, 64, 25, 24, 44, 26, 38, 55, 56, 72, 60, 71, 51, 46, 47, 61, 13].

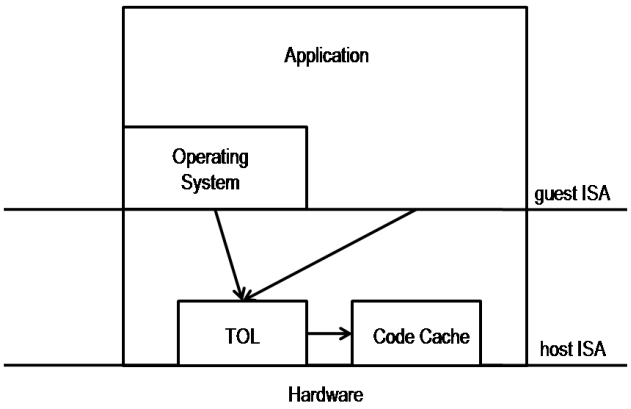

Figure 1.1: HW/SW co-designed processor. Extracted from: J.E.Smith, "Virtual Machines: Versatile Platforms for Systems and Processes", 2005 [67].

Due to the HW/SW collaborative existence in such systems, architects need to decide what part of a particular performance/power/compatibility feature is implemented in hardware and what part is implemented in software. Solutions range from an entire hardware implementation to an entire software implementation, with many of them requiring support from both components. All these design trade-offs require the use of simulators in order to be evaluated, since the direct fabrication of a design is not visible due to cost. Although typical evaluation process requires evaluation of performance and power, in this thesis we are focused only on the performance simulation. The power simulation is the part of the future work and nevertheless opens the same level of challenges.

Simulators and relevant methodologies are established for HW-only processors. The community is supported with many simulators, with a predefined user interface and well defined usage. However, the situation for HW/SW co-designed processors is drastically different. The main difference is that academic open source tools do not exist. This is so because a simulator for HW/SW co-designed processors implies the effort of TOL developing (usually from the scratch since the host ISA is a custom ISA) and this usually takes multiple years.

In order to ease the process of simulator development, some simulation approaches ignore the TOL or assume its behavior to be constant [56, 38]. In these scenarios, the code cache is built off-line and the TOL instruction stream is excluded from the simulation process. In this thesis we verify if this assumption is valid or not. In order to provide that information, we have experiments in which we isolate the execution of TOL instructions. As it is shown in Figure 1.2, TOL behaves very differently for different applications and therefore previous assumptions become not valid. TOL overhead ranges from 0% to 70% of whole

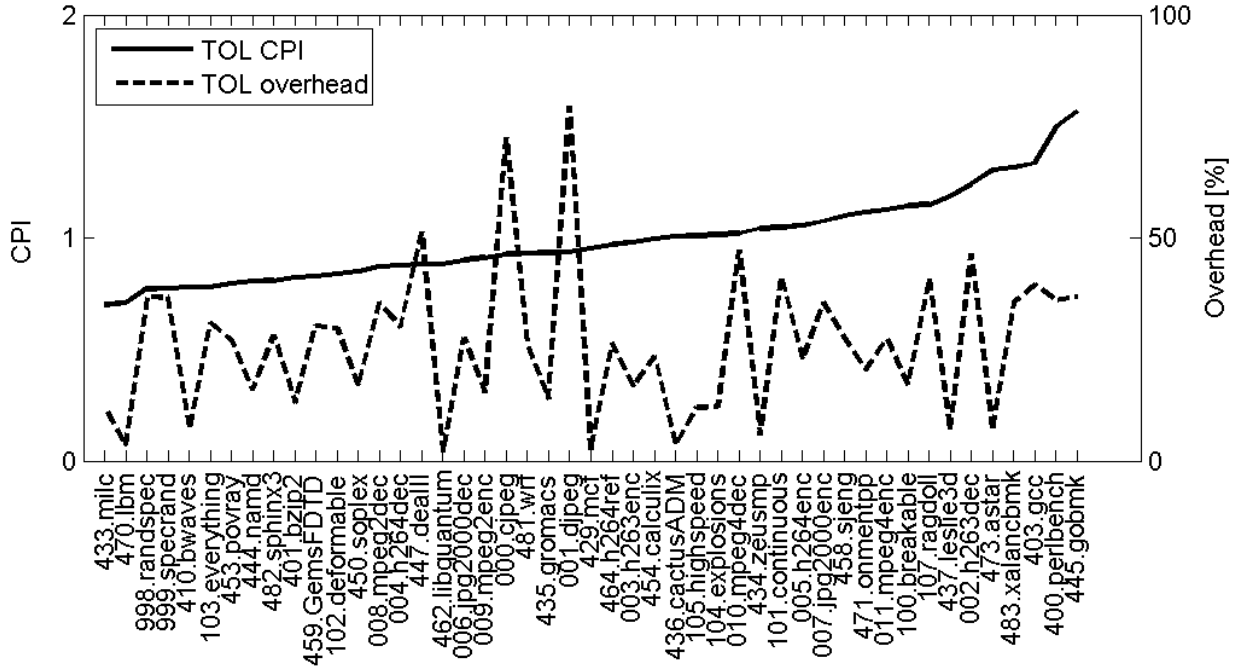

Figure 1.2: TOL CPI and TOL overhead. Applications are sorted from the ones with the lower "with interaction" CPI (left) to the ones with the higher CPI (right).

execution time across different applications. There are few reasons for such behavior. Firstly, different applications imply different behavior of the translator. Secondly, the host microarchitectural behavior of the TOL is not constant and as it is shown it ranges from 0.8 to 1.5 Cycle Per Instruction (CPI) for simple in-order two-way processor. In this thesis (Chapter 4) we explain these reasons more broadly and we study the potential techniques to predict it.

On the other side, typical cycle-accurate simulators are very slow (4-5 orders of the magnitude slower than native execution). This would mean that simulators are not so practical to be used, because the simulation might take years to complete. Therefore computer architects developed techniques that speed up the simulation time without sacrificing the results. However, all of them are used for HW-only architecture.

They are based on the idea where only a part of the application stream is *simulated*, while the rest of the application stream is *skipped*. The simulated part of the application is usually called *sample*. Among many different solutions, samples are usually chosen as dissimilar *phases* and therefore the process of samples' selection is called phase classification. For sampling to be applicable, the architectural state (registers, memory etc.) at the beginning of the sample needs to be maintained. This is done for correctness. On the other side, for accuracy, microarchitectural state (caches, branch predictor etc.) has to be *warmed-up*. This is done usually by executing a few million of instructions prior the sample.

For HW/SW co-designed processors, traditional warm-up and traditional phase classification cannot

be applied *a priori* in order to achieve accurate and fast simulations at the same time. The warm-up process of the HW/SW co-designed processors has to include the warm-up of the TOL as well, which is a way more expansive than the warm-up of the microarchitectural structures, as depicted in Figure 1.3. Y-axis represents the simulation error, while X-axis represents the number of instructions used for warm-up (warm-up length). As it can be observed, the length of the TOL warm-up needs to be at least 3-4 orders of magnitude longer than the length of the microarchitectural warm-up in order to reach similar errors. If small warm-up period is used (1M instructions) the simulation error can be more than 100%. The reason for such behavior is based on the fact that for small warm-up intervals (few million instructions) the code regions in the sample are still in the lowest optimization stage. To overcome that issue, in this thesis we propose a technique which efficiently warms-up TOL structures, and consequently reduces and limits the simulation error (Chapter 5).

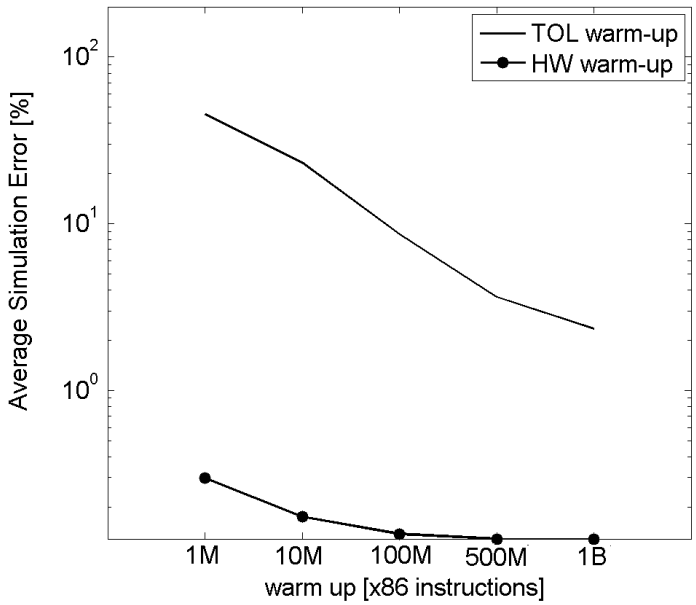

Figure 1.3: Average simulation error vs. warm-up period for HW/SW co-designed processors in the case of: (*i*) TOL warm-up and (*ii*) HW warm-up.

On the other hand, traditional phase classification algorithms are based on the fact that each application's basic block behaves similarly each time executed. In the case of HW/SW co-designed processors, this assumption is not valid due to staged optimization, non-constant TOL behavior etc. Even for HW-only processors there is a variance in the performance of each basic block but this is limited to microarchitectural variances, while for HW/SW co-designed processors this variance is much bigger. Consequently it leads to a 2-3X higher simulation error when applied in the case of HW/SW co-designed processors than in the case of HW-only processors. This is depicted in Figure 1.4 which compares a simulation error of traditional phase classification for HW/SW co-designed processor and HW-only processor with respect to the number

of the selected samples. In order to overcome this issue, in this thesis we propose phase classification which outperforms traditional phase classification in the case of HW/SW co-designed processors (Chapter 6).

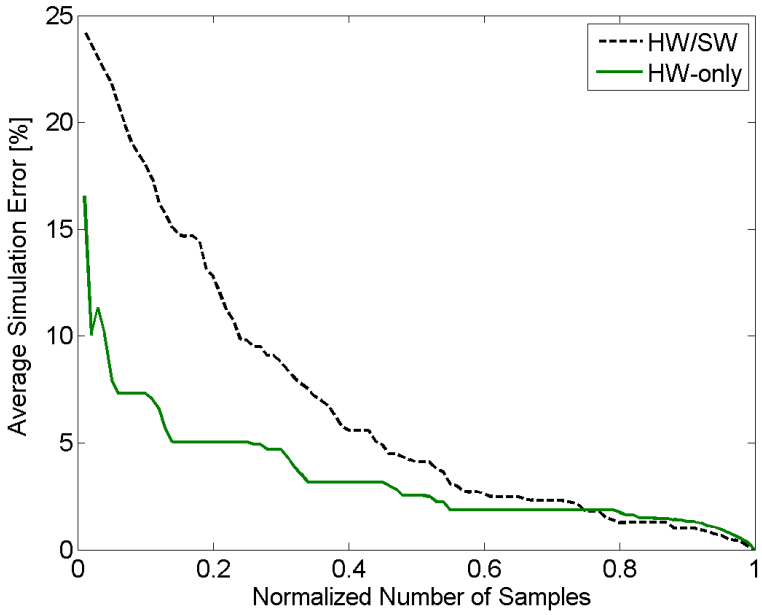

Figure 1.4: Average simulation error vs. number of selected samples for traditional phase classification in two different scenarios: (*i*) HW-only and (*ii*) HW/SW co-designed processor.

1.2 Contributions

The previous section has pointed out the main challenges targeted in this thesis, which are: (*i*) non-constant TOL behavior, (*ii*) inefficient warm-up period and (*iii*) inefficient phase classification algorithm. These facts make inaccurate simulation results. To overcome them, we propose a set of techniques that gives more accurate and faster simulations for HW/SW co-designed processors.

In a nutshell, the contributions of this thesis could be derived into three parts. As the first contribution we study the behavior of Transparent Optimization software Layer (TOL) in order to prove whether TOL simulation is a must or not. The second contribution focuses on the proper warm-up methodology, while the third contribution is related to proposing appropriate phase classification for HW/SW co-designed processors. These contributions are explained in the subsections that follow.

1.2.1 Non-Constant and Non-Predictable TOL Behavior

Many experimental setups for HW/SW co-designed processors are based on the assumption that the TOL overhead is constant or / and ignored and therefore they exclude the TOL from the cycle-accurate

simulations [56, 38]. In this thesis we show that such an assumption could lead to wrong results.

In order to prove that, we firstly study the TOL as an independent application. We show that the behavior of the TOL is not constant and explain that there are three main reasons behind this variance. The first is the microarchitectural interaction between the TOL and the guest application. In order to understand this behavior, we have isolated the behavior of the TOL from the code cache behavior. The results show that the microarchitectural interaction between the code cache and the TOL is not only significant, but also it is not constant (*i.e.* varies for the different guest applications). The second reason is related to the fact that different applications stress differently the functionality components of the TOL, which in this thesis we call *modules*. Some modules, especially the ones that have a significant contribution to the TOL overhead, have been found to vary significantly across the different workloads. The third reason is the fact that after excluding the interaction, the measured CPI for the different modules still shows variability which is different for each module. This variability was found to be more significant for the modules that have lower contribution to the TOL.

In addition to studying the variability of the TOL we also try to predict the behavior based on the *high-level* statistics of the *guest* application. We prove that there exists some correlation only between the high-level guest statistics and the behavior of the modules that are stressed the most. Unfortunately, the modules used for the most aggressive code region optimizations show smaller correlation. For the applications which will stress more these modules, the predictability with high-level statistics is not possible.

1.2.2 Warm-Up for HW/SW Co-Designed Processors

As stated before, the warm-up technique is necessary to maintain accurate simulations. We studied many different warm-up approaches used for HW-only architectures. We show that the warm-up period needed for accurate TOL structures has to be 3-4 orders of magnitude longer than the warm-up of the microarchitectural structures.

To overcome this, we propose a novel warm-up technique used for TOL state warm-up, called Down-scaling Promotion Thresholds, based on adapting the optimization promotion thresholds, which are used as values to promote the optimization stage of a given code region, in order to find the best trade-off between accuracy and simulation time. Such a technique, as opposed to other alternatives, allows evaluation using different TOL and microarchitectural configurations.

We also propose a predictive scheme for the selection of the warm-up setup for a given sample. The solution is based on the high-level statistics and therefore it does not require the cycle-accurate simulation to

collect them. The proposed predictive scheme delivers the warm-up setup very close to the oracle selection.

1.2.3 Phase Classification for HW/SW Co-Designed Processor

We also study the techniques for accurate phase classification. We prove that the traditional phase classification when applied to HW/SW co-designed processors have very low accuracy. This is due to the fact that traditional techniques do not take into account the particularities of HW/SW co-designed processors, such as the handling indirect branches and the staged optimization effect.

Therefore in this thesis we propose a novel off-line phase classification scheme called TOL Description Vector (TDV). TDV is based on the global estimation of TOL particularities and on average gives better accuracy for any number of selected samples than traditional classification. Moreover, TDV has the same average error with 3X less number of samples than traditional classification. Such a trend is visible for different TOL and microarchitectural configurations.

Finally, we show that the fluctuation of different TOL configurations produces higher variability in simulation errors than the fluctuation of different microarchitectural configurations. However, in all studied scenarios TDV outperforms the traditional algorithms.

1.2.4 Side Contributions

In addition to all, we show that the quality of the warm-up cannot be defined solely by the CPI error of the simulation but that the error should be studied as a vector of the errors of many other TOL metrics, such as the number of host instructions and of the dynamic coverage of the different optimization levels. We show that using just the CPI as the error metric is insufficient and can mislead the research in the field of HW/SW co-designed processors.

All simulation techniques are proposed and evaluated for DARCO infrastructure [58], where the guest ISA is x86, while host ISA is PowerPC-like ISA. DARCO stands for Dynamic binary translator from ARCO research group and it is a tool that is developed in collaboration with other colleagues during this thesis. To the best of our knowledge, this is the first academic tool for HW/SW co-designed processors that is published.

1.3 Relationship to Previously Published Work

This thesis extends previously published work. DARCO simulation tool is firstly published in Workshop on Architectural Support for Binary Translators (AMAS-BT) in 2011 [58]. As the continuation of

this work, we also wrote technical report [59] about the workload characterization for HW/SW co-designed processors. It includes basic facts about DARCO and characterizes modern benchmarks suites (SPEC2006, Media Bench, Physics Bench).

Analysis of the TOL behavior is presented in two places. In Workshop on Architectural Support for Binary Translators (AMAS-BT) in 2012 [15], the main facts and initial results are presented. After that, the studies are extended with predictability studies and that work is published in International Conference on Computing Frontiers in 2013 [16].

The proposed solution for the warm-up technique is presented in International Conference on Code Generation and Optimization in 2014 [18]. It mainly includes the facts shown in this thesis.

Finally, the proposed solution for accurate phase classification is published in International Conference on Computing Frontiers in 2014 [17].

1.4 Thesis Organization

This section introduces the organization of the thesis. Chapter 2 describes the necessary background in order to follow the thesis and it is divided into three parts. The chapter firstly presents the concept of HW/SW co-designed processors. After that, it studies existing simulation techniques used for HW-only processors. Finally, the chapter summarizes techniques currently used for HW/SW co-designed processors.

Chapter 3 describes the experimental methodology followed in this PhD thesis. It contains four parts. The first part describes DARCO simulation infrastructure, which is was developed during the period of this thesis. The second part describes the baseline architecture used in this thesis. The third part defines the simulation error and simulation time, while the fourth part introduces the used benchmarks suites used for the evaluation purpose.

The next three chapters focus on the thesis contributions. The first contribution of the thesis, which is the analysis of the TOL behavior, is outlined in Chapter 4, where the TOL is observed as an independent application. The chapter presents the variability results and the main reasons for such a behavior and then it studies the predictability of this behavior. After that, the predictability analysis is presented.

Chapter 5 outlines the second contribution of the thesis, which is the warm-up of the TOL and the microarchitectural structures in the case of HW/SW co-designed processors. The chapter discusses the problem of the warm-up in HW/SW co-designed processors and provides a survey of the existing warm-up techniques. Then it explains and evaluates the proposed scheme.

Chapter 6 outlines the third research contribution of this thesis, which is the off-line phase classifica-

tion in the case of HW/SW co-designs. The chapter firstly describes the problem of the phase classification in HW/SW co-designed processors and studies the existing techniques. Then it presents and evaluates the TDV scheme.

Finally, Chapter 7 summarizes the thesis and discusses the future work.

Chapter 2

Background

This chapter presents the background that is relevant to the thesis. The first section explains more broadly the concept of HW/SW co-designed processors. The chapter continues with the survey of the current simulation techniques used in computer architecture and finishes with the discussion of current trends in the simulation of HW/SW co-designed processors.

2.1 HW/SW Co-Designed Processors

HW/SW co-designed processors utilize a software layer in order to reduce power, reduce area and improve performance. While this software layer has received many different names, we refer to it as the *Transparent Optimization Layer* or TOL, in which dynamic information is used in order to provide Front-Ends of different ISA and the segment-specific optimizations. The Operating System (OS) is not aware of existence of the TOL, as depicted in Figure 2.1-a. It sees only conventional guest ISA as in the case of any off the shelf processor. Figure 2.1-b shows simplified representation of HW/SW co-designed processors that will be used during this thesis.

It is important to notice that HW/SW co-designed processors are in general type of virtual machines, because underlying hardware is hidden from the user. However a general term for such processors is not defined well. In some cases these architectures are called “HW/SW co-designed virtual machines” [67], in some “Binary Translator based Processors” [56]. In this thesis we call them “HW/SW co-designed processors” and it does not apply for ASIP (Application-Specific Instruction-set Processor) or other type of custom processors.

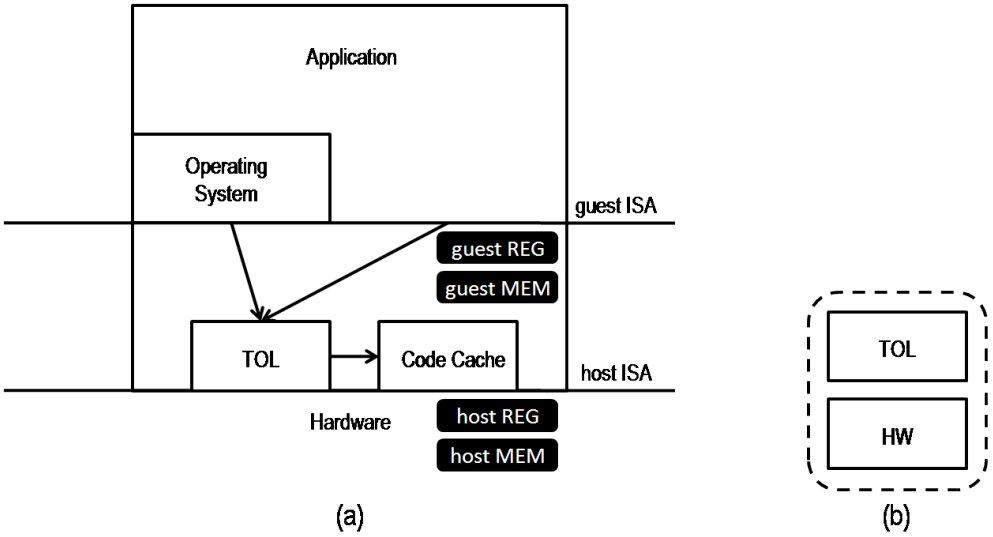

Figure 2.1: HW/SW co-designed processor: (a) more detailed representation with register and memory components and (b) simplified representation. Extracted from: J.E.Smith, "Virtual Machines: Versatile Platforms for Systems and Processes", 2005 [67].

2.1.1 Staged Optimization

TOL is mainly in charge of doing Dynamic Binary Translation (DBT) from the *guest* ISA to the *host* ISA. This is done through several translation/optimization stages, with application profiling support etc. Translated application instructions are stored in the specified part of the memory called *code cache* and the translation is done in several *optimization stages*. A portion of the application being translated is called *code region*.

The main idea of the staged optimizations is the following: less frequently parts of the applications are optimized with lower optimization overhead, while very frequently code regions are optimized very aggressively with the high optimization overhead. Depending on the trade-off analysis, the optimizers can have different number of stages. Code regions can be formatted at the level of instructions, level of basic blocks or some regions that are formed by few basic blocks, such as superblocks, hyperblocks, traces etc. Without loss of generality, in this thesis we model the most common case of three optimization stages: interpretation, basic block optimization and superblock optimizations.

Most commonly, translation firstly starts with instruction by instruction interpretation. When the particular basic block (BB) becomes hot (*i.e.* executed more than a particular threshold value), the entire basic block is translated. If a particular code region is extremely frequently executed, then aggressive optimization techniques and aggressive instruction scheduling are performed in order to produce very efficient translated code. In this case the code regions are not limited to the basic blocks only; they can contain many conse-

quently executed basic blocks and they are typically built in the form of a superblock (SB). Superblocks are discussed in the following subsection. The entire process of the code translation is given in Figure 2.2. As it is depicted, basic blocks' execution counters (EC) are used to support the promotion between the optimization stages and the values that define the promotions are called *threshold values* (*e.g.* TH_{BB} and TH_{SB}).

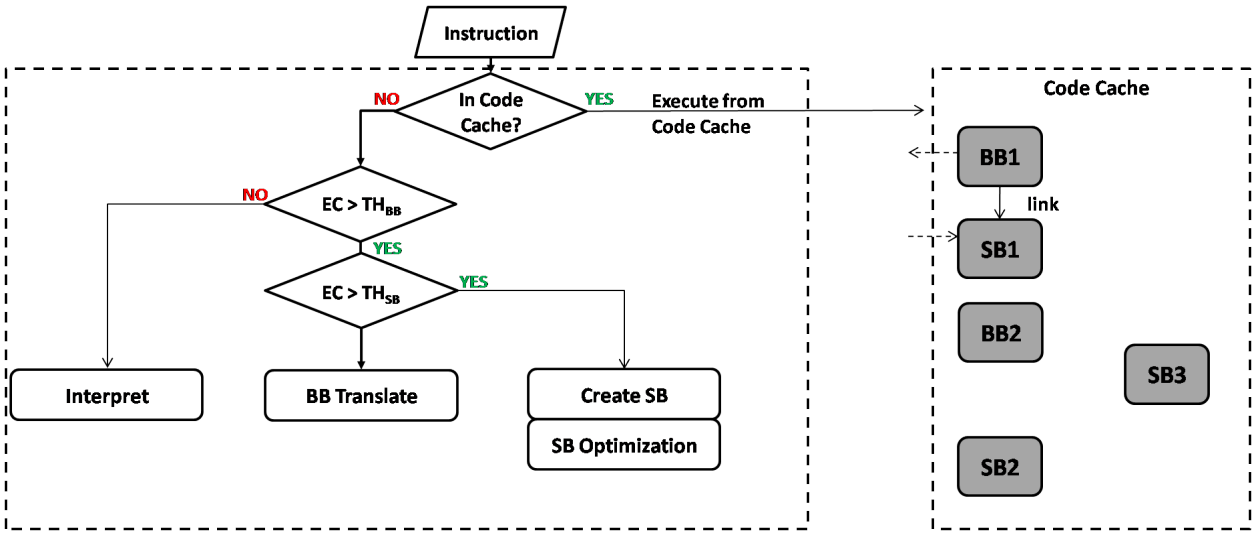

Figure 2.2: Code regions promotions: interpretation, basic block translation (BB) and superblock optimization (SB).

As already said, at the highest optimization stage code regions are usually constructed as superblocks - single-entrance multiple-exit code regions. A superblock follows the most frequently and the most biased execution path. This path is identified with two types of the profiling information: *execution* and *edge* profilers. As discussed, the execution profilers (or execution counters) are used to detect the hot code regions, while the edge profilers are used to detect biased branches. The example of superblocks is given in Figure 2.3-a. Figure illustrates the two most common scenarios for superblock construction. The first superblock includes basic blocks BB1 and BB3, which are selected because basic block BB1 is frequently executed and the branch BB1→BB3 is highly biased. This superblock does not have BB4 since it introduces side entrance. Loop with basic block BB4 creates the second superblock.

In order understand better the concept of the superblocks, optimization stages and code regions, we provide the survey of other code region formation that can be used instead of superblocks , which includes *traces* and *hyperblocks*. For example, in Figure 2.3-a trace contains basic blocks BB5 and BB7. Traces stand for multiple-entry, multiple-exit code regions. Unlike the superblock, they may have a side entrance.

On the other side, hyperblocks have control flow inside itself. One of the examples would be the code

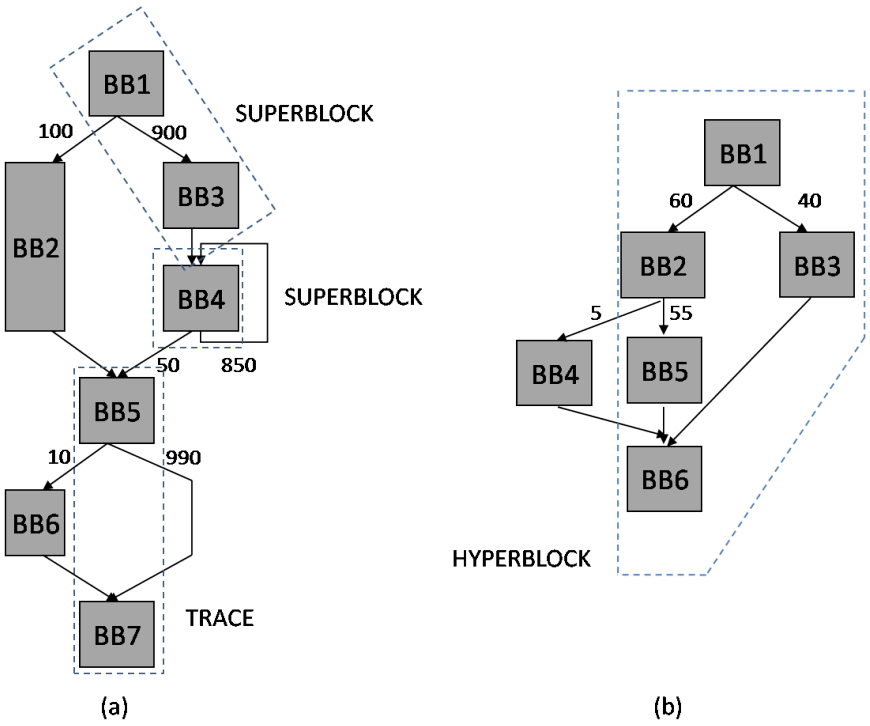

Figure 2.3: Examples for superblocks, hyperblocks and traces.

region formed in Figure 2.3-b. Unlike the superblocks, hyperblocks are used when the branches are not biased.

Profiling information used for code region formation can be collected in two manners: execution counters based and time based. Execution counters based profiling profiles the frequency of desirable events. On the other side, in time based profiling (JVM, CMS) the notion of time (*e.g.* CPU cycles) is used to trigger the sampling event during which the profiling information is collected. In this thesis we focus only on execution counters based profiling, but nevertheless our proposed methodology works for time based profiling as well.

2.1.2 Architectural State

Besides code regions translation and optimization, TOL has to ensure precise guest execution as well, which means precise exceptions handling, precise execution of self-modifying code and memory management [67]. These requirements are caused by the fact that guest registers and memory and host registers and memory are physically decoupled between each other, which is illustrated in Figure 2.1. Many different approaches exist in order to solve these issues, but their survey is not presented in this thesis, due to the fact that they are not the main reason for the effects shown in this thesis and therefore they are not simulated (Section 3). More details can be found in [67].

2.1.3 Switching between the TOL and the Application

TOL can be also seen as an application and the host processor perceives instruction execution as switching phases of TOL and code cache execution. The TOL execution can be considered as *overhead* whereas the code cache execution is the code that provides application forward-progress. Figure 2.4 depicts the switching between the TOL and the code cache. This switching happens continually during the execution and is shown at the right part of the figure, which presents the instruction stream executed on the host machine. Beside that natural classification, in this thesis we further divide TOL execution into more fine grained *modules* each of which is responsible for a specific TOL functionality. In Figure 2.4 these parts are marked as module A, module B and module C. The examples of the modules could be specific module for basic block translation, specific module for superblock formation etc.

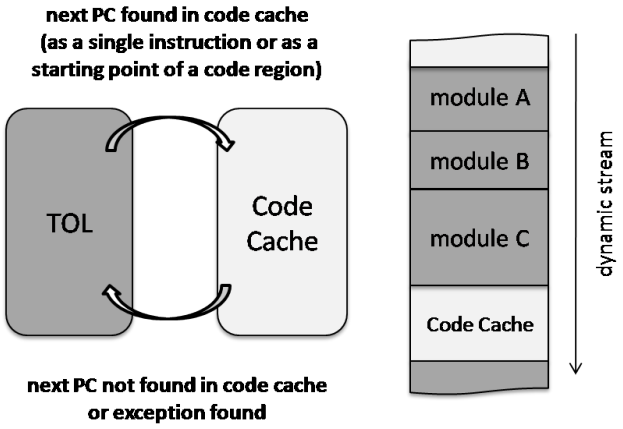

Figure 2.4: Illustration of switching between the TOL and the translated application stored into the code cache.

2.1.4 Linking

TOL intervention is not needed every time the execution of code regions is terminated. At the beginning, after the execution of the particular code region the control flow goes back to TOL. However, this is not necessary for the cases when code region finishes with a direct branch and the next code region is already translated. In these cases the execution can be continued directly from the code cache by storing a direct branch (or "*link*") between two code regions in the code cache. As an example in Figure 2.2 basic block BB1 and superblock SB1 are linked and in that case TOL execution, illustrated with dashed pointing line, is bypassed.

Linking is used as a performance feature. This would mean that the approach that has linking will be

faster than approach without linking.

2.1.5 Projects and Industrial Examples

The concept of HW/SW co-designed processors can be found in a few industrial products released in the past: DAISY [26], BOA [7], Crusoe and Efficeon [24, 42] (both known as Transmeta's processors). In all these products the host processor is VLIW processor, while guest ISA is different among them - Crusoe and Efficeon has x86 guest ISA, whereas BOA and DAISY have Power PC guest ISA. Since HW/SW co-designed processors suffer the most from the translation overhead, some solutions like PARROT [62] and Reply [57] reduce the TOL overhead by performing profiling and run time optimizations in hardware. Obviously, they are faster, but they are less flexible and may result in a power consumption increase and additional validation cost. Recently NVIDIA's SoCs (System on a Chips), such as Tegra K1, are based on Denver processors [13, 61]. Denver architecture implements the ARMv8-A 64/32-bit instruction sets using a combination of simple hardware decoder and software-based dynamic binary translation, so it can be considered as a HW/SW co-designed processor.

2.1.6 State of the Art

The research in this area goes in many directions. Due to this HW/SW collaborative combination in such systems, designers need to decide what part of a particular performance/power/compatibility feature is implemented in hardware and what part is implemented in software. Solutions range from an entire hardware implementation to an entire software implementation, with many of them requiring support from both components. Examples of techniques that require support from hardware and software include, but are not limited to, the dual-address return address stack [41] or flook stack [45] to handle subroutine calls and returns, memory disambiguation support for data speculation optimizations [24], the link pipe to handle indirect branches [45] and hardware support to accelerate the execution of particular TOL components as the instruction decoder [38] or the optimizer [60]. These solutions may also require the addition of new host instructions in order to define the interface between the host hardware and the Transparent Optimization Layer (TOL). In addition, the decision on which part is implemented in hardware and which in software may also depend on the details of the microarchitectural implementation, such as the configuration of the memory hierarchy or the branch predictor design.

2.2 Computer Architecture Simulators

Simulators are mainstay of the research in computer architecture. It is impossible to build several versions of the hardware to try alternatives and on the other hand systems are getting too complex to be analytically modeled. This section presents the terms and techniques used for a simulation in computer architecture. We firstly explain different simulation levels and after that we explain simulation techniques which are used in order to speed up simulations for HW-only processors, bust still having accurate results.

2.2.1 Simulation Levels

Simulators are very expensive in terms of development, validation, testing and simulation time. Therefore they are built through different simulation levels. Simulation time of each level depends on the level of details of an architecture being simulated. Among many different classifications, one classification is predominant in the literature and it divides simulators into two main categories: functional and cycle-accurate. In a nutshell *functional* simulators emphasize achieving the same function as the modeled components (what is done), while *cycle-accurate* simulators accurately reproduce the performance/timing features (when is it done) of the targets in addition to their functionalities.

Figure 2.5 presents the functional and cycle-accurate simulation levels. The functional simulation corresponds to the simulation of the architectural behavior, such as registers, memory etc.; while cycle-accurate simulation corresponds to the simulation of the microarchitectural behavior such as caches, branch predictor, scoreboard, etc. According to the literature [9, 75], functional simulators are usually 1.5-100X slower than native execution, while cycle-accurate simulators are 4-5 orders of magnitude slower than native execution.

Notice that at each of these levels some components can be excluded from the simulation which makes the simulation time shorter, but provides lower accuracy. For instance during the functional simulation memory addresses can be tracked or not, during cycle-accurate simulation wrong paths can be simulated or not etc. In Figure 2.5 these components are represented by dashed square boxes. For clarity reasons and without loss of generality, only subset of all components is depicted.

Splitting the design of a simulator between functional and cycle-accurate helps the process of simulation development. Typically engineers firstly develop the functional and then on the cycle-accurate simulator. However, the splitting does not help only the simulation development, it also provides an opportunity to speed up the simulation while keeping results accurate, as it is going to be explained in Section 2.2.2.

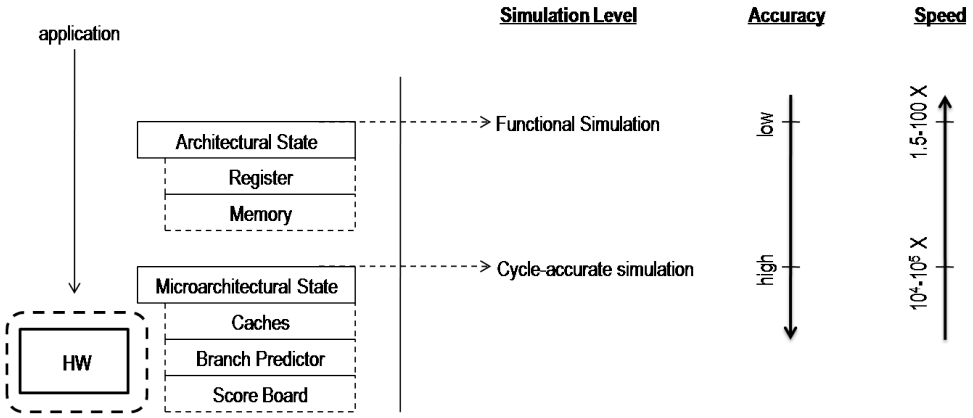

Figure 2.5: Different levels of the simulation accuracy for HW-only processors.

Simulators for HW/SW co-designed processors also require different simulation levels. However, the situation in this case is a bit different since a TOL simulation level is needed as well, as depicted in Figure 2.6. From simulation perspective this means that there is one more level of the simulation compared to the HW-only architectures (Figure 2.5). For HW/SW co-designed processors there are two types of functional simulations: (*i*) guest functional simulation and (*ii*) TOL functional simulation. TOL functional simulation stands for the TOL execution on the host hardware and it maintains the host architectural state together with TOL profilers and code cache, as well as anything else the TOL wants to keep.

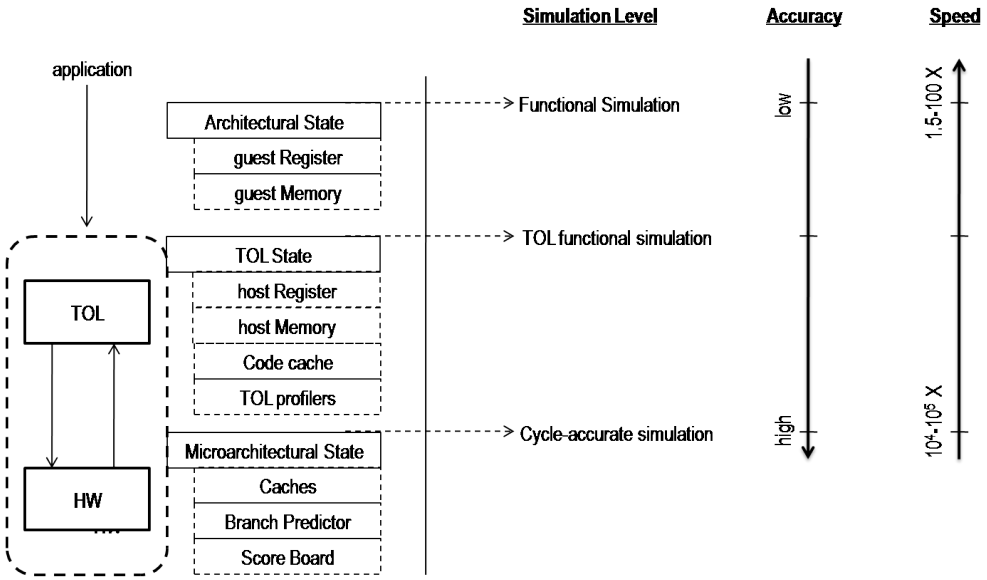

Figure 2.6: Different levels of the simulation accuracy for HW/SW co-designed processors.

Similar to the case of HW-only architectures some components can be excluded from the simulation which makes the simulation time shorter, but provides lower accuracy. In Figure 2.6 these components are

represented by dashed square boxes. For clarity reasons and without loss of generality, only subset of all components is depicted.

It is important to notice that strictly speaking the microarchitectural state in the case of HW/SW co-designed processors includes the state of the TOL structures and the state of the host hardware structures. However, in this thesis we refer to the microarchitectural state as the state of the host hardware structure only, whereas the state of TOL structures is named as the TOL state. This is done for the following reason. Hardware components are common for both types of processors (HW-only and HW/SW co-designed). In order to highlight that, we keep calling hardware state equally in both cases - microarchitectural state.

2.2.2 Simulation Techniques

The problem of having fast and accurate simulations dated up on long time, since computer architecture simulators are 4-5 orders slower than native execution. To address this problem, computer architects developed plenty alternative simulation techniques, where the final goal is to have accurate simulation results in a reasonable amount of time. The basic idea of such techniques is inspired by the fact that total simulation time is represented as:

$$s = f \cdot Instr_f + c \cdot Instr_c \tag{2.1}$$

where s stands for total simulation time, f for speed of the functional simulator, c for speed of the cycle-accurate simulator and $Instr_f$ and $Instr_c$ for number of the simulated instructions that are simulated using functional and cycle-accurate simulators respectively. Since f and c are constant for a given simulator, the only solution to reduce simulation time is to reduce $Instr_f$ and $Instr_c$. For authoritative simulations $Instr_f$ and $Instr_c$ are equal to the number of simulated instructions. Simulation techniques that reduce these numbers are divided into three categories [75]: (*i*) reduced input set, (*ii*) truncated execution and (*iii*) sampling, which are explained below.

Reduced Input Set

The basic idea behind the reduced input set is to modify the reference input set in a way to reduce the simulation time [75]. In other words we decrease the both $Instr_f$ and $Instr_c$. The primary advantage of using reduced input sets is that the entire behavior of the program is simulated in detail, including: initialization, the main body of the computation, and cleanup. The main disadvantage is that their results may be very dissimilar compared to those produced by the reference inputs. In addition, developing reduced input sets can be a very tedious and time-consuming process. The examples of reduced input set can be found in SPEC

2000 and SPEC 2006 where benchmark suites include the test and train input sets. However simulation errors in this case are unacceptably high and therefore this approach is not studied in this thesis.

Skip / Warm-Up simulation

In this scenario only a part of the application stream is simulated with cycle-accurate simulator ($Instr_c$). The idea is represented in Figure 2.7. Instead of the simulating entire application (Figure 2.7-a), the performance (Cycles Per Instruction - CPI) is estimated by simulating only part of the application which in this thesis we call *collect* period (Figure 2.7-b). By simulating only a part of the application, the simulation time can be significantly reduced ($\approx$ 1000X). However, for this technique to be applicable, the microarchitectural state (caches, branch predictor etc.) has to be authoritative at the beginning of the collect period. This would again require the cycle-accurate simulations from the beginning of the application, which is still very expensive for the collect periods that are far away from the beginning of the application (Figure 2.7-c). Therefore the microarchitectural state is warmed-up with a few million instructions prior to the collect period (Figure 2.7-d). If the warm-up period is not selected properly, the fidelity of the statistics might be affected. As the warm-up length is longer, the results are more accurate. Warm-up period can be reached in two different manners: (*i*) by functional simulation or (*ii*) by skipping instructions and restoring the architectural state form the checkpoint made from the previous run. More about warm-up and its applicability in the case of HW/SW co-designed processors can be found in Chapter 5.

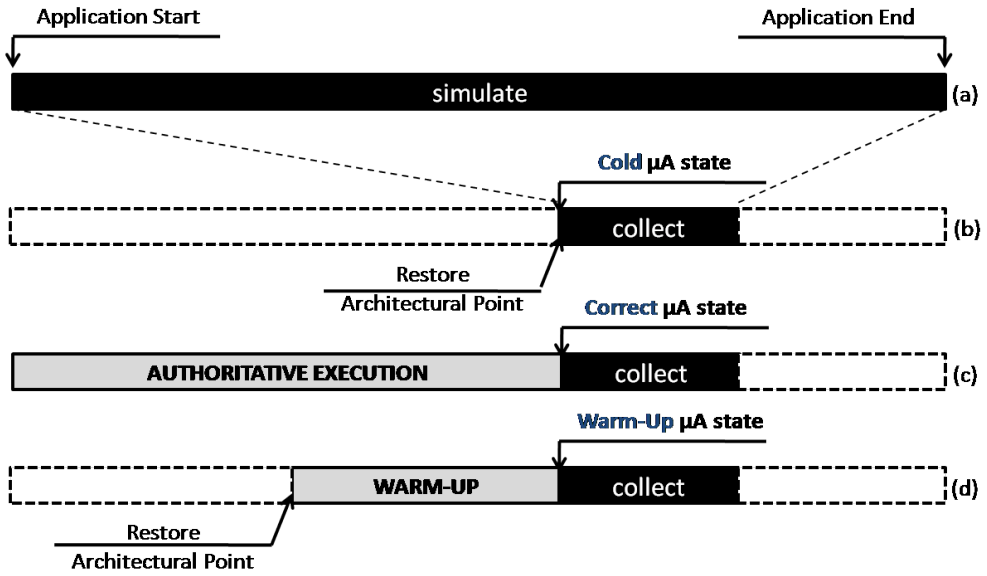

Figure 2.7: Different simulation methodologies: (a) authoritative simulation, (b) skipped simulation without authoritative microarchitectural state, (c) skipped simulation with authoritative microarchitectural state and (d) skipped simulation with warmed-up microarchitectural state.

Sampling

One step further in the simulation is sampling simulation. Instead of simulating only one collect period, as it was the case in skip / warm-up technique, this technique uses many collect periods among the application. Each of these collect periods is called *sample*. By having many samples, the behavior of the application can be better represented, since they are spread among entire application. The process of sampling is illustrated in Figure 2.8, where n samples are selected for the simulation.

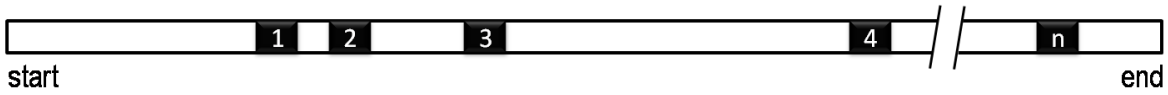

Figure 2.8: Sampling simulation methodology.

In that case, the Cycles Per Instruction (CPI) in application execution is calculated as:

$$CPI_{est} = \Sigma_{i=1}^{n} \alpha_i \cdot CPI_i^{sample}$$

$$\Sigma_{i=1}^{n} \alpha_i = 1, \tag{2.2}$$

where CPI_i^{sample} is the CPI of sample and α_i is the weighted coefficient of that sample. Weighted coefficient (α_i) refers to the percentage of the non selected samples which are similar to the i-th selected sample. The selected samples need to be warmed-up for the same reasons explained in skip / warm-up approach.

In the literature many different sampling techniques exist, out of which SimPoint, COTSon and SMARTS are the most famous ones [75].

In the case of **SimPoint** [66] the samples are usually chosen as the ones which show the most phases' dissimilarity between each other and they are weighted with different coefficients (α_i). On the other hand, the samples that are not chosen show similarity with any of the selected ones. The similarity / dissimilarity between the samples is estimated by employing off-line phase classification, which is based on profiling of the basic block execution frequency. SimPoint is the most used sampling technique in computer architecture.

COTSon [28] also simulates only phases that show dissimilarity between each other, but it predicts them based on the on-line classification. The on-line classification is based on observing the picks in some statistics that comes from internal simulation events, such as number of page walks etc. The issue that comes with this approach is that different simulators can have different internal events, so they can sometimes be irrelevant for the phase changing.

On the other hand, **SMARTS** [73] is performing periodic sampling, based on statistical sampling

theory. Compared to the SimPoint, SMARTS has shorter sampling intervals, and it is able to provide better accuracy. Moreover, the number of simulated instructions depends on the desired accuracy. On average, SMARTS is giving more accurate results, but at same time needs more intervals than SimPoint. Despite the fact that this is a promising solution, the community does not use this methodology a lot, mainly because it is a black box approach.

2.3 State of the Art Simulations for HW/SW Co-Designed Processors

As stated before, the community is facing the lack of tools and simulation techniques for HW/SW co-designed processors. The process of the simulation is no well defined, there are not open source tools, etc. Therefore, different research studies for HW/SW co-designed processors use different simulation approaches and this section describes them. To the best of our knowledge, there are two approaches that solve the issue of the lack of simulation tools and one approach that solves the issue of the lack of simulation techniques. The lack of the tools is compensated in two manners: (*i*) by simulating Dynamic Binary Translators (DBT) with same ISAs for guest and host or (*ii*) by ignoring or assuming constant TOL behavior. Regarding the simulation techniques, the only methodology that proposes simulation techniques for HW/SW co-designed processors is AstroLIT, which is mainly developed to compare Transmeta's processors with traditional x86 processors.

2.3.1 Same ISAs DBT

Since HW/SW co-designed processors are heavily based on DBT, the infrastructure for DBTs could be used for the purposes of the simulation of HW/SW co-designed processors [60]. This would mean simulating any DBT system (DynamoRio [1], Strata [65] etc.) on the top of any existing host cycle-accurate simulator. The input of DBT is the guest application, while the ISA of underlying simulator is the host ISA. However such a simulation scenario does not accurately model HW/SW co-designed processors. As illustrated in Figure 2.9 most of the available DBT infrastructures have same host and guest ISAs, usually CISC ISA (Figure 2.9-a), whereas in reality for HW/SW co-designed processors the host ISA is usually custom RISC or VLIW ISA (Figure 2.9-b) while the guest ISA is usually CISC ISA. This would mean that the synergy between the HW and the TOL, which is reflected in additional instructions specialized for TOL only, cannot be simulated by DBT infrastructures, because specialized instructions have to be supported by host cycle-accurate simulator and existing CISC cycle-accurate simulators do not support them. Moreover, the optimization techniques discussed in Section 2.1 can have drastically different behavior for different

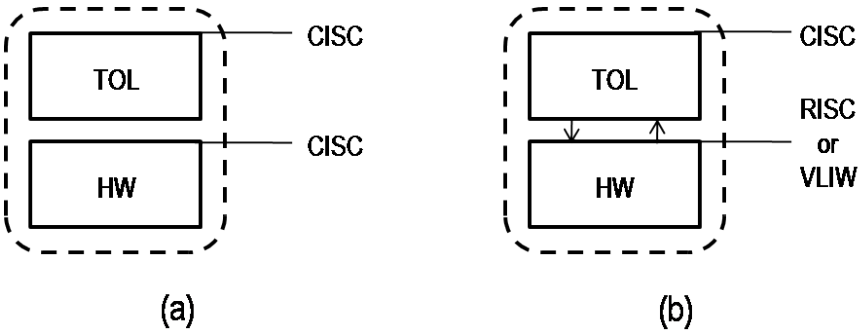

Figure 2.9: Comparison between the simulation methodology (a) using existing DBTs and (b) authoritative simulation

host ISAs (*e.g.* RISC vs. CICS). As discussed before, the typical translation and optimization scenario for HW/SW co-designed processors is CISC to RISC or CISC to VLIW, since they offer more optimization opportunities in comparison with CISC to CISC optimization. Therefore, due to the aforementioned reasons, executing DBT systems on the top of the cycle-accurate simulator does not accurately model HW/SW co-designed processors and TOL and host cycle-accurate simulator should have support for a custom host ISA.

2.3.2 Constant TOL Behavior

The second approach used to evaluate HW/SW co-designed processors supports different host and guest ISA, but unfortunately assumes TOL overhead to be constant [23, 38] or ignored (AstroLIT [56]). These two simulation approaches are depicted in Figure 2.10. These approaches use internal ISAs that are supported by existing academic simulators.

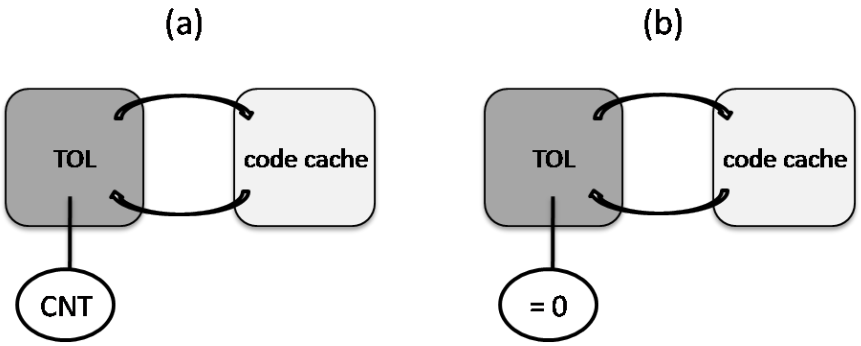

Figure 2.10: Current Simulation Techniques for HW/SW co-designed processors: (a) constant overhead and (b) ignored overhead.

The best example is the usage of PTLsim simulator. PTLsim is a system level, x86 simulator where the x86 instructions are simulated using internal RISC like operations called uOps. This models a pure CISC to RISC translation and it allows support for optimizations. However, the execution of the code that

should correspond to TOL execution cannot be simulated. This is so because the part that performs TOL functionality should be complied for uOps ISA and this is not possible since uOps in not generic ISA and corresponding compiler does not exist. Therefore, the researchers assume the constant TOL behavior to ease the simulation process. As it is going to be shown later in Chapter 4, such an assumption can lead to significant simulation errors and many misunderstandings. More about that can be seen in Section 4.

2.3.3 AstroLIT

The AstroLIT methodology [56] proposes a solution for sampling-based simulation of HW/SW co-designed processors. It is the only work that points to main challenges of the simulation of HW/SW co-designed processors and its differences in comparison with traditional HW-only processors. The work is initially designed in order to compare Transmeta x86 architecture with traditional Intel x86 architecture and therefore it has limitations.

The AstroLIT lists and solves three issues that are visible in the case the sampling simulation for HW/SW co-designed processors: (*i*) empty host architectural state at the beginning of the sample, (*ii*) empty code cache at the beginning of the sample and (*iii*) inaccurate profiling information affected by sampling simulation. These issues are mainly caused by inefficient warm-up for HW/SW co-designed processors. The first issue is clearly reflected by the fact that the guest architectural state (guest registers, guest memory etc.) is not enough to restore the whole architectural state of HW/SW co-designed processor, since the host architectural state is needed as well (host registers, host memory etc.). Similar applies for the code cache. Unless we have efficient warm-up, the code cache is empty at the begging of the sample. The third issue reflects the fact that the profiling information are also wrong (equal to zero) at the beginning of the sample and therefore it has to be warm-up as well, otherwise the code regions will be differently constructed. These issues we also target in the Chapter 5.

Figure 2.11 illustrates how these issues are solved by the AstroLIT methodology. Firstly, in addition to the guest architectural state, AstroLIT creates a checkpoint of the host architecture as well. Then the code cache is warmed-up by iterating over the same collect period, until the state of the code cache becomes stable. At the end, the TOL overhead is ignored.

As stated before, this solution has many limitations, mainly because AstroLIT has been initially designed to compare Transmeta x86 architecture with traditional Intel x86 architecture. Checkpointing of the host architectural state does not allow changes in TOL in this case. This is explained better in Chapter 5. Ignoring TOL overhead is acceptable for the traces that have highly repeatable code properties, which is

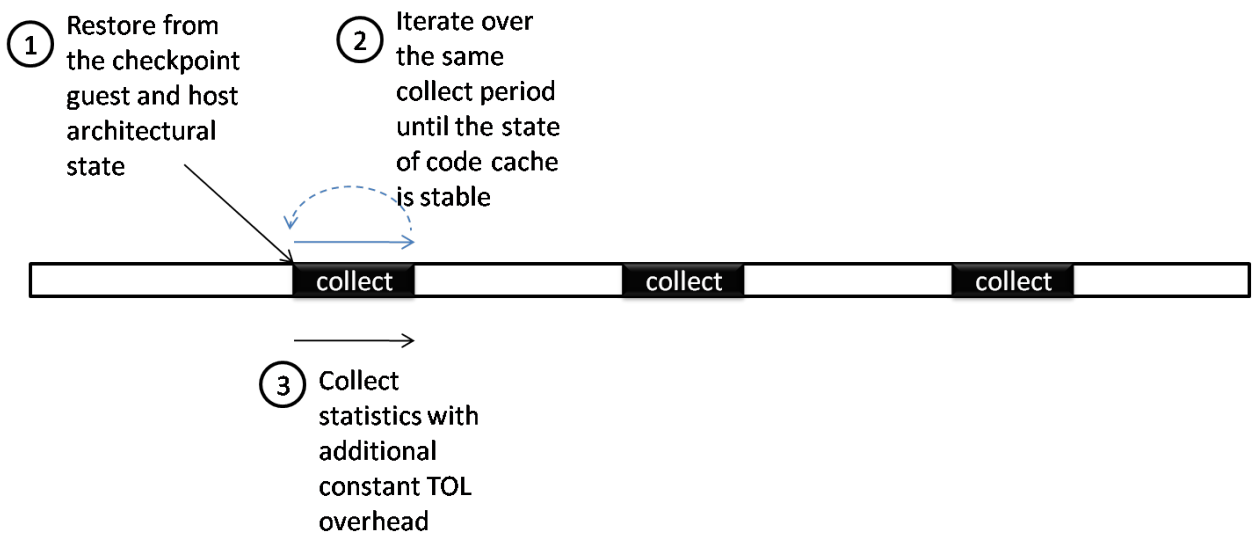

Figure 2.11: AstroLIT simulation methodology.

the case in the scenario of AstroLit. For the traditional benchmark suites used in computer architecture this is not the case. Moreover, iterating over the same collect period in order to warm-up the code cache is acceptable for small traces, but for application with big footprint it is prohibitively expensive.

Chapter 3

Experimental Infrastructure

This chapter presents the experimental infrastructure used in this thesis. That infrastructure was developed during the thesis in collaboration with other members of ARCO research group and it is named DARCO (Dynamic binary translator from ARCO) [58]. The chapter also includes information about used benchmark suites and definitions of the used metrics.

3.1 DARCO Infrastructure

DARCO was developed as an infrastructure specialized for research on HW/SW co-designed processors. It is an academic tool, with reasonable assumptions, which simulates the user level only. It avoids corner cases, such as system calls, exception and interrupts. It has a *modular* approach, supports *extending the ISA* and has solid *debugging interface*, which are the most important requirements for a simulator for HW/SW co-designed processors.

DARCO models the execution of x86 guest ISA (CISC) on PowerPC host machine (RISC). The reasoning behind selecting x86 as the guest ISA is its wide usage. On the other hand, PowerPC is a live RISC ISA, supporting vector instructions which are recently very intensively used. Also PowerPC has compiler's tools support.

To ease the process of the development, to provide a good debugging interface and to support "skip / warm-up / collect" simulation, DARCO is built in a manner that it contains of four main components which interact as depicted in Figure 3.1. Its main components are: the x86 component, the PowerPC component, the PowerPC cycle-accurate simulator and the Controller. Other components depicted in Figure 3.1, such as Process Tracker, State Checker, Debugger etc., are discussed later in this section.

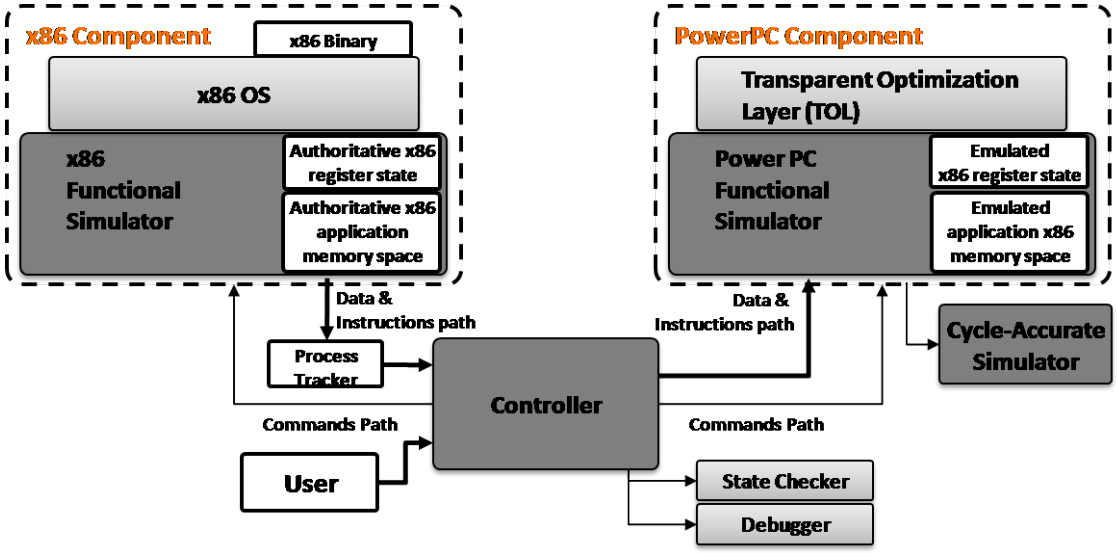

Figure 3.1: DARCO block scheme.

The x86 component provides an x86 full-system functional emulator on top of which an unmodified operating system is executing. The x86 component keeps the correct guest architectural state (authoritative state) and is the only component that interacts with the Operating System. It is responsible for guest *functional simulation* (discussed in Section 2.2.1) and, as it is shown later, it provides the support to avoid the simulation of some corner cases and to have an efficient debug mechanism.

The PowerPC component provides the host processor functional model for DARCO and it executes the Transparent Optimization Layer (TOL) on the top. The PowerPC component has been updated to support host ISA extensions. It is responsible for *TOL functional simulation* (discussed in Section 2.2.1) and execution of the translated code. This component contains the emulated state, whereas as discussed earlier, x86 component contains authoritative state. State checker, which is called by Debugger, compares these two states.

The PowerPC cycle-accurate simulator component is responsible for *cycle-accurate simulation* (discussed in Section 2.2.1).

The Controller is the main interface of DARCO with the user. It provides full control over the execution of the application as well as debugging utilities.

The x86 and PowerPC functional emulators are modified versions of QEMU (Quick EMUlation tool) [4]. The rationale behind selecting QEMU as functional emulator is that it is live and constantly updated tool. Moreover, the high execution speed of QEMU is a significant factor, since it helps the infrastructure to be more efficient. As for the rest of the components, *i.e.* the TOL, the Controller and the PowerPC cycle-accurate simulator are in-house developments.

The execution flow of the simulator is led by the PowerPC component and it passes through three distinct phases: *initialization*, *execution* and *synchronization* (Figure 3.2).

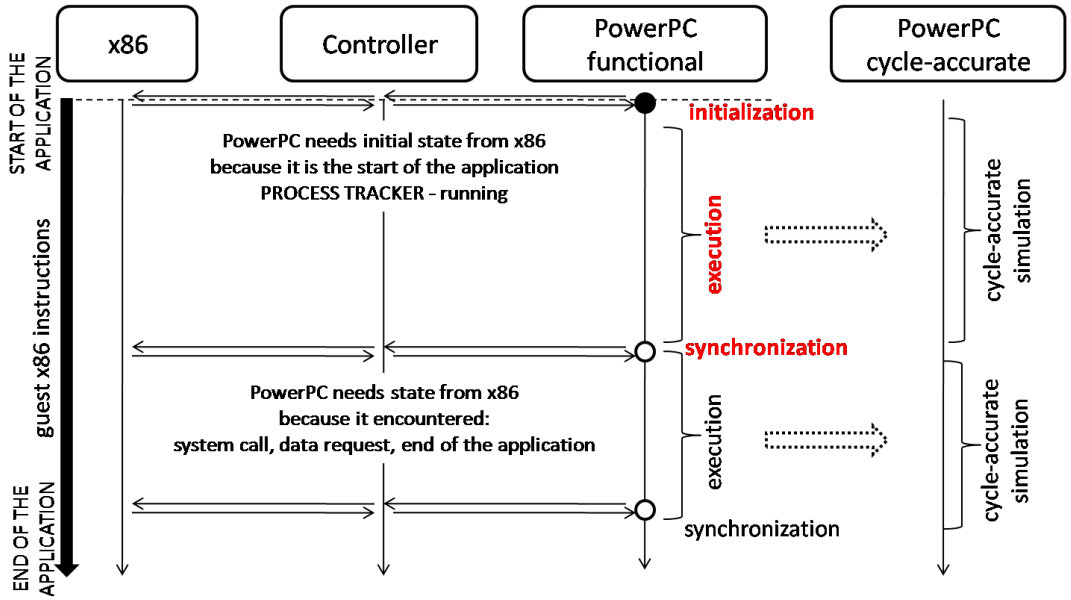

Figure 3.2: Three phases of DARCO: initialization, execution and synchronization.

During the *initialization phase*, the Controller first starts the PowerPC component which in turn, initiates the execution of the TOL. The PowerPC component then remains idle until the Controller sends the x86 register state of the application to be executed to it.

As for the x86 component, when launched, it initiates the execution of the application defined by the user. Notice that to date the current version of DARCO supports only user-level guest instructions. Therefore, when the x86 component reaches the system call EXECVE (which always takes the place at the beginning of an application) the execution pauses and the process tracker is employed. The process tracker is used through the execution of the application in the x86 component in order to ease synchronization and tracking of the changes made to the x86 state, register and memory of the application.

A process tracker is initialized with the application's CR3 value, which can be used to distinguish the specific process from the rest of the dynamic instructions. CR3 is one of the control registers (CR) in x86 architecture. The whole process is illustrated in Figure 3.3 where the left part of the figure depicts dynamic instructions from different applications being executed, while the right part of the figure depicts the values of CR3 register. Whenever a particular application is executed, the value of the CR3 register has a unique value. Therefore, whenever the value of the CR3 register is equal to the value that is set at the initialization stage (*e.g.* value X), the simulator knows that currently executed instructions belong to

the tracked application. For instance, for the example depicted in Figure 3.3 instructions that belong to the tracked application are numerated with the following numbers: 4, 5, 6, 10, 11, 12, 13, 14 and 18.

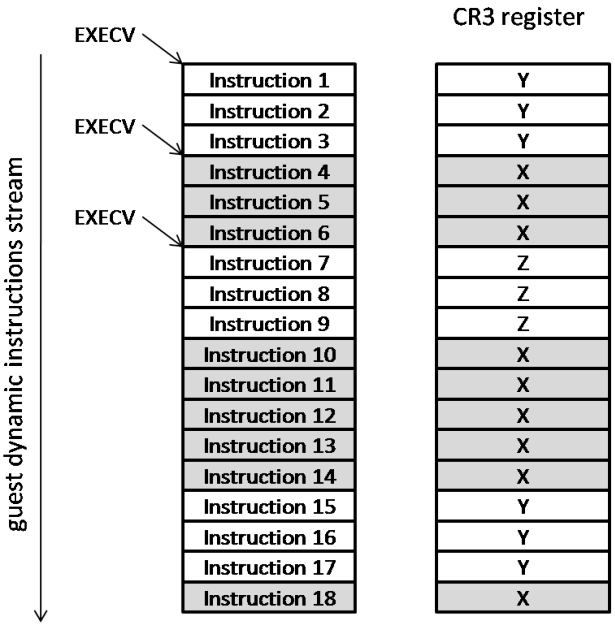

Figure 3.3: Illustration of the process tracker in DARCO.

After the process tracker is initialized, the x86 component send the initial x86 state of the application to the Controller. The initialization phase is completed when the Controller sends this state to the PowerPC component. At this point in time, the x86 register state is the same in both components.

During the *execution phase*, the TOL begins by executing code from the initial %eip it received during the initialization phase. All changes made to the x86 register state from the emulation of the x86 instructions are stored in the "emulated x86 register state" which resides in the memory space of the TOL. Changes made to the memory space of the x86 application are stored in the "emulated application x86" memory space which also resides in the memory space of the TOL. While the x86 application is making forward progress under the TOL, the x86 component remains idle and its memory state untouched.

The *synchronization phase* is initiated by the PowerPC component when any of following three events occurs during the execution phase; (*i*) data request, (*ii*) system call or (*iii*) end of application.

The data request event is raised when the PowerPC component encounters a load or store instruction that accesses an x86 memory location for the first time. The subsequent actions from the other different components are depicted in Figure 3.4. The PowerPC component sends a request to the Controller for the particular data page along with the total number of dynamic x86 basic blocks that were executed until this point. Then, it remains idle until the request is satisfied. The Controller forwards the request to the x86

component, which in turn continues the execution of the application until it reaches the same execution point as the PowerPC component (remember that the x86 component remained idle after the initial launch of the application). When the correct execution point is reached, the data page is sent to the Controller and forwarded to the PowerPC component. This process guarantees that after every synchronization phase, the x86 application state, register and memory, is identical between the x86 component and the TOL. Otherwise the system complains and execution is aborted. This is also a useful technique to debug DARCO. The exact same process is followed for the other two events, system calls and end of application.

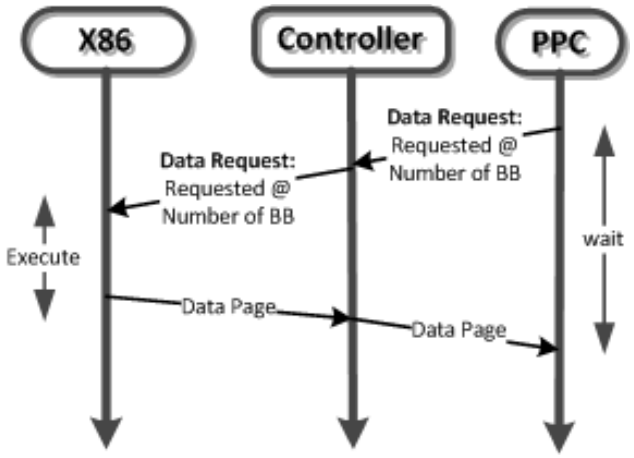

Figure 3.4: Communication between the components in the case of the synchronization due to data request.

System calls raise the synchronization event because the TOL only models user-level code. The synchronization phase will fetch the modifications done by a system call from the x86 component. The communication process is similar to the one depicted in Figure 3.4.

As for the end of application, the synchronization phase is necessary in order to verify that the execution of the application on the PowerPC component was correct.

3.2 Modeled HW/SW Co-Designed Processor

The modeled architecture of a HW/SW co-designed processor is presented in this section. The modeled architecture closely models the Transmeta Crusoe processor [24], with a difference that Crusoe relies on a host VLIW processor, whereas we model a RISC processor. The main parts of the architecture, which are a Transparent optimization software layer and a host hardware, as explained in the rest of the section.

3.2.1 Transparent Optimization Software Layer (TOL)

The Transparent Optimization software Layer (TOL) of DARCO supports the execution of the guest x86 ISA on the host PowerPC hardware. It generates and optimizes an equivalent PowerPC instruction stream for guest x86 stream. Figure 3.5 illustrates this process, showing the translation of a simple x86 move instruction. Like all conventional compilers, TOL has an intermediate representation (in this case called qOps) in order to efficiently apply standard optimization techniques. In addition to the PowerPC code, TOL produces "exit-stubs" (Not Taken and Taken) in order to jump back to the TOL run-time when the execution of translated code region is finished. Instructions that correspond to linking are placed in these "exit-stubs". Besides aforementioned instructions, "exit-stubs" contain "fake instructions" as well (in this case eob_x86). In particular this instruction is used by PowerPC functional simulator in order to ease the process of the simulation, counting the number of dynamically executed basic blocks to help synchronization between x86 and PowerPC functional simulators. During the thesis we will explain other used fake instructions. Notice that fake instructions are not send to PowerPC cycle-accurate simulator, only to PowerPC functional simulator. The rest of the host instructions, called PowerPC code, are the instructions that correspond to the execution of guest instruction, in this case x86 move.

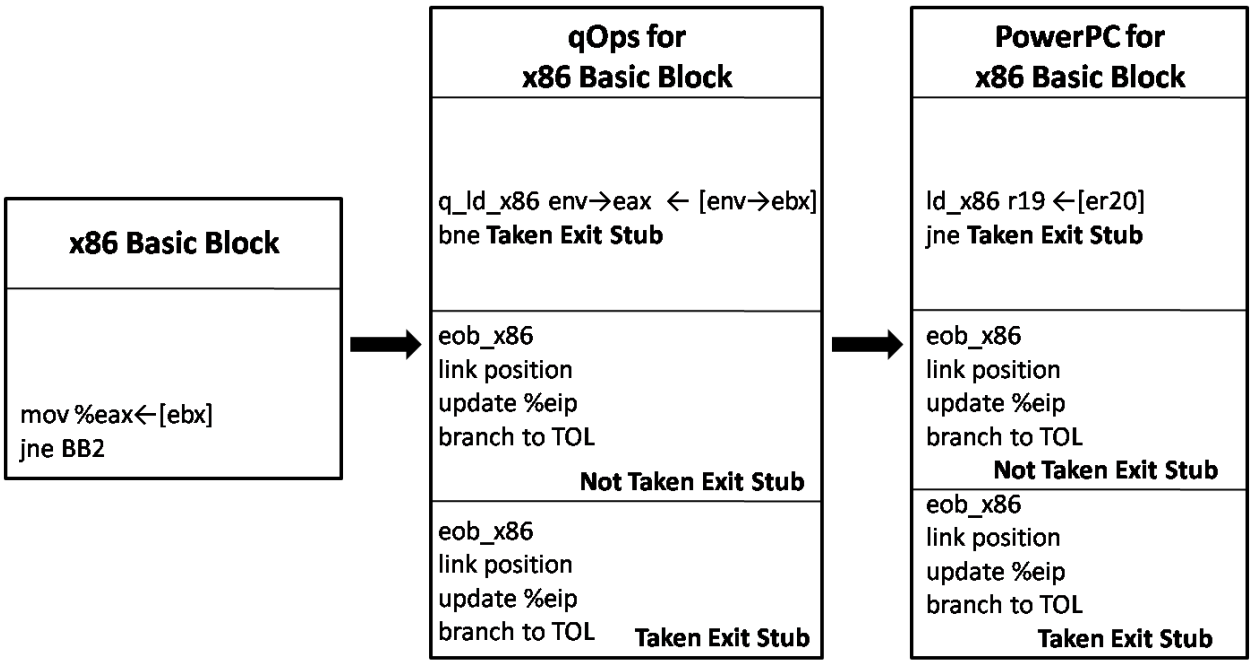

Figure 3.5: Translation flow of a simple *mov* x86 instruction.

Execution of the guest application is done through three optimization stages: interpretation (I), basic block translation (BB) and superblock optimizations (SB), as it is done in most of the Dynamic Binary

Translators (DBTs). In order to support promotions, TOL has two kinds of profiles: edge profilers and dynamic execution counters. The promotion is illustrated in the figure 3.6. The TOL starts by interpreting the x86 instructions. If the execution counter of particular basic block is greater than a predefined threshold (TH_{BB}), that basic block is promoted to BB translation, with some basic optimizations. Furthermore, if that counter is greater than a second predefined threshold (TH_{SB}), the basic block is considered to be hot, and it becomes the starting point of superblock. Superblocks are created based on *edge profilers*, which profile the biasing of the branches for a particular basic block. The superblock is built when the edges between basic blocks in the superblock are highly biased. Experimentally, it has been shown that the optimal threshold values are: TH_{BB}=5 and TH_{SB}=10000, as it is shown in [59]. Superblocks are created by including basic blocks until the probability to reach the end of the superblock is below 80% based on profiling information. More specifically, there are several ending conditions for a SB:

1. Probability to exit the superblock before the end has to be less than a threshold.
2. The outcome of the branch is not biased according to the bias threshold.
3. The biased direction is the beginning of a new superblock.
4. The last basic block in the superblock has a backward branch.

The translator prepares the superblock in qOps and then forwards it to the optimizer.

As the most frequent part of the code, superblock passes through several conventional optimizations:

1. copy/constant propagation
2. constant folding
3. common sub-expression elimination
4. dead code elimination
5. linear register allocation
6. list instruction scheduling [6, 52]
7. aggressive control speculation
8. aggressive data speculation.

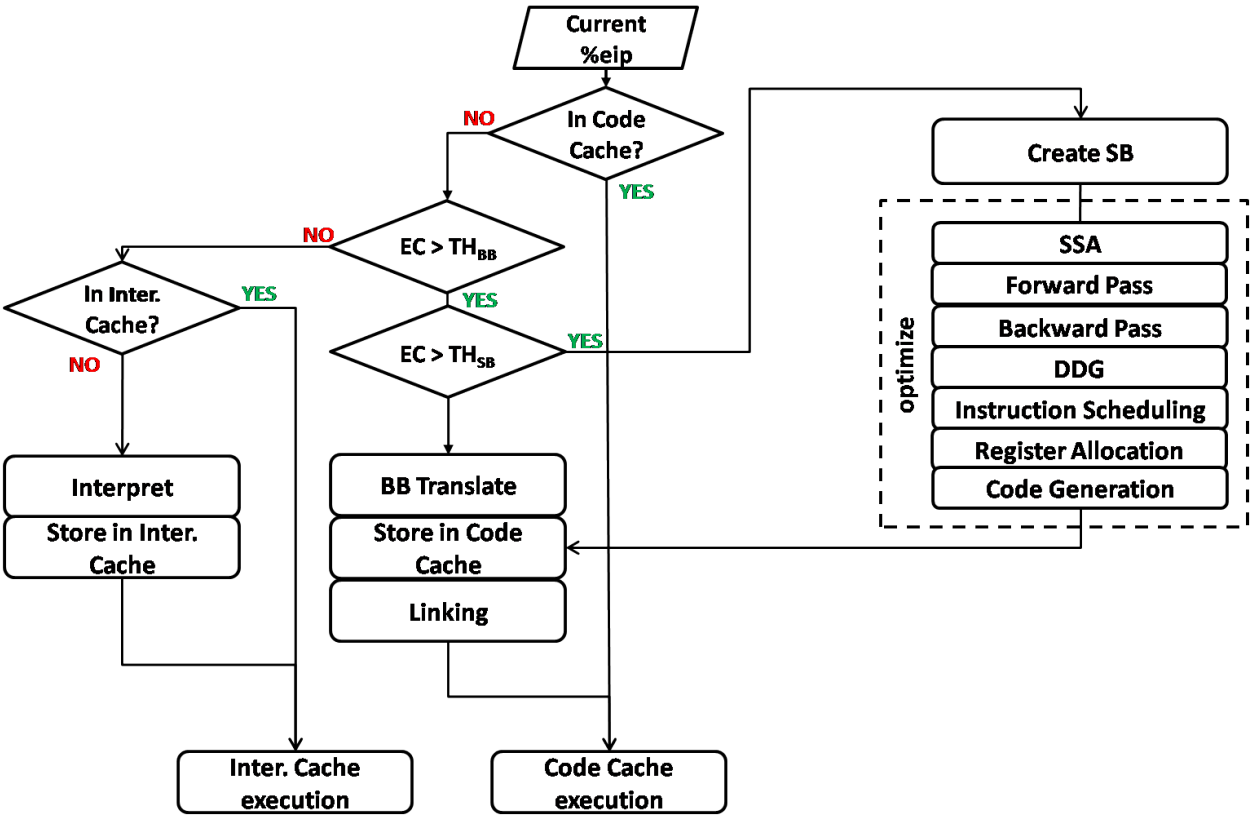

Figure 3.6: Code region promotion.

Moreover, the TOL also performs other well known DBT optimizations such as code regions linking [10] and indirect branch devirtualization [37].

In order to ease the development, the interpretation is modeled as the translation of each instruction and the translated code is stored in the part of memory called “interpretation cache”.

The introduced TOL is based on the QEMU emulator [4], but mainly it is an in-house development. To address the desirable model, the QEMU emulator is extended with profilers, promotions to different optimizations levels and conventional optimization techniques.

The functionality of the TOL modeled in this thesis is split into six main categories: (*i*) interpretation, (*ii*) basic block translation, (*iii*) superblock optimization, (*iv*) code cache look-up, (*v*) code regions linking and (*vi*) others. In order to better understand the modules division, we show Figure 3.7. In additional to baseline classification, the shaded box describes the most frequent execution path, which corresponds to the steady state. Further, these six main categories are divided into 19 smaller specialized *modules* as follows.

1. **Interpretation**

   This category contains only one module, since the process of the interpretation has very light overhead.

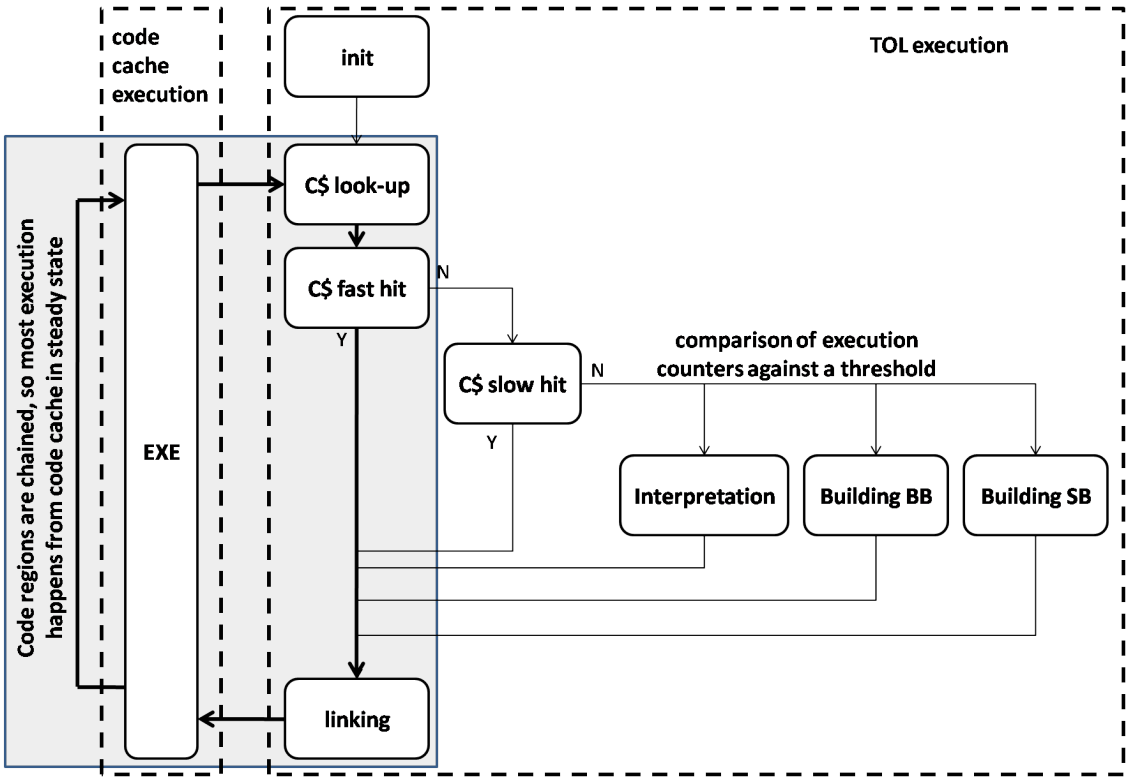

Figure 3.7: Modules and code cache in execution flow.

- *Interpreter Translation*

  This is the part of the TOL which translates the guest instructions and writes the translated instructions in the interpretation cache. Each guest instruction has a particular set of host instructions that emulates the functionality.

2. **Basic Block Creation**

Similar to the Interpretation group, this category corresponds to the basic block translation overhead. This overhead is divided into two parts.

- *BB Translator*

  Translate *guest* instructions to intermediate representation instructions (IR) that are used to create host code which is going to be stored into the code cache.

- *BB Generation Code*

  Generate code from the IR and write it into the code cache.

3. **Superblock Creation and Optimization**

Similar to the previous two categories, this category corresponds to the superblock translation and optimization overhead. It contains the following modules.

- *SB Init*

  Some initialization instructions before starting the SB creation and optimization.

- *SB Region Preparation*

  The construction of the hot regions, in this case superblocks. This module includes analysis of previously executed basic blocks and looks for a candidates that can be part of the superblock.

- *SB Region Formation*

  Convert the region into a superblock.

- *SB Forward Pass*

  Perform the conventional optimizations by traversing the SB top down: copy/constant propagation, constant folding, common sub-expression elimination.

- *SB Backward Pass*

  Perform dead code elimination by traversing the superblock bottom up.

- *SB DDG / MemAliasing*

  Build the Data Dependency Graph (DDG) and perform Memory Disambiguation to add dependencies between memory instructions.

- *SB Instruction Scheduling*

  Schedule the IR code using a List Scheduling Algorithm, according to DDG and hardware units latency.

- *SB Register Allocation*

  Map variables from IR into architectural registers, using a Linear Scan Register Allocation.

- *SB Generation Code*

  Generates and store translated code into code cache.

4. **Code Cache Lookup**

   This category is in charge of the code cache Look-Up. When the TOL encounters the next PC, it first checks if the corresponding basic block or superblock has already been translated and stored in the code cache. This process is called cache look-up. Since the interpretation in DARCO is cached, we have two look-ups: one for the interpretation cache and another for the code cache. The code cache lookup contains the following modules.

   - *I$ Find*

     Look-up into interpretation code cache.

- *C$ Find*

  Initialization for the look-up into the code cache.

- *C$ Fast Hit*

  The look-up of TOL studied in this thesis contains two hash tables, as QEMU does [4]. The first lookup table contains the information about recently translated code regions, while the second lookup table contains information about other code regions. The overhead for checking the first hash table is represented by this module. On average in 95% of the cases the is a hit in the first look up table, and therefore it is called Fast Hit.

- *C$ Slow Hit*

  The overhead which represents the checking of the second, more complex, look-up table. This second, more complex look-up takes place if the first look-up fails.

5. **Code Regions Linking**

   As explained in Section 2.1 the linking optimization is used in order to avoid TOL intervention in the cases when it is not needed. Such cases are for instance when a basic block ends with direct branch and the next basic block (or a code region) is already translated and stored in the code cache. The TOL overhead which is responsible to perform the linking between the particular code regions (basic block or superblock) is covered by this category and it contains only one module:

   - *Linking*

     Check whether basic block (or superblock) can be linked with the code regions executed before and write the corresponding linking instructions (direct branches) in the code cache.

6. **Others**

   This category corresponds to the parts of the TOL that cannot be classified into the five main categories listed above. It contains the following modules.

   - *Init Code*

     Initialization part of the TOL.

   - *TOL Main Loop*

     Contains the main loop of the TOL and conditional instructions for directing the flow of the TOL.

Assuming that the guest code is already translated, the only functionalities that the TOL has to perform are the *C$ look-up* and the *Linking*. Usually, if the basic block or superblock is in the code cache, it is found directly by the *C$ Fast Hit* module. The steady state does not include the translation overheads, so the *Interpretation*, the *BB creation* and the *SB creation* are excluded from the most frequent path.

In order to distinguish the TOL execution between different modules, we insert specialized "fake instructions" between the modules. As discussed in Section 3.1, these fake instructions are not sent to PowerPC cycle-accurate simulator. When the PowerPC component receives the specialized instructions, it is able to switch appropriate counters to the one that correspond to particular module. In that way the statistics for each module are kept separately.

3.2.2 Host Hardware

The host processor is modeled as a simple in-order processor with the configuration which is shown in Figure 3.8. The reasoning behind selecting simple in-order hardware is the fact that the one of the goals of HW/SW co-designed processors is to offload complex functionalities in software and therefore making the hardware design much simpler. The closest industrial processor to the simulated processor is PowerPC 440 [54].

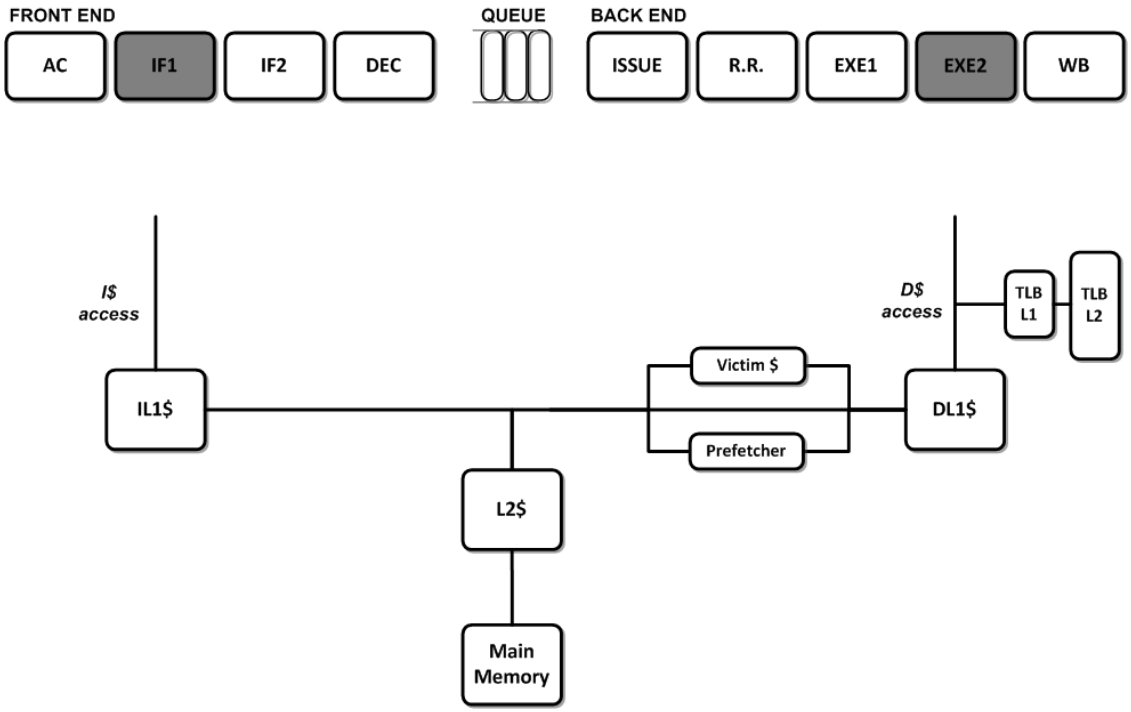

Figure 3.8: In-order pipeline with memory organization.

The pipeline has 9 stages that are distinguished into two parts: the *front-end* and the *back-end*. These

two parts are decoupled by issue queue.

The *front-end* has a purpose to read, decode and store the instructions. It contains four stages: Address calculation stage (AC), Instruction fetch stage 1 (IF1), Instruction fetch stage 2 (IF2) and Decoder (DEC).

- In AC stage PC register value is assigned. There are three possibilities for the PC value: (*i*) the next PC, (*ii*) the target of taken branch and (*iii*) the address of the wrong path during branch misspredicition.
- During IF stage (IF1 and IF2) entire cache line is loaded in the read buffer. The latency of instruction cache is assumed to be 2 cycles (IF1 and IF2). During every instruction cache access, maximum ISSUE WIDTH number of instructions is loaded. This number could be smaller due to different alignment.
- In DEC stage the loaded instructions are decoded. It classifies instructions in eight different groups: simple integer (ALU INT), complex integer (MUL INT), simple floating point (ALU FP), complex floating point (MUL FP), vector instructions (VEC), branches (BR), loads (LD) and stores (ST). If it encounters a branch, the branch predictor is employed and it sends a next PC address to the AC stage. Otherwise nothing is sent to AC stage and instructions are sent to a queue to be executed in the back-end part of the pipeline.

The *back-end* is in charge for issuing and execution of instructions. It contains five stages: Issue stage (ISSUE), Read register stage (RR), Execution stage 1 (EXE1), Execution stage 2 (EXE2) and Write-back stage (WB).

- Data hazards and execution units' capacity are checked in Issue stage (ISSUE).
- During Read register stage (RR) the scoreboard is checked for a status of registers and execution units usage.
- In Execution stages (EXE1 and EXE2) the instruction is executed. There are five different functional units where instructions can be executed: simple integer unit (ALU INT), complex integer unit (MUL INT), simple floating point unit (ALU FP), complex floating point unit (MUL FP) and vector unit (VEC).
- The results are written in the registers Write-back stage (WB).

The host processor has 64 registers, grouped in 2 sets with 32 registers. The first set is used for TOL execution, while the second is used for translated application execution.

We model two levels of cache memory hierarchy with separated instruction and data cache on the first level. IF1 and EXE2 are the stages where memory access happens respectively for Instruction and Date cache. They are marked by darker boxes (Figure 3.8). TLB is assumed only for data memory (because the instruction TLB misses are negligible) and it is two levels TLB. In addition we model stride data prefetcher and victim cache.

As previously mentioned, the modeled processor has the scoreboard mechanism to stall instruction execution in in-order processor. Each of registers and functional units has its own entry which indicates if the register or functional unit is currently used or not.

Branch prediction is supported by gshare branch predictor and it is done for both types of the branches: direct and indirect. In the case of direct branches, only branch decision (taken or not taken) is predicted. In the case indirect branches the address of a branch is predicted as well by storing the previous address in a hash table. Wrong paths for both cases are modeled.

All microarchitectural parameters, which make a default configuration, are summarized in Table 3.1.

3.3 Simulation Error and Simulation Time Reduction

In all contribution chapters (Chapters 4, 5 and 6) we compare the proposed simulation techniques with the authoritative simulation (AS) in terms of *simulation error* and *simulation time reduction*. The goal is to have the minimal *simulation error* with the maximal possible *simulation time reduction*.

It is very important to know for which metric we have to compute the simulation error. The typical reported metric for performance in computer architecture is Cycles Per Instruction (CPI). For HW/SW co-designed processors, CPI can refer to the guest CPI (gCPI) and the host CPI (hCPI) depending on the type of counted instructions. The more important metric is gCPI where instructions refer to guest instructions (in our case x86 instructions) because it allows comparing two different configurations for HW/SW co-designed processors. On the other hand hCPI is the CPI metric where instructions refer to host instructions (in our case PowerPC-like instructions). Different setups for HW/SW co-designed processors leads to different number of host instructions. Therefore, this metric can be used only to observe the performance on the host hardware but it cannot be used to compare different configurations of HW/SW co-designed processors. Therefore when we report the simulation error, we are focused on the error of gCPI.

However, in this thesis we are not limited to the gCPI error only. The errors for other metrics, such as number of host instructions or the percentage of dynamic application stream executed in SB mode (SB coverage), are also computed and presented. This is done in order to ensure that alternative simulations

Table 3.1: Microarchitectural Parameters.

Component	Parameter	Value
General	Issue width	2
	Issue queue size	16
	ALU INT latency	1
	MUL INT latency	2
	ALU FP latency	2
	MUL FP latency	5
	VECTOR latency	10
Branch predictor (G-share)	Size of history register	12
Cache I1 (32KB)	Number of sets	64
	Block size	64
	Associativity	4
	Replacement policy	LRU
	Hit latency	1
Cache D1 (32KB)	Number of sets	64
	Block size	64
	Associativity	4
	Replacement policy	LRU
	Hit latency	1
Cache L2 (512KB)	Number of sets	512
	Block size	128
	Associativity	8
	Replacement policy	LRU
	Hit latency	16
	Main Memory hit latency	128
Stride Prefetcher	Number of entries	32
TLB L1	Size of block	8
	Number of sets	8
	Replacement policy	LRU
	Hit latency	1
TLB L2	Size of block	32
	Number of sets	8
	Replacement policy	LRU
	Hit latency	16
	Miss latency	512
Number of execution units	Number of ALU INTs	2
	Number of MUL INTs	2
	Number of ALU FPs	2
	Number of MUL FPs	2
	Number of VECTORs	2
Number of memory ports	Read	1
	Write	1

deliver accurate results for other statistics which are relevant for HW/SW co-designed processors. There are many scenarios where focusing only on gCPI error may lead to wrong conclusions. They are going to be explained later in this thesis, in Section 5.6.

The simulation error for *gCPI* and for *the number of host instructions* are calculated as relative errors between the authoritative simulation (AS) and alternative simulation technique (technique):

$$error_{gCPI}^{tecnhique} = \left| \frac{gCPI^{tecnhique} - gCPI^{AS}}{gCPI^{AS}} \right|, \tag{3.1}$$

$$error_{\#instr}^{tecnhique} = \left| \frac{\#instr^{tecnhique} - \#instr^{AS}}{\#instr^{AS}} \right|. \tag{3.2}$$

On the other side, if the metric is expressed as a ratio, as it is the case in SB coverage, the error is defined as absolute error:

$$error_{SB\%}^{tecnhique} = \left| SB\%^{tecnhique} - SB\%^{AS} \right|. \tag{3.3}$$

Simulation time reduction is simply defined as the ratio between the simulation time for authoritative simulation and alternative simulation (technique):

$$timeReduction = \frac{time^{AS}}{time^{tecnhique}}, \tag{3.4}$$

The simulation time is measured in terms of the number of *host* instructions. We avoid using wall-clock time to measure the simulation time as our simulations run in a cluster system where machine configurations are very different, and we do not have explicit control on where specific simulations are executed. The number of guest instructions is not valid metric to estimate simulation time because different setups for HW/SW co-designed processors can lead to a different number of host instructions and the number of host instructions is the metric that has influence on total simulation time. For informational purpose, simulation time of the host cycle-accurate simulator is around 1M instructions per second.

3.4 Benchmark Suites

In this thesis we use 4 benchmark suites in order to evaluate our proposed simulation techniques: SPEC2006INT, SPEC2006FP, MediaBench and PhysicsBench. These suites allow a broader scope of experiments. Every utilized benchmark is a single threaded application. Benchmarks are compiled with gcc 4.2.0 and run on top of an unmodified Fedora 5 operating system.

Except the benchmark's name, SPEC benchmarks originally come with the number, which gives the following representation of the benchmark: *number.name*. Therefore we represent MediaBench and PhisicsBench in same manner. MediaBench benchmarks are represented with 0xx.*name*, while PhysicsBench benchmarks are represented with 1xx.*name*.

SPEC benchmark suite [5] is widely used in the research community to evaluate the performance

of computer systems. SPEC stands for the set of benchmarks given by Standard Performance Evaluation Corporation and SPEC2006 is the latest announced version. SPEC2006 contains two suites: SPEC2006INT and SPEC2006FP, which basically refer to the number of floating points instructions that appear in the benchmark's instruction stream. If more than 30% of dynamically executed instructions are floating points instructions, the benchmarks is defined as floating point benchmarks (FP), otherwise it is defined as integer benchmark (INT). In total we simulate 28 SPEC benchmarks. The applications and their input parameters are summarized in Table 3.2.

Table 3.2: SPEC Benchmarks used in simulations along with the most relevant execution command-line arguments.

Benchmark	Input Parameters
400.perlbench	-I. -I./lib checkspam.pl 2500 5 25 11 150 1 1 1 1
401.bzip2	chicken.jpg 30
403.gcc	166.i -o 166.s
410.bwaves	
429.mcf	inp.in
433.milc	< su3imp.in
434.zeusmp	
435.gromacs	-silent -deffnm gromacs -nice 0
436.cactusADM	benchADM.par
437.leslie3d	< leslie3d.in
444.namd	–input namd.input –iterations 38 –output namd.out
445.gobmk	–quiet –mode gtp ¡ 13x13.tst
447.dealII	23
450.soplex	-m3500 ref.mps
453.povray	SPEC-benchmark-ref.ini
454.calculix	-i hyperviscoplastic
458.sjeng	ref.txt
459.GemsFDTD	
462.libquantum	
464.h264ref	-d foreman_ref_encoder_main.cfg
470.lbm	3000 reference.dat 0 0 100_100_130_ldc.of
471.omnetpp	omnetpp.ini
473.astar	rivers.cfg
481.wrf	
482.sphinx3	ctlfile . args.an4
483.xalancbmk	-v t5.xml xalanc.xsl
998.specrand	1255432124 234923
999.specrand	1255432124 234923

MediaBench [2] is representative of real workloads that can be found in media or embedded processors. In total there are 12 benchmarks and they are summarized in Table 3.3 together with input parameters.

PhysicsBench [74] is a suite that models the future interactive entertainment application with many interactive objects using explosions, breakable objects and cloth effects. In total there are 8 benchmarks and they are summarized in Table 3.4. Notice that these benchmarks do not have input parameters.

Table 3.3: MediaBench benchmarks used in simulations along with the most relevant execution command-line arguments.

Benchmark	Input Parameters
000.cjpeg	-quality 13 -outfile output_base_4CIF_96bps.jpg input_base_4CIF.ppm
001.djpeg	-ppm -outfile output_base_4CIF_96bps_dec.ppm input_base_4CIF_96bps.jpg
002.h263dec	-o3 input_base_4CIF_96bps.263 output_base_4CIF_96bps_dec_%03d
003.h263enc	-x 4 -a 0 -b 8 -s 15 -G -R 30.00 -r 3508000 -S 3 -Z 30.0 -O 0 -i input_base_4CIF_0to8.yuv -o output_base_4CIF_96bps_15.raw -B output_base_4CIF_96bps_15.263
004.h264dec	input_base_4CIF_96bps_decoder.cfg
005.h264enc	-d input_base_4CIF_96bps_encoder.cfg
006.jpg2000dec	-f input_base_4CIF_96bps.jp2 -F output_base_4CIF_96bps_dec.ppm -T pnm
007.jpg2000enc	-i input_base_4CIF.ppm -F output_base_4CIF_96bps.jp2 -T jp2 -O rate=0.010416667
008.mpeg2dec	-b input_base_4CIF_96bps.mpg -o3 output_base_4CIF_96bps_%03d
009.mpeg2enc	input_base_4CIF_96bps_15.par output_base_4CIF_96bps_15.mpg
010.mpeg4dec	-y -s 4cif -r 30 -an -g 9 -bf 3 -i input_base_4CIF_96bps.avi -s 4cif -an -y output_base_4CIF_96bps_dec_%03d.ppm
011.mpeg4enc	-s 4cif -r 30 -i input_base_4CIF_%03d.ppm -r 30 -s 4cif -b 39 -bf 3 -vcodec mpeg4 -an -vstats -y output_base_4CIF_96bps_15.avi

Table 3.4: PhysicsBench benchmarks used in simulations.

Benchmark	Input Parameters
100.breakable	/
101.continuous	/
102.deformable	/
103.everything	/
104.explosions	/
105.highspeed	/
106.periodic	/
107.ragdoll	/

In order to understand the reasons behind different simulation error vs. simulation time reduction properties across different applications, we provide Table 3.5 that characterizes all the benchmarks that are utilized in this thesis. For each benchmark the following statistics are given: percentage of TOL overhead (TOL%), dynamic instruction stream vs. static instruction ratio in term of guest x86 instructions (d/s), number of unique superblocks (uSB), percentage of dynamic instructions being executed in superblock mode (SB%) and number of dynamically executed guest instructions until the completion (d). Each of these characteristics is responsible for the particular behavior that is going to be discussed during the analysis of different simulation techniques.

Table 3.5: Benchmarks' Characterization.

Benchmark	TOL%	d/s	uSB	SB%	d
400.perlbench	41.8	47811.1	3164	98.0	0.15E12
401.bzip2	12.2	326714.6	227	99.9	0.19E12
403.gcc	25.2	36199.3	4917	97.5	0.08E12
410.bwaves	4.6	232291.2	173	99.7	2.7E12
429.mcf	3.8	350410.1	269	98.8	0.38E12
433.milc	2.7	120065.4	842	99.5	1.28E12
434.zeusmp	2.7	120065.4	842	99.8	2.45E12
435.gromacs	4.5	105758.1	714	99.6	2.22E12
436.cactusADM	3.7	104440.2	219	99.8	2.22E12
437.leslie3d	1.2	114625.2	334	99.6	3.42E12
444.namd	5.1	195898.5	483	99.6	2.88E12
445.gobmk	34.9	59936.6	3666	98.5	0.23E12
447.dealII	0.3	84909.9	49	100.0	2.30E12
450.soplex	18.3	167945.7	341	99.8	0.53E12
453.povray	28.3	132895.8	611	99.4	1.21E12
454.calculix	6.3	67886.1	1923	98.8	8.24E12
458.sjeng	32.0	211114.2	1039	99.6	2.45E12
459.GemsFDTD	9.9	88328.2	1207	99.1	2.29E12
462.libquantum	1.7	384910.1	38	100.0	2.93E12
464.h264ref	21.8	81957.9	1408	98.8	0.32E12
470.lbm	4.1	388655.7	42	99.9	0.14E12
471.omnetpp	15.3	120006.9	1066	99.5	0.75E12
473.astar	1.1	212989.5	310	99.9	0.41E12
481.wrf	6.0	33216.6	542	99.6	4.26E12
482.sphinx3	9.4	122704.2	866	99.6	2.85E12
483.xalancbmk	38.2	43966.3	1750	99.1	1.21E12
998.specrand	43.3	101380.2	170	97.7	0.61E9
999.specrand	43.3	101380.2	170	97.7	0.61E9
000.cjpeg	64.5	4224.9	17	85.4	0.06E9
001.djpeg	73.1	2865.5	19	83.9	0.04E9
002.h263dec	37.2	11762.0	163	87.1	1.05E9
003.h263enc	8.0	206072.5	410	99.7	0.02E12
004.h264dec	23.8	94815	877	96.1	3.91E9
005.h264enc	19.6	124288.2	1474	98.8	0.15E12
006.jpg2000dec	23.0	17843.2	98	94.8	0.41E9
007.jpg2000enc	25.7	27796.4	460	93.5	0.78E9
008.mpeg2dec	32.9	86646.4	240	97.4	1.05E9
009.mpeg2enc	5.4	205313.0	411	99.7	0.02E12
010.mpeg4dec	52.6	4274.9	140	80.7	0.47E9
011.mpeg4enc	21.7	8189.6	351	78.3	0.45E9
100.breakable	12.2	52540.7	466	93.7	1.95E9
101.continuous	27.2	9502.5	314	82.8	0.33E9
102.deformable	20.7	61430.4	728	95.7	1.76E9
103.everything	13.9	93620.0	861	97.6	3.91E9
104.explosions	8.2	119273.8	512	97.4	3.54E9
105.highspeed	4.9	117295.4	480	97.8	3.42E9
106.periodic	17.1	8731.2	223	81.0	0.27E9
107.ragdoll	21.3	5585.7	132	77.1	0.18E9

Chapter 4

Variability and Predictability of the Transparent Optimization Software Layer

This chapter analyses the variability and predictability of the behavior of the Transparent Optimization software Layer (TOL) and concludes that it is not constant and that cannot be predicted. Therefore this chapter suggests that TOL has to be simulated, instead of being ignored or assumed constant as it is the case the literature. The chapter is organized as follows. Section 4.2 discusses the related work. Section 4.3 presents the variability studies, while predictably studies are given in Section 4.4. Evaluation of different simulation approaches is given in Section 4.5. Finally section 4.6 summarizes the chapter.

4.1 Introduction

As it was explained in Section 2.3.2, recent papers that are focused on HW/SW co-designed processors consider the behavior of the TOL to be constant or ignored. For these reasons this chapter analyzes whether TOL behavior is constant or not in terms of cycles per host instruction (hCPI). Please notice that constancy of the TOL is not limited only on the number of cycles, but it looks into microarchitectural behavior of the TOL on the top of the host processor. More details about this are given in Section 4.3. We show that the hCPI varies significantly across the different guest applications that are studied. Consequently, the assumption of constant or ignored TOL could lead to wrong results and conclusions.

According to our experimental data there are three main reasons behind the variance of the TOL behavior. The first is the microarchitectural interaction between the TOL and the guest application. In order to understand this behavior, we have isolated the behavior of the TOL from the application being emulated.

The results show that the microarchitectural interaction between the emulated application and the TOL not only is significant but also it is not constant (*i.e.* varies for the different guest applications). The interaction is in the order of 10% of the TOL CPI.

The second reason is related to the fact that different applications stress differently the modules of the TOL. Some modules, especially the ones that have a significant contribution to the TOL overhead, have been found to vary significantly across the different workloads. For example the interpreter coverage was found to vary from 5% to 75%.

The third reason is the fact that after excluding the interaction between the TOL and the code cache, the measured CPI for the different modules still shows variability which is different for each module. This variability was found to be more significant for the modules that have lower contribution to the TOL.

By showing that the TOL does not have a constant behavior and that the percentage of the TOL overhead is not always small, this chapter suggests that the TOL cannot be excluded from the simulation process as this could easily result in misleading conclusions.

In addition to studying the variability of the TOL we also try to predict the behavior it will have based on high-level statistics of the emulated *guest* application. Our results show that there exists some correlation between the high-level *guest* binary statistics and the behavior of the modules that are stressed the most. Unfortunately, the modules used for code region optimizations show smaller correlations. For the applications which will stress more these modules, the predictability with high-level statistics is not possible. All the simulation approaches that try to ease the simulation effort and which are analyzed in this chapter are summarized in Figure 4.1: constant, ignored and predicted TOL behavior. .

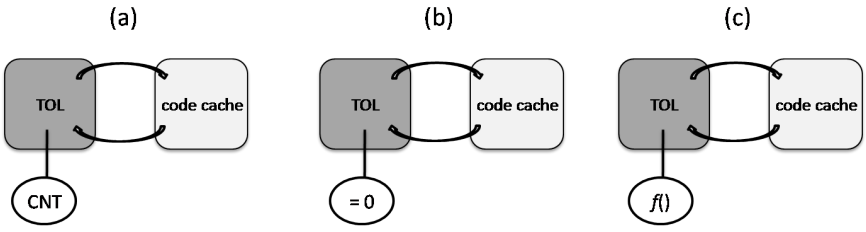

Figure 4.1: Simulation techniques that avoid TOL simulation.

Besides the analysis of constant or/and ignored TOL behavior and its predictability, this chapter also provides a microarchitectural characterization of the TOL modules. It shows which modules have the worst microarchitectural behavior and where developers need to pay special attention while developing such systems.

In short, the main contributions of this chapter are:

1. We analyze the microarchitectural behavior of the TOL at the level of the SW layers modules for 48 different applications (SPEC2006INT, SPEC2006FP, MediaBench and PhysicsBench).

2. We show that the microarchitectural behavior of the SW layer has significant variability and we explain the reasons, which are the interaction with *guest* application and the input dependency.

3. We study the possibility of predicting this variability with high-level *guest* statistics.

4. We prove that the simulation of a TOL is necessary as otherwise the conclusions drawn might be misleading.

4.2 Related Work

The relevant work to the one presented in this chapter can be classified into two major parts: characterization of DBT systems and deterministic behavior of JVM (Java Virtual Machine). Although there is a large number of publications that quantify the overhead in DBT architectures [20, 50, 22, 10, 26] the authors do not classify the source of this overhead. Other studies focus on one specific overhead: interpretation overhead [50, 68], branch resolution [70], indirect branches overhead [32] etc. Our results are more detailed as they are not limited to the global overhead but show the breakdown. We compared our data with [32] which also uses SPECINT2006 and found the trends of the total overhead to be the same.

Borin and Wu provide a detailed breakdown of the overhead [14]. The authors launched StartDBT on real hardware and measured the individual overhead. They modified the StarDBT, executing only particular module of the DBT, while the execution of the other modules is faked by taking results from previous run. The StarDBT has 2 level of promotion and results are given for SPEC2000. The paper does not show the microarchitectural behavior of different modules across different applications. It states that code duplication and Return Address Stack (RAS) related overhead cause on average 7.2% of the overhead, due to the fact that call and return do not use the RAS.

In the past, the DBT's non-deterministic behavior was studied for JVM. The main unpredictability is caused by the non-deterministic behavior of the garbage collector (GC) algorithm [12, 31], not by hardware interactions. Sweeney *et al*. [69] point to the unknown performance characteristics of the Java programs, such as significant IPC improvement for an adaptive configuration and significant IPC degradation before GC.

4.3 TOL Variability

The Transparent Optimization software Layer (TOL) can be seen as independent application executed on the host hardware. Therefore the entire performance of the HW/SW co-designed processor can be represented as:

$$gCPI = gCPI_{TOL} + gCPI_{C\$} \tag{4.1}$$

$$gCPI = \frac{hInsturctions_{TOL}}{gInsturctions} hCPI_{TOL} + gCPI_{C\$} \tag{4.2}$$

As it can be seen the performance of TOL has two sources of variance: $\frac{hInsturctions_{TOL}}{gInsturctions}$ and $hCPI_{TOL}$. The first source ($\frac{hInsturctions_{TOL}}{gInsturctions}$) requires only TOL functional simulation, so can be obtained relatively easily. On the other hand $hCPI_{TOL}$ depends on the cycle-accurate host simulations and therefore requires more time and effort to be obtained. This would mean the following. If the behavior of the TOL is not constant ($gCPI_{TOL}$), the thing that could lead to skipping TOL simulation would be the case when $hCPI_{TOL}$ is constant. Therefore in the rest of the chapter we are focused on $hCPI_{TOL}$.

In order to better understand $hCPI_{TOL}$ behavior, we have run our experiment under two different scenarios: (*i*) the instructions of the TOL and of the emulated application are interleaved (as it would occur in a real system) and (*ii*) only the instructions of the TOL are fed to the cycle-accurate simulator. This is done in order to isolate the TOL to better understand its behavior. In this latter scenario, the interaction between the TOL and the emulated application is removed. In the rest of this chapter we refer to these scenarios as "*with interaction*" and "*without interaction*", respectively (Figure 4.2).

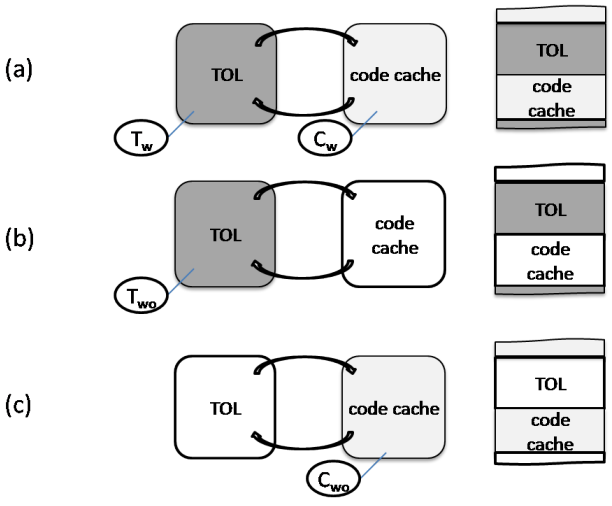

Figure 4.2: Illustration of experiments that determine the interaction between the TOL and the code cache: (a) authoritative simulation, (b) simulation that excludes the code cache and (c) simulation that excludes TOL.

The applications used for these experiments consist of SPEC2006 benchmark suite, MediaBench and PhysicsBench. The simulation period starts at the first instruction of each application because we want to fully study all TOL modules, and some of them are active only at the beginning of the translation process. Therefore we do not skip some amount of instructions. At the same time, in order to avoid these modules being the dominant factor in the simulation period, we simulate a very large number of instructions which is set to be 4 billion x86 instructions. Actually some of our applications have fewer than 4 billion instructions and as such they run to completion.

4.3.1 TOL Behavior at the Global Level

As explained, the variability of TOL is firstly studied by isolating the TOL execution. This would mean that hCPI can be represented as:

$$hCPI_{TOL} = hCPI_{TOL}^{w/o} + interaction, \tag{4.3}$$

where $hCPI_{TOL}^{w/o}$ is the hCPI TOL under "*without interaction*" scenario and *interaction* is mictroarchitectural interaction between the TOL and the code cache.

Figure 4.3 presents the hCPI TOL under the two scenarios ("*with interaction*" and "*without interaction*") and the TOL overhead (the percentage of TOL execution compared to the total execution time). The X-axis presents the applications, sorted according to their CPI for "*with interaction*" scenario. The TOL overhead varies from almost 0% to 80% (secondary Y-axis). This behavior is expected due to various application characteristics, such as number of indirect branches and the variance in the dynamic / static instruction ratio [59].

As can be seen from Figure 4.3, the CPI of the TOL under the "*with interaction*" scenario varies significantly, ranging from 0.7 to 1.6. This wide range is due to three factors. The first factor, which is studied later in isolation, is the fact that the TOL, as most applications, has different execution profile for different inputs (*i.e.* for different guest applications). This is explained in Section 4.3.2.

The second factor is that the different TOL modules behave differently depending on the guest application, *i.e.* the CPI of the TOL modules is not constant among the different guest applications. We study this further later in Section 4.3.2.

The third factor is the microarchitectural interaction between the application and the TOL. Comparing the "*without interaction*" scenario with the "*with interaction*" scenario (Figure 4.3), we can see that the interaction always causes performance degradation, on average by 10%. This is so because although the

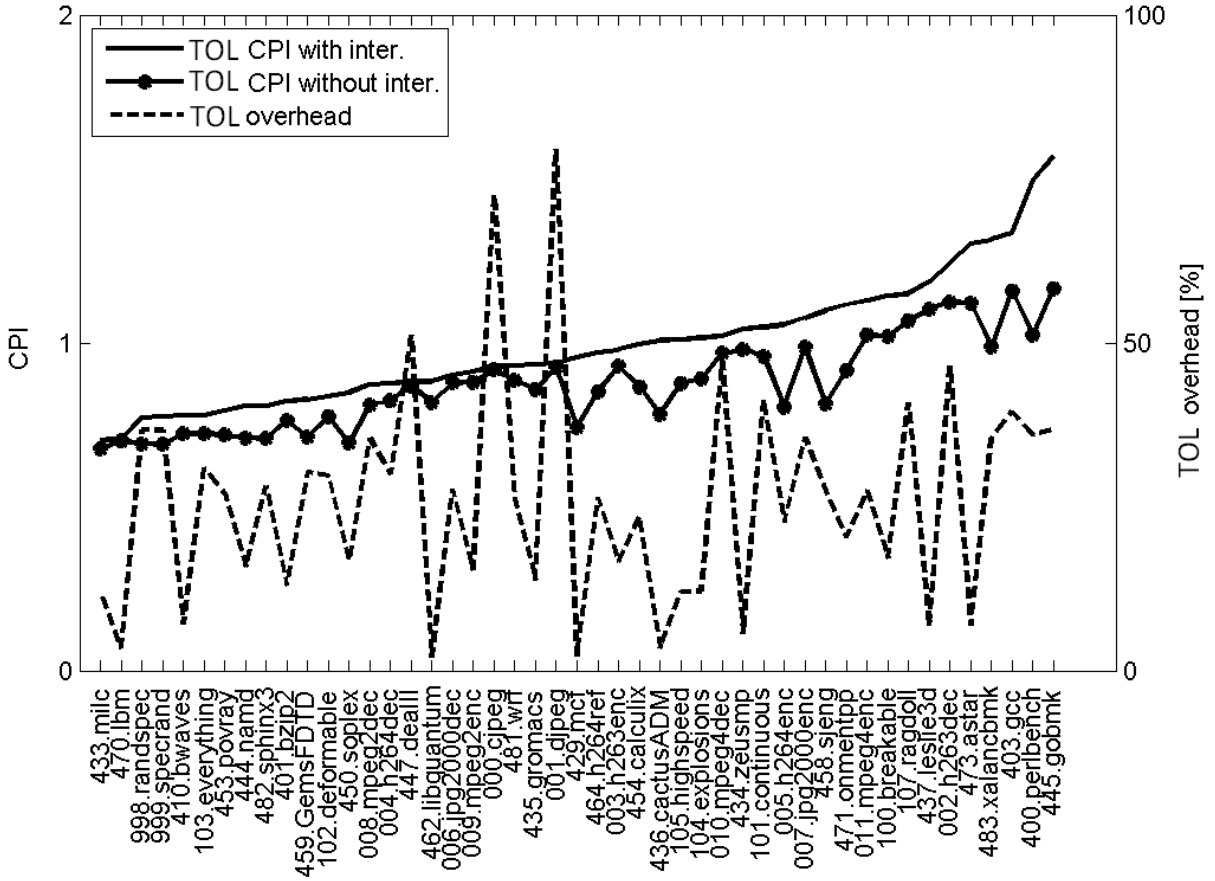

Figure 4.3: TOL CPI "with interaction" and "without interaction". Applications are sorted from the ones with the lower "with interaction" CPI (left) to the ones with the higher CPI. Left part of the figure illustrates the order of executed instructions on the dynamically executed trace on the host.

TOL and the application cooperate to guarantee forward progress, they compete for the microarchitectural resources (caches, branch predictor etc.). Moreover, the effect of the interaction is also not constant among the different applications. The effect of microarchitectural interaction is later discussed in more details in Section 4.3.2.

4.3.2 TOL Behavior at the Level of Modules

In order to further analyze and explain the TOL CPI variance, we start by isolating the TOL modules. The TOL, as an independent application, contains different parts specialized for a particular operation (modules). For different emulated applications, each particular module has different contribution weight. In other words, we can present the total CPI of the TOL as:

$$hCPI_{TOL} = \frac{\sum cycle_{module}}{hInsn_{total}} = \frac{\sum (hInsn_{module} \cdot hCPI^{w/}_{module})}{hInsn_{total}} \tag{4.4}$$

$$hCPI_{TOL} = \sum(\omega_{module} \cdot hCPI^{w/}_{module}) \tag{4.5}$$

$$hCPI_{TOL} = \sum(\omega_{module} \cdot hCPI^{w/o}_{module}(1+\alpha_{module})) \tag{4.6}$$

where $hCPI^{w/}_{module}$ and $hCPI^{w/o}_{module}$ stands for the hCPI of each module for the "*without interaction*" and the "*with interaction*" scenarios respectively. α_{module} stands for interaction coefficient for module, while ω_{module} stands for the contribution weights in overall TOL overhead. They are defined as:

$$\alpha_{module} = \frac{hCPI^{w/}_{module} - hCPI^{w/o}_{module}}{hCPI^{w/o}_{module}}. \tag{4.7}$$

$$\omega_{module} = \frac{Insn_{module}}{Insn_{total}}. \tag{4.8}$$

Notice that they differ across applications due to their different properties. Notice that the contribution weights refer to the number of *instructions* spent in each module and not to the number of cycles.

Variability of the TOL module contribution - ω_{module}

Figure 4.4-a shows the range of contribution weights (ω_{module}) for the different modules where these modules are sorted according the average contribution of all applications. As explained before, the results are collected for 19 modules among 48 applications. The results in Figure 5-a are presented in the form of a boxplot which is a convenient way of graphically depicting groups of numerical data showing their five number summaries: the smallest observation (min), lower quartile, median, upper quartile, and the largest observation (max). The circles in the figure present the outliers, the statistic data which are positioned outside of the expected [min max] interval.

According to the experimental data showed in Figure 4.4-a, which presents the contribution of each module to the total TOL overhead, there are only 6 modules with significant contribution. These are:

- *Interpretation Translation*,
- *TOL Main Loop*
- *C$ Fast Hit*
- *C$ find*
- *BB Generation Code*
- *Linking*.

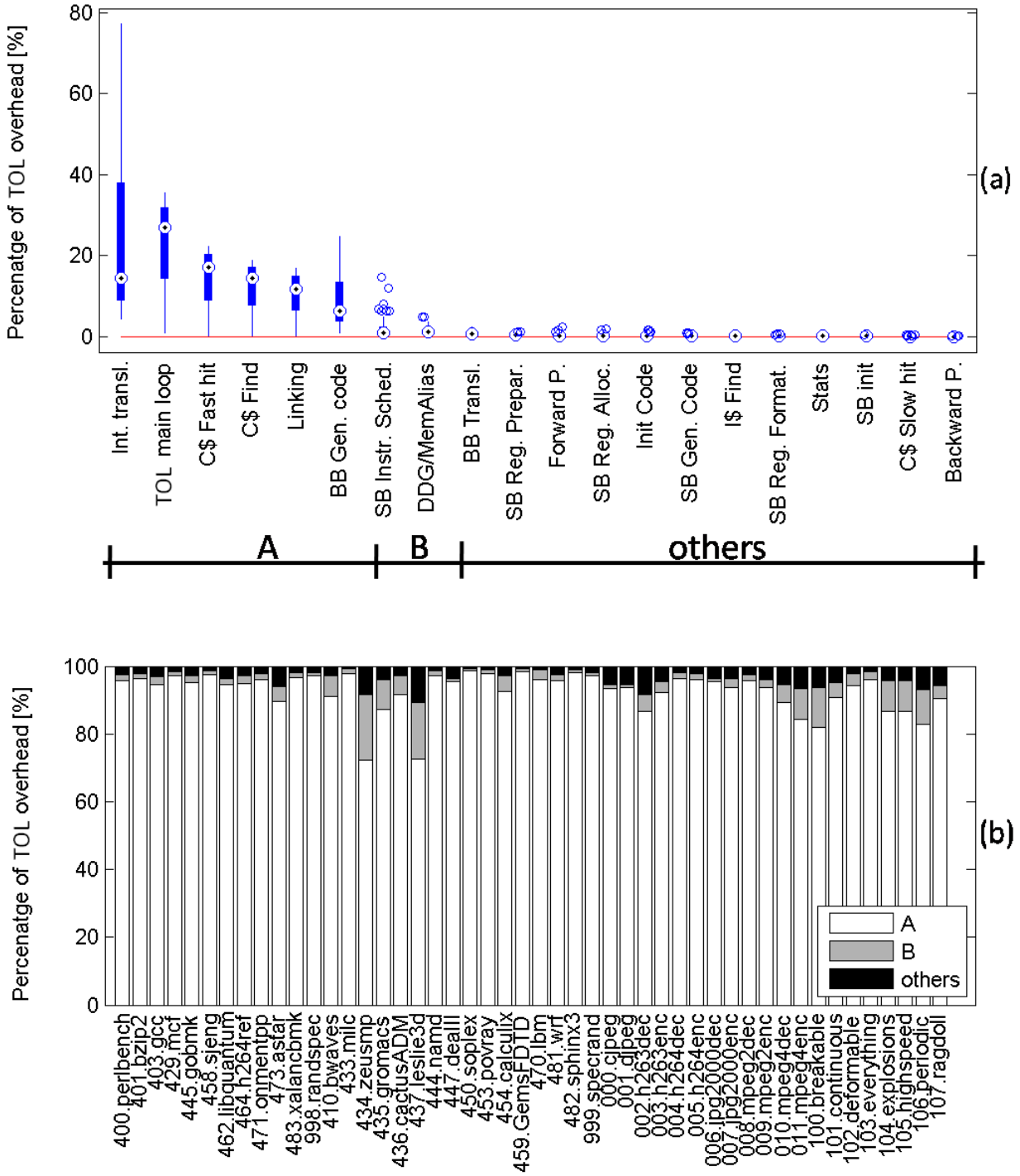

Figure 4.4: Contribution of TOL module with respect to the TOL (ω_{module}): (a) boxplot and (b) contribution across different applications.

and they have a wide range of values. For the purpose of further experimental nomenclature, we will call this group of modules the "*Group A*".

There are two other modules which have an impact in the total overhead close to 10% for some applications. They are:

- *SB Instruction Scheduling*
- *SB DDG / MemAliasing*.

We will name this group the "*Group B*".

Considering groups A and B, Figure 4.4-b depicts the contribution weights breakdown across all applications. For most of the applications, *Group A* covers more than 90% of total overhead. However, notice that the group B is significant for the following applications: 434.zeusmp, 437.leslie3d, 011.mpeg4dec, 100.breakable, 104.explosions, 105.highspeed and 106.periodic. For these applications there is relatively high number of unique superblocks and relatively high number of generated PowerPC instructions. The number of unique superblocks correlated with how many time SB modules (SB Instruction Scheduling and SB DDG / MemAliasing) are called, while the number of generate PowerPC instructions correlated with the execution time of SB modules (SB Instruction Scheduling and SB DDG / MemAliasing).

From the results of Figure 4.4-a it is clear that the TOL modules are not stressed equally for the different guest applications. As an example consider the interpreter, whose contribution ranges from 5% to 75% over the applications studied. As explained before, this is the second factor that causes the behavior of the TOL to vary across applications.

Variability of the TOL module CPI - $hCPI^{w/o}_{module}$

Up to this point, the experimental results show that the contribution of each module to the TOL overhead is application dependent. The question that is answered by the next set of results is whether each particular module has the same microarchitectural behavior (*i.e.* the same CPI) across the different applications. According to the experimental data which are depicted by Figure 4.5, we can observe two types of the modules:

- modules with narrow range of CPI

  (*BB Generation Code*, *Interpreter Translation*, *Init Code*, *SB instruction scheduling*, *SB Region Preparation*)

- modules where CPI varies significantly

  (*SB DDG / MemAliasing*, *SB Init* and *I$ Find*).

In addition, we can observe that the SB creation modules have relatively high CPI, especially the *SB DDG / MemAliasing* module (the median is 10 cycles per instruction).

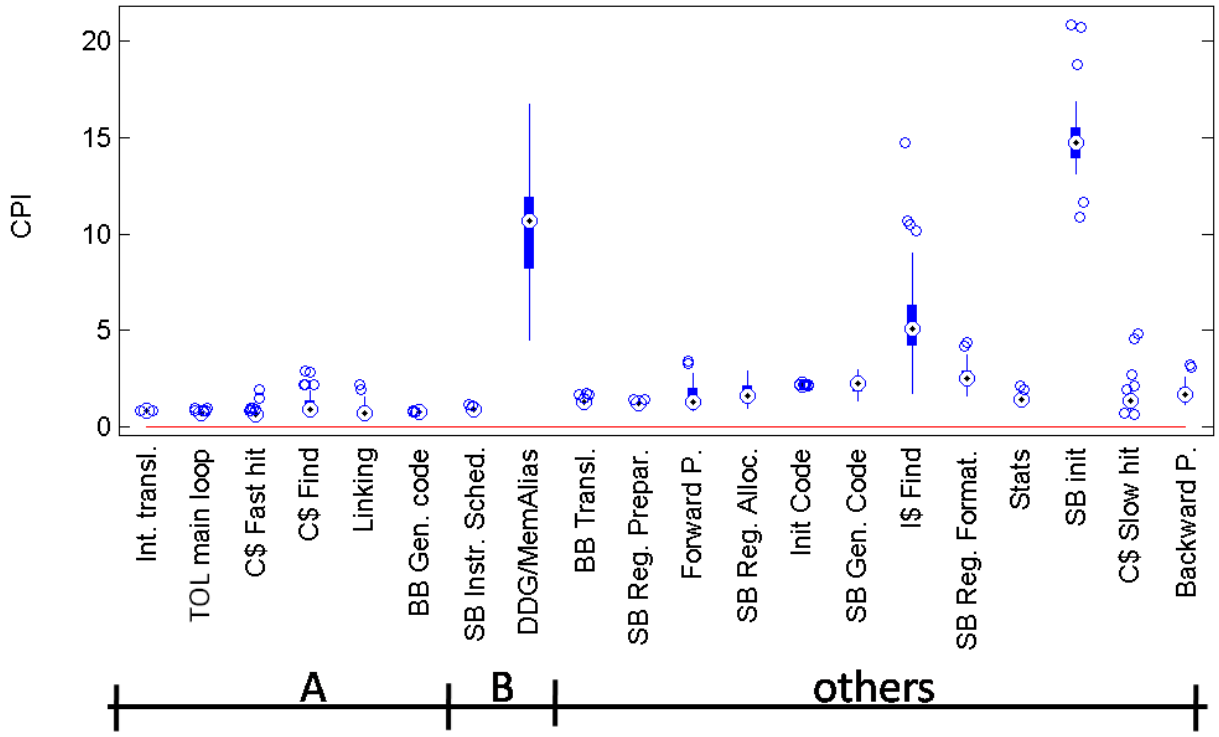

Figure 4.5: TOL CPI per module "without interaction" presented with the boxplot. Modules are sorted from the ones with the higher contribution (left) to the ones with the lower contribution.

In order to better understand the reasons behind the CPI variance we look into more details for the modules of *Group A* (which are the ones with the highest contribution). The experimental results are presented by Figure 4.6. The primary Y-axis represents the CPI_{module} (both cases, "*with interaction*" and "*without interaction*"), while the secondary Y-axis represents the D$ and branch prediction misses per 1K instructions for that particular module "*without interaction*". In all the graphs, the X-axis shows the applications sorted according to the CPI "*without interaction*". The CPIs for the cases "*with interaction*" and "*without interaction*" are marked respectively with squares and stubs.

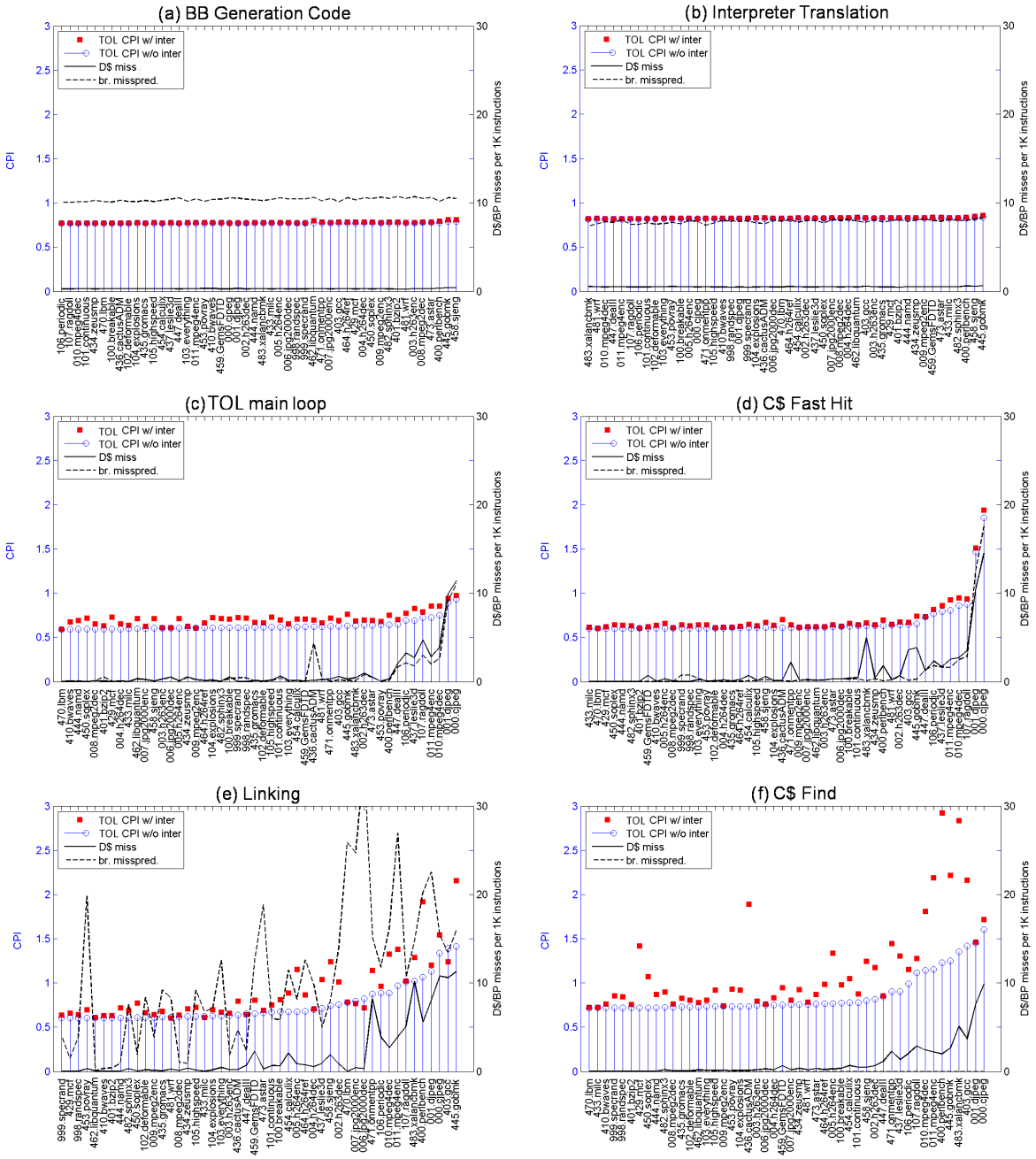

Figure 4.6: Deeper look into the CPI of *Group A*. Applications are sorted from the ones with the lower "without interaction" CPI (left) to the ones with the higher CPI, based on the particular module: (a) *BB Generation Code*, (b) *Interpreter Translation*, (c) *TOL Main Loop*, (d) *C$ Fast Hit*, (e) *Linking* and (f) *C$ Find*.

The first observation is that the *BB Generation Code* and *Interpreter Translation* modules have almost constant CPI, with negligible variance across all applications. Moreover, the interaction with the application code is also almost negligible (squares). This is due to fact that these modules are mainly active at the beginning, which means that at that moment the code cache is mostly empty, so there is a small interaction. Another conclusion that can be drawn is that the *BB Generation Code* and *Interpreter Translation* modules are not affected by the type of instructions of the guest application; although different applications have different instruction mix, the CPI of these two modules is mostly constant.

The second interesting observation is related to the *TOL Main Loop* and the *C$ Fast Hit* modules. The CPI of these two modules has a different profile compared to the previous cases (*BB Generation Code* and *Interpreter Translation*). In particular, these modules have higher dependency on the guest application and at the same time they are affected more by the interaction with the guest application. Towards explaining this behavior, notice that the CPI for these two modules has the highest value for the same applications: 000.cjpeg, 001.djpeg, 010.mpeg4dec, 011.mpef4enc, 437.leslie3d, 106.periodic and 107.radgoll. These applications have small amount of simulated instructions for the specific observed modules and at the same time a high number (compared to the rest of the applications) of D$ misses.

To better understand the influence of the D$, we show Figure 4.7. This figure presents the TOL CPI simulations with perfect D$ for "without interaction" simulation scenario. The CPI of these modules under this perfect D$ scenario shows no or little variance. This leads us to conclude that the reason behind the variance is the D$ misses.

The final observation includes the *Linking* and *C$ find* modules. These modules also present variable CPI. Besides the already mentioned applications (000.cjpeg, 001.djpeg, 010. mpeg4dec, 011.mpef4enc, 437.leslie3d, 106.periodic, 107.radgoll), there are four additional applications which cause these modules to have high CPI: 400.perlbench, 403.gcc, 483.xalancbmk and 445.gobmk. The common factor for these additional applications is that they have a high number of created superblocks (number of static superblocks). The SB creation module is very data intensive; consequently the TOL modules executed after it suffer from poor D$ performance. For DARCO, *Linking* is the module that follows the SB creation in the execution flow and therefore suffers from the aforementioned effect. Consequently, the *Linking* module spends more time waiting for microarchitectural resources. This is reflected by the increased D$ and branch predictor miss rates (depicted in the Figure 4.6). According to our experimental data, as expected, this is a cascaded effect; *i.e.* it affects all the modules executed after the SB creation (the execution path is *Linking* $\rightarrow$ *code cache execution* $\rightarrow$ *C$ Find*). This effect does not affect the *Interpretation* and *BB translation* modules, since after

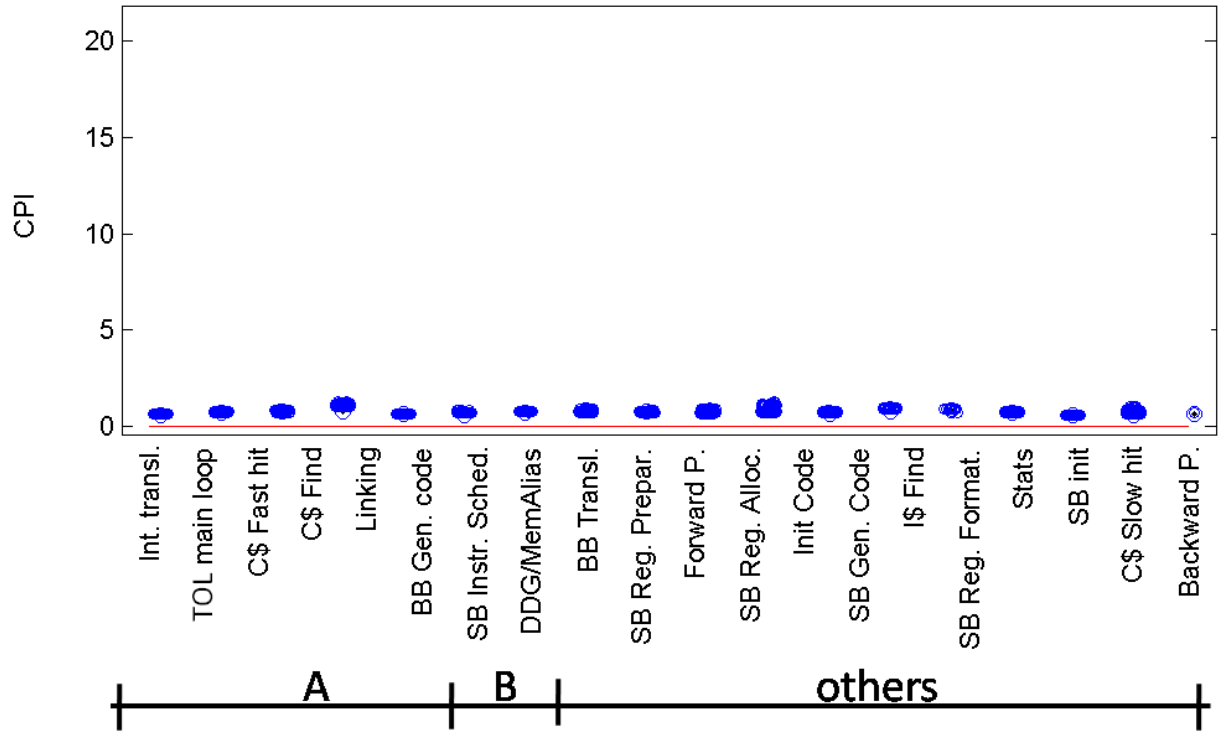

Figure 4.7: TOL CPI per module "without interaction" presented with the boxplot for the perfect caches configuration. Modules are sorted from the ones with the higher contribution (left) to the ones with the lower contribution.

the SB creation the activity of the interpreter and BB translator is relatively small.

Variability of the microarchitectural interaction of the TOL module - α_{module}

In this section we study the interaction effects for the most impacting modules, *i.e.* the ones of *Group A* in Figure 4.6. From Figure 4.6 it is possible to draw two conclusions. The first is that the mean value of the interaction is not constant across the different TOL modules. The modules that suffer mostly from the interaction are the *C$ Find* and the *Linking*. As for the second conclusion it regards the variance of the interaction across the different applications. This variance is not constant which means that the specific module has different interaction for different applications. Moreover, the variance of the *C$ Find* and *Linking* modules is correlated among the applications studied, that is, if for a specific application the variance of the *C$ Find* is high, it is also high for the *Linking* module. Therefore we can conclude that this variance is due to the characteristic of the application and not of the specific module.

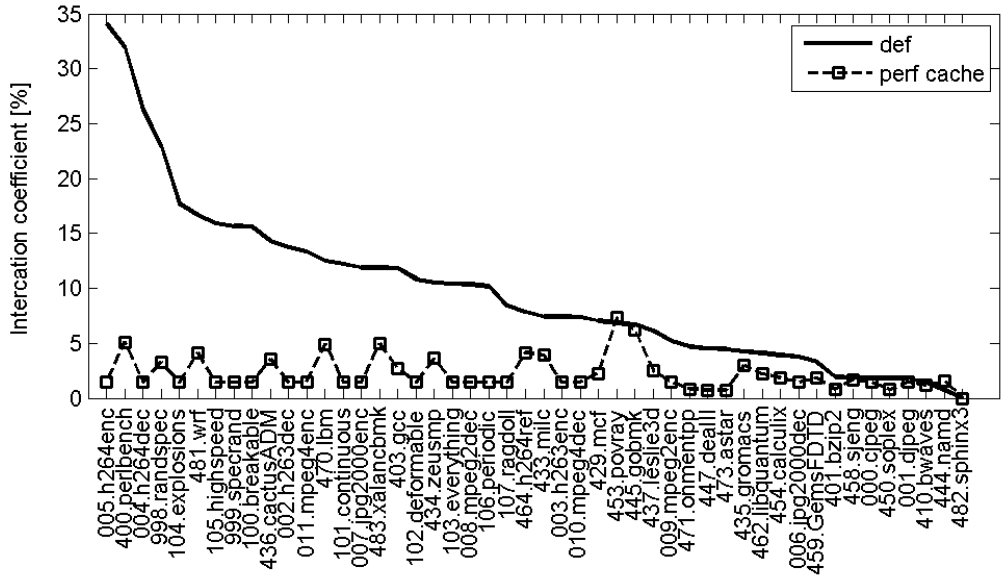

Figure 4.8: Interaction coefficient for default (def) and perfect data cache (perf cache) configurations. Applications are sorted from the ones with the higher interaction coefficients (left) to the ones with the lower interaction coefficients.

The main factor that affects the interaction is the switching frequency between the TOL and the code cache. Lower switching frequency decreases the effect of competition for microarchitectural resources. On the other hand, as this frequency increases the interaction effect becomes more pronounced. The reason behind this is related to the "pollution" in the several microarchitectural structures the workloads (code cache code and TOL) cause to each other.

The structure that is mostly responsible for this effect is the D$. This is verified by the fact that the effect of the interaction was found to be minor when we repeated our experiments with a perfect D$. This is shown in Figure 4.8. With perfect D$, the interaction is around 5%, whereas without perfect D$ it can be above 30%. When switching from code cache to TOL code, the code cache data is evicted from the D$ (especially if the executed TOL module is data intensive). Then, when execution returns to the code cache, the code will suffer from additional D$ misses which would not be the case if the switch did not happen.

4.3.3 The Effect of Different Microarchitectural Configurations

The results presented up to now are based on one microarchitectural configuration, summarized in Table 3.1. In order to verify the generality of the results we repeated the previous studies for four additional microarchitectural configurations. These configurations have different sizes for the D$ or the L2$ and are as follows:

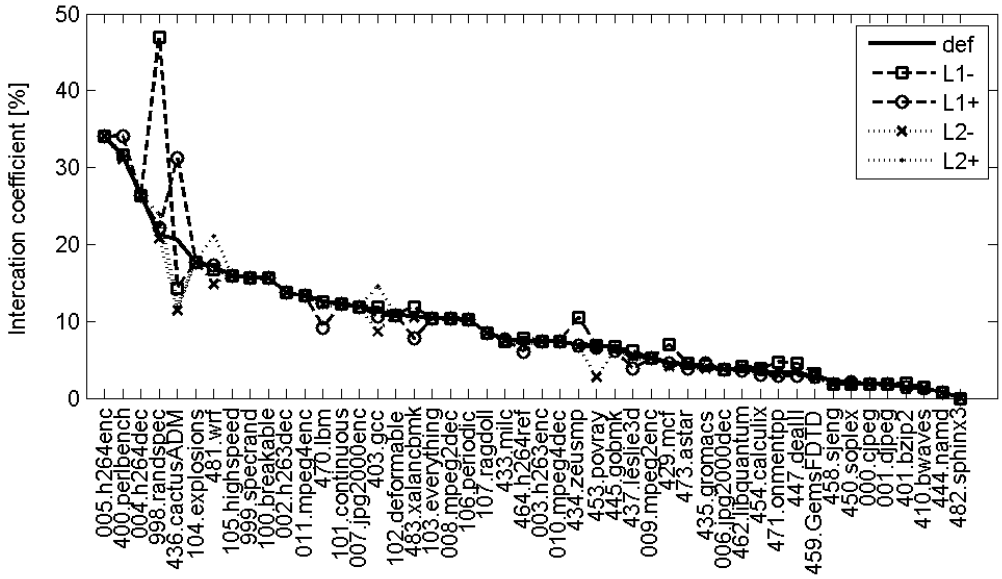

Figure 4.9: Interaction coefficient for different microarchitectural configurations: def (default), L1- (D$L1=16KB), L1+ (D$L1=64KB), L2- (L2$=256KB) and L2+ (L2$=1MB). Applications are sorted from the ones with the higher interaction coefficients (left) to the ones with the lower interaction coefficients.

- D$L1=16KB
- D$L1=64K
- L2$=256KB
- L2$=1MB.

Although the absolute numbers change slightly (which was the expected behavior) the trends and the conclusions remain the same across the different configurations. This means that regardless of the configuration, the TOL microarchitectural behavior is not constant and depends on the emulated application. At the same time, the microarchitectural interaction is significant and non constant. According to our results (Figure 4.9) the interaction is affected by the cache sizes. In particular, we found the average interaction to increase from 10.9% to 11.8% when the D$ size was halfed and to decrease from 10.9% to 10.3% when it was doubled. As for the case of the second level cache, the effect was smaller; for a half size L2 the average interaction increased from 10.9% to 10.92% and when the L2 cache was doubled it decreased from 10.9% to 10.86%.

4.4 TOL Predictability

The goal of this section is to answer the question: “Is it possible to predict the TOL behavior based only on high-level, *guest* statistics?”. If the TOL behavior is indeed predictable then it is possible to perform microarchitectural studies without including it in the simulation period. This would speed up and at the same time simplify the simulation process. Notice that constructing the predictive model for most of the behaviors in computer architecture is quite complex task and probably requires separate thesis. Therefore in this section we examine the predictability based on the statistics that community believes that affect the TOL behavior the most.

As said, the statistics that are used as inputs to the predictive model should be collected at the guest level (high-level statistics) and should not require the host cycle-accurate simulation. The statistics that are used for this work are the statistics that are known from the literature to affect the behavior of the TOL the most. They trigger long operations in terms of host instructions [59] and the ones that are studied are listed as follows:

- number of calls
- number of returns
- number of indirect branches
- number of static basic blocks
- number of dynamic basic blocks
- number of static instructions
- number of dynamic instructions.

The reason behind selecting the number of calls, returns and indirect branches as part of the input set is because each such x86 instruction is translated into a (usually long) sequence of host instructions. The number of static instructions is relevant for the same reason, each static instruction needs to be interpreted, a process that requires a large number of host operations. The same applies for the number of static basic blocks which also need to be translated. As for the number of dynamic instructions and dynamic basic blocks, they are relevant because they correlate to the amount of instructions spent during the optimization process. The set of these statistics is labeled as X.

	Y^{tr}	X^{tr}		
	CPI	calls (c)	Static BB (s)	Dynamic BB (d)
APP1	**1.0**	100	1	1
APP2	**1.1**	80	1	20
APP3	**2.6**	1	20	80
APP4	**0.6**	80	20	20
APP5	**2.0**	20	20	60
APP6	**2.1**	20	20	60
APP7	**2.5**	20	1	80
APP8	**0.7**	80	20	20

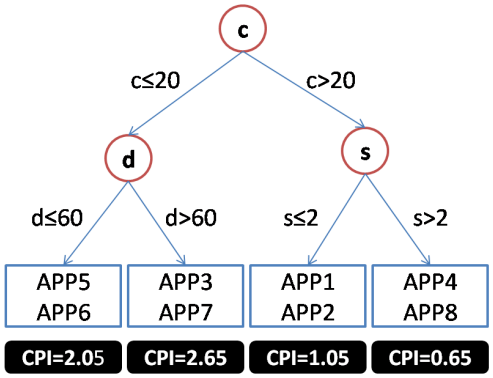

Figure 4.10: Example of input table and regression tree construction. The example is similar to the one presented in [8].

The output of the prediction describes the sources of non-constant behavior of the TOL: (*i*) module contribution (ω_{module}), (*ii*) $hCPI^{w/o}_{module}$ and (*iii*) interaction (α_{module}). The model is per-module, so in total there are 19 such models (one per module) each of which predicts the interaction, contribution and CPI of the particular module. The set of these predicted statistics is labeled Y.

The model is based on Regression Trees [19]. We have tried multiple different models to do the prediction (such as Linear Regression, Interpolation, etc.), but this particular one is the one that delivers the minimum error. Regression Trees is usually used model in computer architecture to examine the predictability of some behavior [8]. It is a mathematical tool which partitions data into sets such that the members of the set have minimum variance between them. More about the regression trees can be found in [19] and [8]. Here we will just explain the basic concept using the example in Figure 4.10 (which is similar to the one found in [8]).

Assume that the model predicts the CPI of an application based on 3 statistics, the number of calls (C), the number of static basic blocks (S) and the number of dynamic basic blocks (D). The model is configured to classify the applications in 4 groups (more groups could be selected). This data is presented by Figure 4.10 and is going to be used to train the model (Y^{tr} = CPI and X^{tr} = C, S, D).

The resulting regression tree, constructed based on the *training data* (X^{tr} and Y^{tr}) is depicted by a binary decision tree in Figure 4.10. As it can been observed, eight applications are classified into four groups with mean values: 2.05, 2.65, 1.05, 0.65.

After training, the regression tree is used to predict the CPI of new applications. Thus, if we have a new application (in this example APP9) with the data $Y = 1.95$, and $X = (c = 15, s = 20, d = 50)$, the model will predict value $Y^* = 2.05$, based on training sets (Y^{tr} and X^{tr}). The value $Y^* = 2.05$ is selected because

the path that is followed in the Regression Tree is: $c \leq 20$ ($c = 15$) and $d \leq 60$ ($d = 50$).

For the Regression Tree, which is developed to predict the behavior of the TOL, we used a big training set. In particular, this set includes multiple samples from each application (each sample has a length of 10M instructions). As for the number of chambers (isolating groups), it was set to 20 as this setup was the one that gave the best results. In total we have 15080 samples.

In order to evaluate the predictability, we perform the cross variations test (similar to [8]). The whole set of samples is divided into 10 randomly chosen subsets, such that each subset contains 10% of the whole population. For each of these sets, for instance set i: $\{X_i$ and $Y_i\}$, the other 9 subsets are used as a *training data* for the Regression Tree (RT_i). Then, the predicted values Y_i^* are compared with the measured Y_i values of the subset.

The measure of predictability is defined as the *Relative Error* of prediction:

$$RelativeError = \frac{\sum_{i=1}^{10} \sum_{k=1}^{K} (y^*(i,k) - y(i,k))^2}{var(Y)} \tag{4.9}$$

In similar studies [8], the authors consider that the dependent parameter is predictable when the relative error is less than 0.15 - here we will follow the same assumption.

Figure 4.11 presents the relative errors for all 3 sources of variance, across all modules: ω_{module}, $CPI_{module}^{w/o}$ and interaction (α_{module}).

The results of *Group A*, which is the one with the most important contribution to the TOL overhead, are split into two categories. The first category includes the modules that are not predicted well by the model, these are the *Interpreter Translation* and the *BB Generation Code*. Notice that this variance is consistent across the different factors, *i.e.* contribution (ω_{module}), $CPI_{module}^{w/o}$ and Interaction (α_{module}).

The other modules of *Group A*, *i.e. TOL Main Loop*, *C$ Fast hit*, *C$ Find* and *Linking*, have better predictability across all factors. This is due to the fact that these modules correlate well with the selected x86 statistics.

As for the other groups, the predictability is low. This however, does not affect significantly the results as their overall contribution is low.

Notice that according to the prediction model (Figure 4.11) the *Interpreter Translation* and the *BB Generation Code* modules have high variance. However, according to the experimental data presented by Figure 4.6 both modules have a constant behavior. This observation clearly shows that this predictive model is not accurate enough and therefore cannot be used as a substitute to the simulation process. Further work is necessary on predicting the behavior of the TOL.

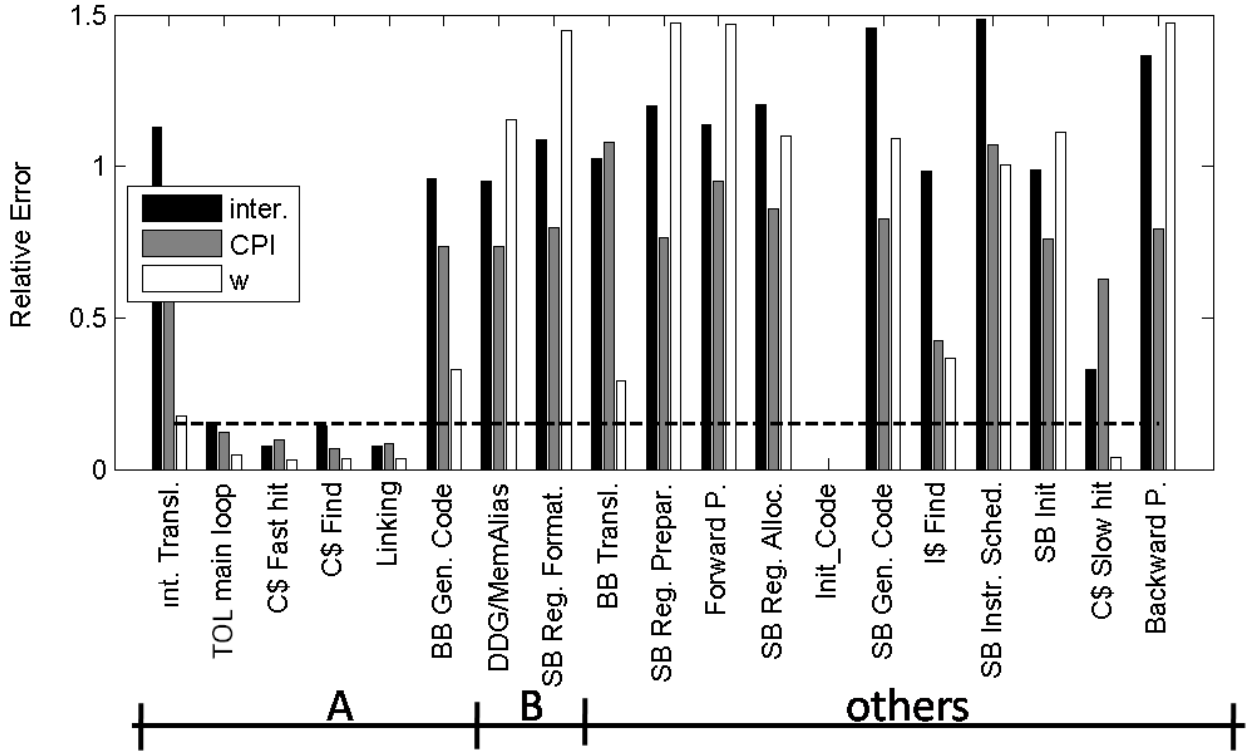

Figure 4.11: Relative errors of predictions based on Regression Trees for following TOL statistics: (*i*) ω_{module}, (*ii*) CPI_{module} and (*iii*) $interaction$. The modules are sorted from left to right according to their average contribution.

4.5 Results

After analyzing and discussing variability and predictability of the TOL, this section shows the simulating errors results for TOL not being included in the simulation. Figure 4.12 presents the simulation errors for gCPI among all utilized applications for the following simulation cases: (a) assumption of the constant TOL behavior (CNT), (b) assumption of the ignored TOL behavior (ZERO) and (c) assumption of the predicted TOL behavior (PRED). These are the approaches that can ease the process of building simulation infrastructures and that up to now has been used in the literature.

As expected, after detailed analysis in the previous chapters, gCPI error is very high in the cases when TOL simulation is skipped. This is especially visible in the cases where TOL overhead is relatively big, such as 000.cjpeg, where error is around 80%. On other side, smaller TOL overhead leads to the smaller simulation error. Having this rule to the extreme, for the cases with negligible TOL overhead simulation error is almost negligible. Unfortunately only few applications have this feature.

Assuming that the TOL has the constant behavior gives slightly better simulation errors than assuming ignored behavior; a few percentage better error across all applications. On the other side, although prediction

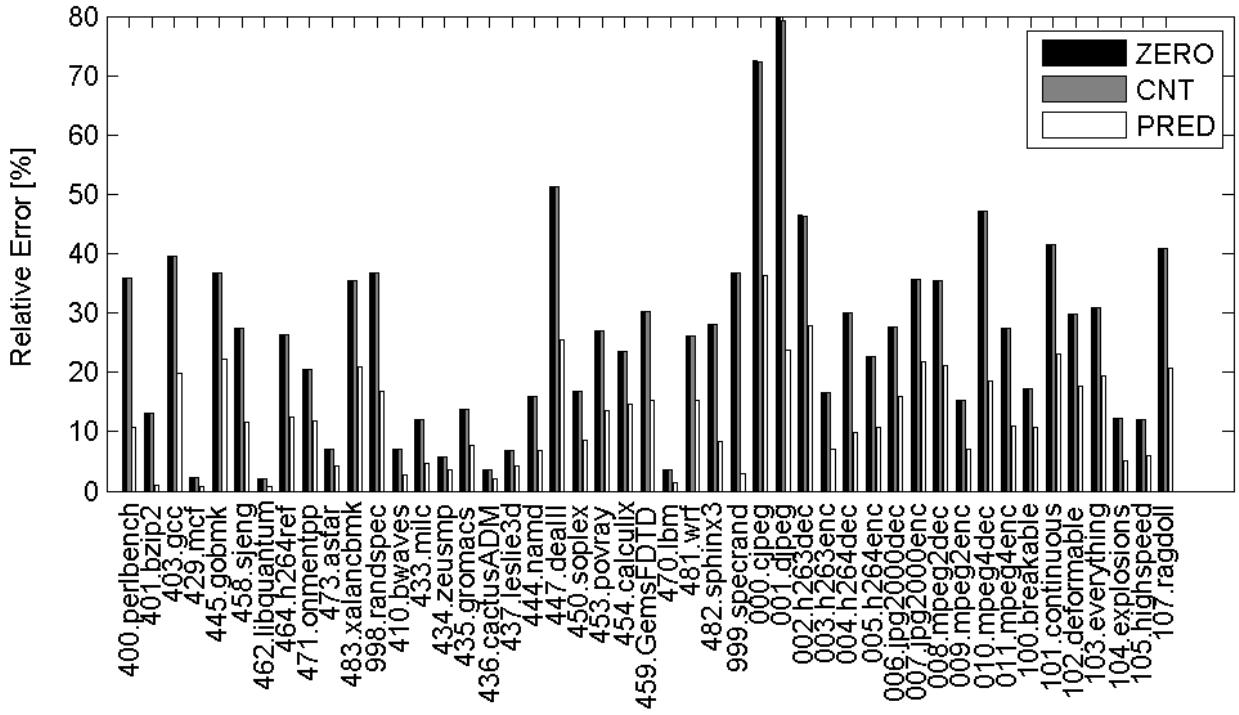

Figure 4.12: Simulation error in the case of: (a) assumption of the constant behavior (CNT), (b) assumption of the ignored behavior (ZERO) and (c) assumption of the predicted behavior (PRED). Applications are sorted according to the benchmark suite: SPEC2006INT, SPEC2006FP, MedaBeanch and PhisicsBench.

based on high-level statistics gives better errors, they are still not accurate enough. In some case, like for 447.dealII, the simulation error when prediction is higher than 25%, due to high number of superblocks being constructed. All these facts lead to the conclusion that entire simulation for HW/SW co-designed processors is absolutely a must and TOL cannot be avoided during the simulation process in any sense.

As discussed in Chapter 3, in this thesis besides gCPI simulation errors we calculate also simulation errors for number of host instructions and percentage of dynamic code being executed in superblock mode (SB coverage). However in the cases when TOL simulation is skipped, the error for SB coverage is 0, because TOL does the formation of the code regions and it optimizations perfectly, whereas the simulation error for number of host instructions is similar to the error of gCPI. Therefore these results are not presented.

4.6 Conclusions

In this chapter we studied the microarchitectural behavior of the TOL. We showed that the behavior of the TOL depends on the guest application being emulated and consequently it cannot be omitted by any microarchitectural study.

The variance of the TOL behavior was studied in detail and identified the three most important reasons.

The first is related to the fact that different applications stress the TOL in a different way leading to the TOL modules having different contributions. The second reason is the fact that the behavior of these modules varies for different applications. Finally, we showed that the interaction between the code of the TOL and of the guest application is significant and decreases the TOL performance noticeably.

In an effort to predict the behavior of the TOL using high-level *guest application* statistics we tried several known predictive models. We found the regression trees to be the approach that delivered the highest accuracy. We used a set of statistics that are known to have a dominant effect on the execution profile of the TOL but the results of the prediction algorithm were not accurate. This leads us to conclude that predicting instead of simulating the TOL behavior is not accurate enough, at least with the known predictive methods. Together with variable behavior of the TOL it proves that the cycle-accurate simulation of entire TOL is absolutely necessary.

Chapter 5

Warm-Up for HW/SW co-designed processors

This chapter proposes a technique that accurately warms-up the state in the case of HW/SW co-designed processors. The chapter is organized as follows. Section 5.2 discusses the related work. Section 5.3 describes the problem of the TOL warm-up, while Section 5.4 provides the survey of the existing warm-up techniques. The proposed warm-up scheme is explained in Section 5.5. Section 5.6 describes the experimental methodology and the results are presented in Section 5.7. Finally, Section 5.8 concludes the chapter.

5.1 Introduction

In the previous chapter we showed that cycle-accurate simulation of entire HW/SW co-designed processor design is a must. As explained in Chapter 2, cycle-accurate simulations are usually 4-5 orders of magnitude slower than native execution. Therefore in order to have faster, but still accurate simulations, current simulation methodologies restore the architectural state from a checkpoint or trace file, warm-up the microarchitectural structures for a few million instructions (caches, branch predictor, etc.) and then obtain cycle-accurate statistics from that point using a cycle-accurate simulator. In this chapter, we show that warm-up methodologies for HW-only processor designs are not suitable for simulation of HW/SW co-designed processors.

Warming-up just the microarchitectural structures is not sufficient for HW/SW co-designed processors as this would not guarantee a warmed-up TOL state. Notice that warming-up the TOL is extremely important for bounding the error because an inaccuracy in the TOL state has significantly higher simulation time than an inaccuracy in the microarchitectural state. According to our experiments, an inaccuracy in

TOL (*e.g.* a code region is not optimized) can be 3-4 orders of magnitude more expensive compared to a microarchitectural inaccuracy (*e.g.* a cache miss).

In this chapter, we propose a novel simulation methodology that targets HW/SW co-designed processors. Our proposed technique is based on choosing appropriate promotion thresholds (TH_{BB} and TH_{SB}) and warm-up lengths based on the guest ISA characteristics of each application sample. It achieves a very good trade-off between simulation time and accuracy: an average error of 0.75% with a reduction in the simulation time by 65X). As opposed to other alternative techniques, the proposed technique satisfies the two following requirements: (*i*) it has a good trade-off between accuracy and simulation time (accuracy concept) and (*ii*) it can be used to evaluate the entire HW/SW design *i.e.* the microarchitecture, the TOL, the host ISA and the interactions among these components (*general applicability concept*).

The contributions of this chapter are:

1. We show that the traditional simulation techniques used for HW-only processors have a prohibitive simulation time in order to deliver acceptable accuracy when applied to HW/SW co-designed processors.

2. We show that the quality of the warm-up cannot be defined solely by the CPI error of the simulation but that the error should be studied as a vector of the errors of many other TOL metrics, such as the number of host instructions and of the dynamic coverage of the different optimization levels. We show that using just the CPI as the error metric is insufficient and can mislead the research in HW/SW co-designs.

3. We propose a TOL warm-up technique that chooses the appropriate promotion thresholds and warm-up length based on guest ISA characteristics for every particular application sample. The proposed technique, as opposed to alternative strategies, satisfies the two aforementioned requirements: accuracy and general applicability. In fact, we show that the technique achieves a low error at a low simulation time (average error of 0.75% with an average simulation time reduction of 65X) and that it is tolerant to different TOL and microarchitectural configurations.

5.2 Related Work

The challenge of warming-up the microarchitectural state before simulation has been the focus of many research papers. Falcon et al. [28] proposed efficient solution for choosing an appropriate warm-up period concluding that the warm-up has to start from the most recent (to the simulated interval) application

phase change. Since the data cache is typically the structure that affects performance the most, several researchers focused on choosing an appropriate warm-up period for it [34, 35, 27]. By analyzing the reuse latency between memory accesses to the same address, these techniques identify the appropriate warm-up period leading to average error lower than 1%. Other research projects attack the problem of warming-up the branch predictor [43].

The works presented above target the warm-up of the microarchitectural structures which covers just one of the challenges for HW/SW co-designed processors evaluation. In this chapter, we concentrate on the warm-up of the TOL state which, according to our results, has orders of magnitude more impact for these architectures.

The work closest to ours is astroLIT [56], in which the authors propose a methodology to reuse existing checkpoints / traces for HW-only designs to evaluate HW/SW co-designed processors. These traces contain only a few million instructions of the collect period in HW-only processors. AstroLIT iterates over the same trace several times until a steady state is reached and the CMS (the equivalent to the TOL) is properly warmed-up. Although their technique makes a lot of sense in their environment and under their constraints, it is not generally applicable to HW/SW co-design evaluation, because the technique uses perfect profiling information (in other words, profiling information from the future since the dynamic stream of the warm-up is the same as the dynamic stream of the collect period).

Accurate TOL warm-up shares some challenges with the evaluation of JIT (Just In Time) compilers; the main issues however are different. In particular, the main challenge for the evaluation of JIT compilers is the non-deterministic behavior of the system because it is composed of many cooperative threads (application, JIT compiler and garbage collector threads) [29, 31, 63, 39]. In order to avoid this non-deterministic behavior, researchers usually dump and restore the *compilation plan*, which consists of the profiling information (such as edge profilers) and the compilation decisions. We compared our proposed technique to this solution and we show that although it achieves good accuracy at a reasonable simulation time, it does not meet the general applicability requirements. Furthermore, the focus of the proposed simulation technique is on improving the simulation accuracy at a low simulation time rather than the non-determinism as the HW/SW co-designed processors do not suffer from this issue.

Choosing the most relevant samples for a given application is beyond the scope of this chapter and it is studied in next chapter.

5.3 TOL Warm-Up

The target of the warm-up for a given sample is to enable performance simulations where this sample behaves as close as possible to how it would behave for the *authoritative execution* with the minimum possible simulation time. We refer to the authoritative execution as a simulation of the particular trace or sample but starting the simulation from the first instruction of the application. Whereas for HW-only architectures the warm-up should provide an "accurate as possible" *microarchitectural* state (*e.g.* cache, branch predictor, etc.), for HW/SW co-designed processors it should provide accurate TOL state as well. As for the most impacting components of the TOL state, these are the *code cache* and the *profiling information*.

Figure 5.1 summarizes the simulation process of a single application sample for HW-only and for HW/SW co-designed processors with the X-axis representing time. In HW-only processor designs (Figure 5.1-a), once the architectural state has been restored, the HW structures are warmed-up for a few million instructions (Y_{hw}) before simulation statistics are collected (Z). As for the case of HW/SW co-designed processors (Figure 5.1-b), the HW warm-up needs to be preceded by a TOL warm-up period (Y_{TOL}). The typical *error/ simulation time* trade-off applies for the TOL warm-up as well. The longer this warm-up period is, the closer the state will be to that of the authoritative execution.

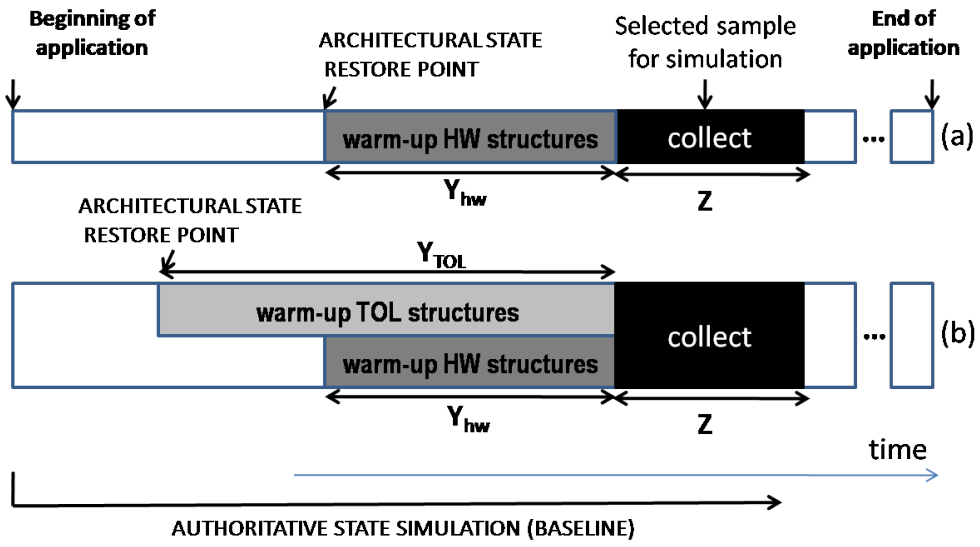

Figure 5.1: (a) Simulation process for HW-only processors. For accurate simulation the HW warm-up is enough. (b) Simulation process for HW/SW co-designed processors. For accurate simulation it is necessary to warm-up both the HW structures and the TOL.

Notice that the TOL warm-up period does not include cycle-accurate simulation. This is not done for accuracy (actually it affects negatively the total error) but for reducing the simulation time as the TOL

warm-up period is usually very long. As the HW structures get warmed-up much faster than the TOL, the cycle-accurate simulation is only included at the last few million instructions.

As explained earlier, inaccuracies in TOL warm-up have a cost that is orders of magnitude higher compared to the inaccuracies caused by the microarchitectural warm-up (*e.g.* code region optimization is in the order of tens of thousands of cycles whereas a costly microarchitectural event such as an L2 miss in the order of hundreds). The crucial requirement for accurate HW/SW co-designed simulation of an arbitrary application sample is for the guest code segments to be in the same optimization state they would have been for the authoritative execution. By "*same optimization state*" we refer not only to the optimization level at which the guest code is optimized, but also to characteristics such as the region formation (*e.g.* whether a superblock from an *if-then-else* statement includes the *if-then* or the *if-else* paths), the set of applied optimizations (*e.g.* whether dead code eliminations is applied or not, the unrolling factor for loops), among others.

In order to clarify the promotion process, we use the example presented by Figure 5.2 which assumes promotion thresholds of 3 and 7 respectively. This means that a code region is promoted to the first optimization level after it is executed 3 times and to the second level after 7 executions. During the authoritative execution (Figure 5.2-a), this code region is already at the highest level when the collect period starts, as it was executed 8 times before the collect period started. However, during the TOL warm-up period (Figure 5.2-b) this code region did not execute enough times and it is still at level 1 when the collect period starts.

This scenario suffers from two sources of inaccuracy. The first is that during the first two executions (execution 6 and 7), this basic block will be executed at a different optimization level compared to the authoritative case (level 1 instead of level 2). The second source of inaccuracy, which is more important, is that the optimizer will kick-in during the collect period to optimize this basic block which has just reached the promotion threshold (7 executions). The host instructions that are due to the execution of the optimizer are a clear overhead (compared to the authoritative execution) and do not contribute in any way to the application forward progress. Given that the cost of the optimizer is high [25], the skew in the results will be significant.

This example clearly shows that it is crucial for a warm-up technique to guarantee that the TOL state before the collect period is such that as many basic blocks as possible are at the optimization level they would have been for the authoritative execution.

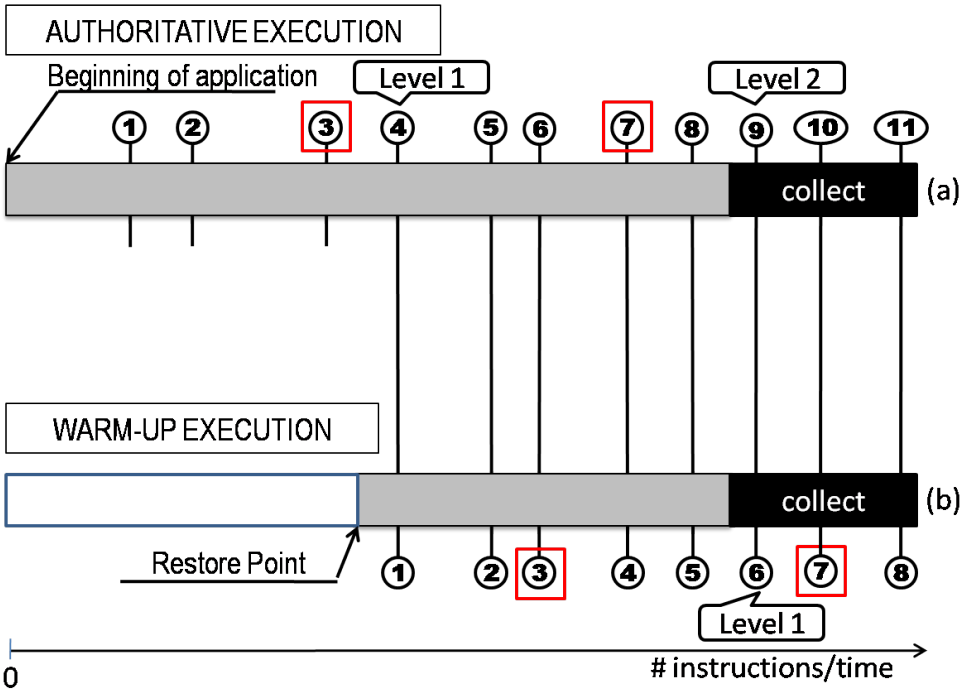

Figure 5.2: Code region optimization during warm-up. The numbers in circles present the number of executions of a specific basic block: (a) Authoritative Execution, (b) Execution after warm-up.

5.4 Existing Warm-Up Alternatives

This section presents existing simulation techniques adapted to the needs of warming-up for HW/SW co-designs.

5.4.1 Authoritative State Simulation (AS)

This approach implies having the TOL and the cycle-accurate simulator active from the beginning of the application and collecting statistics just for the selected sample. Although it provides the authoritative results, *i.e.* it incurs zero error, it is the most time consuming technique and it is prohibitively expensive when the selected sample is far away from the beginning of the application. The error of the techniques, described next and in the Section 5.5, uses this scheme as the ***baseline*** (Figure 5.3-a).

5.4.2 Fast Authoritative State Simulation (FASS)

This configuration is similar to the previous one as the TOL is active from the beginning of the application. However, in this case, the cycle-accurate simulator is only activated close to the collect period in order to decrease simulation time. The error of FASS compared to AS can be attributed to the non-perfect

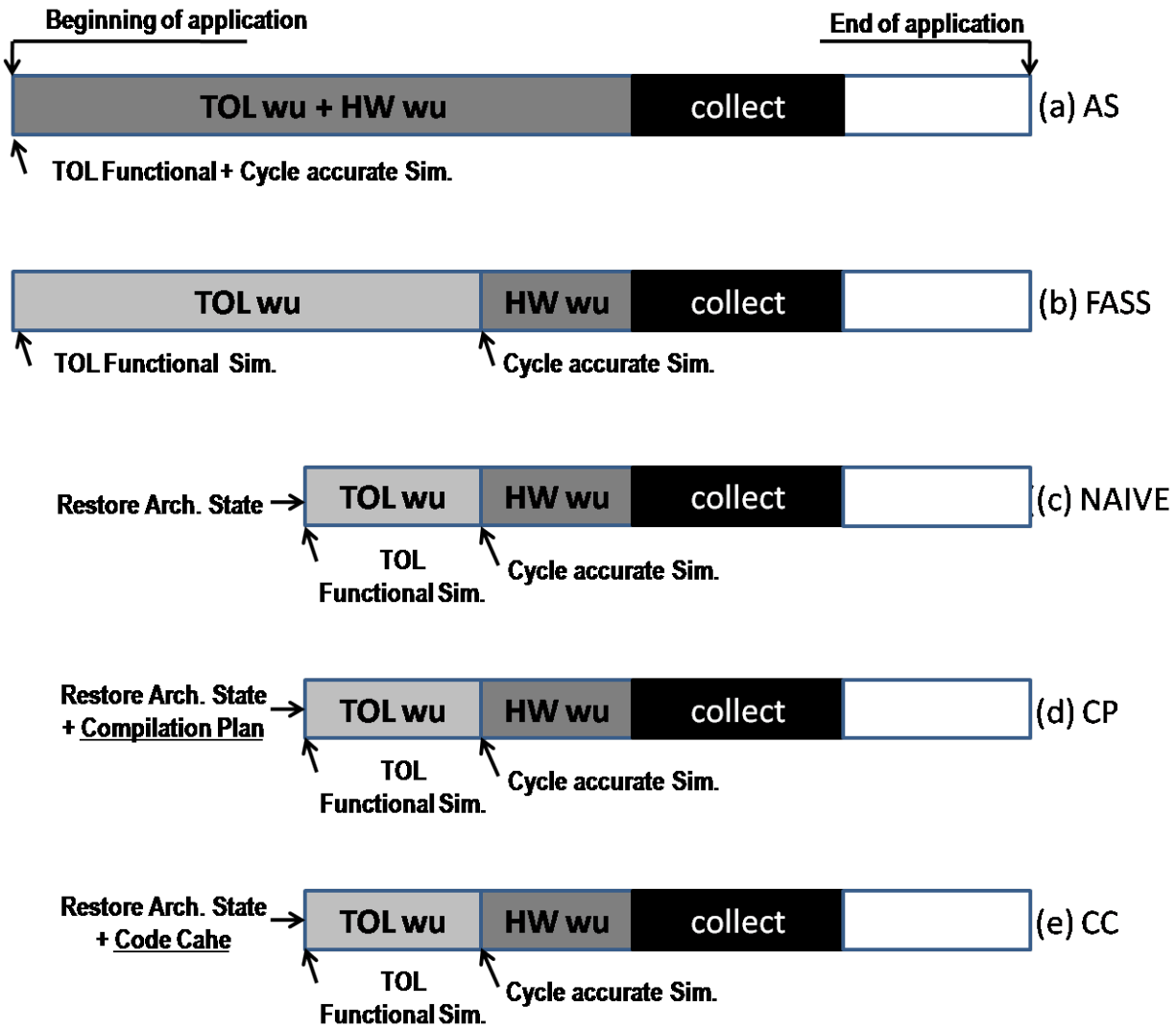

Figure 5.3: Warm-up simulation techniques: (a) Authoritative state simulation (AS), (b) Fast authoritative state simulation (FASS), (c) Naive TOL warm-up simulation (NAIVE), (d) Compilation Plain TOL warm-up simulation (CP) and (e) Restoring the code cache state warm-up simulation (CC).

HW warm-up (Figure 5.3-b). This technique is still very expensive since it requires running the TOL from the beginning of the application. This becomes a problem for studies that involve changing the TOL since all the application samples would need to be regenerated. Notice that both the AS and the FASS have 100% accurate TOL state. The FASS however suffers from inaccuracies due to the HW warm-up.

5.4.3 Naive TOL Warm-Up Simulation (NAIVE)

This configuration implies restoring the architectural state from a checkpoint. Then the TOL is warmed-up for a period of time followed by a HW warm-up period where the cycle-accurate simulator is active. The NAIVE technique suffers from errors introduced by both the TOL and the HW warm-up

inaccuracies (Figure 5.3-c). NAIVE is not a good alternative if one wants to bound the error, since it suffers from "different optimization level" issues (explained in Figure 5.2). During our evaluation, we found this problem to be very significant. For example, 437.lesie3d has an error of 400% for TOL warm-up of 1M instructions!

5.4.4 Compilation Plan TOL Warm-Up Simulation (CP)

This configuration is similar to the previous configuration where the simulation starts from the checkpoint. In this case however (Figure 5.3-d), the TOL compilation plan is restored from a previous run and the contents of the code cache and the profilers are created off-line before simulation. This saves significant simulation time as the large number of iterations needed to identify the hot code are not necessary. Similar to the NAIVE, the CP technique suffers from HW and TOL warm-up errors.

Although, as shown later, CP provides a pretty good trade-off between accuracy and simulation time, it does not always meet the general applicability requirement for HW/SW co-designed processors. As explained earlier, the general applicability requirement state that the warm-up technique has to be applicable for every configuration of HW/SW co-designed processors (for every microarchitectural configuration, for every TOL configuration). The cases for which CP does not meet the general applicability requirement are the ones for which the dynamic profile information needed for the region formation and region translation is not accurate. This happens when the high-level profile statistics, such as the edge profilers, are not enough to provide the complete behavior of the execution and need to be augmented with microarchitectural statistics. A few relevant examples to show that CP is not general enough are:

Case 1 - *Software prefetching*: The TOL observes the execution of the application and identifies the delinquent loads which it targets through adding prefetch instructions in the code. As the miss behavior of each load depends on the specifics of the cache hierarchy and the HW prefetcher, the profiling information is not enough unless it is augmented with HW statistics. This of course means that the statistics need to be recollected for each explored HW configuration.

Case 2 - *Control speculation*: The TOL might decide to convert biased branches into asserts based on branch profiling information up to the point where the optimization is performed [55]. Such profiling information depends on how the TOL has built the other code regions and how they behave at run-time. Hence this dynamic profiling information cannot be summarized accurately by a compilation plan from a previous run using a particular TOL configuration.

Case 3 - *Software controlled power gating*: HW/SW co-designed processor has the ability to dynami-

cally inform the TOL about what HW structures are candidates for power gating for specific code sequences, allowing the TOL to react by generating more energy-friendly code regions. This information also depends on the dynamic behavior of the processor and its configuration and cannot be summarized correctly in a single compilation plan.

Notice that these cases apply when the TOL is designed with strong synergy with the hardware, which is not the case for current JIT compilers. However, since CP is a commonly used technique in such experiments, we evaluate it (Section 5.7). To conclude, CP is a good alternative for several cases, but it limits the scope of research that can be conducted.

5.4.5 Restoring the Code Cache State Warm-Up Simulation (CC)

One could take the compilation plan strategy to the extreme: restoring the entire raw contents of the code cache and the profiling information from a previous run. However, we fully discard this option as it has similar simulation time to the CP but is even more limiting. For instance, researchers would need to regenerate the code cache contents whenever they implement a new optimization or change the parameters of an existing one.

Table 5.1: Summary of Existing Techniques.

Technique	G. Applicability	Simulation Time	Accuracy
AS	ok	huge	perfect
FASS	ok	huge	good
NAIVE	ok	small	bad
CC/CP	not always	small	good

Table 5.1 summarizes the aforementioned warm-up techniques showing their characteristics from the point of the feasibility, simulation time and accuracy. It shows the need of finding a new warm-up technique which meets both requirements: *accuracy* and *general applicability*.

5.5 Downscaling Promotion Thresholds (TH)

This section explains the proposed technique which meets both requirements: *accuracy* and *general applicability*.

5.5.1 Methodology Scheme

The technique we propose is based on downscaling the promotion thresholds during the TOL warm-up phase with respect to the original thresholds which allows code blocks to be promoted faster. However,

similarly to the aforementioned NAIVE and CP techniques, this technique also suffers from HW and TOL warm-up inaccuracies.

The proposed technique is depicted in Figure 5.4. After restoring the architectural state, the code cache and the TOL profiling information are empty. To allow code blocks to get promoted faster, the promotion thresholds are scaled down before TOL warm-up. To maintain the execution characteristics during the collect period, the original values of the thresholds are restored after TOL warm-up completes.

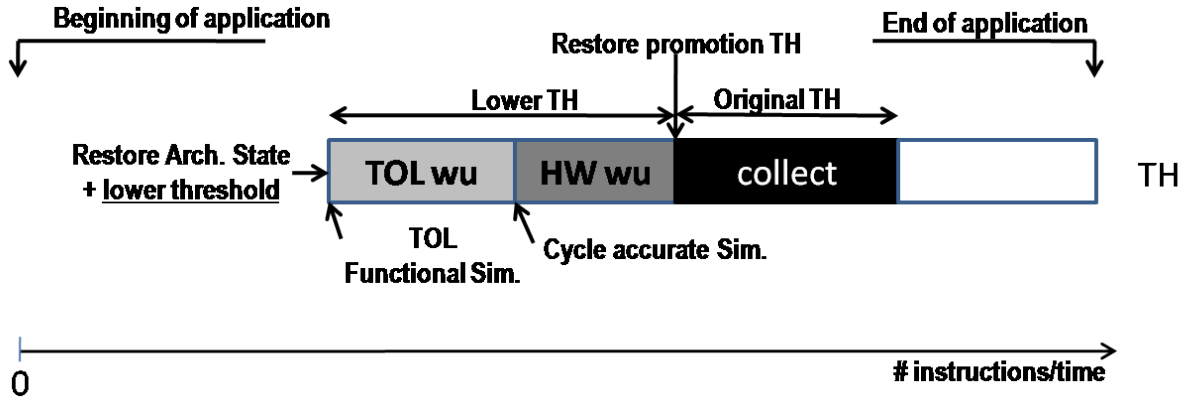

Figure 5.4: Downscaling Promotion Thresholds during TOL warm-up (TH).

The warm-up process for TH is defined by the pair $\{Th, WU_{length}\}$. Th describes the value of the downscaled promotion threshold which is used to promote code to higher optimization stages. WU_{length} describes the amount of guest instructions used for TOL warm-up. Whereas the warm-up *simulation time* solely depends on the warm-up length, the accuracy of the simulation depends on the $\{Th, WU_{length}\}$ pair.

The technique uses a cost function to balance the simulation time with the accuracy. In particular, a very long warm-up period will usually lead to more accurate warm-up, but it is not necessarily the best trade-off between simulation time and accuracy. Having two degrees of freedom (threshold and warm-up) instead of just one (either the threshold or the warm-up), allows the technique to better filter which code blocks are promoted. This is better shown with the example of Figure 5.5. In authoritative execution, code

AUTHORITHATIVE EXECUTION			WARM-UP PERIOD 1			WARM-UP PERIOD 2		
A	B	C	A	B	C	A	B	C
1100	50	200	10	8	7	15	9	11

Figure 5.5: Example of the interaction between the warm-up threshold and the warm-up length in Down-scaling Promotion Thresholds. Values describe number of execution of code blocks A, B and C before the collect period.

blocks A, B and C are executed 1100, 50 and 200 times respectively before the collect period. Let us assume for simplicity that there is only one optimization level and that the promotion threshold is 100. In this case only code blocks A and C will get optimized in the authoritative execution. During Warm-up Period 1 of a given length, code blocks A, B and C are executed 10, 8 and 7 times respectively. In this particular example, there is no scaled promotion threshold that would properly select the code blocks to be optimized. For example, if the threshold were scaled down to 7, blocks A and B would be promoted, whereas if it were scaled to 6, blocks A, B and C would be promoted. For a longer warm-up period (Warm-up Period 2), such a threshold exists. If the threshold were scaled down to 10, then code block B would correctly be filtered out and not promoted during warm-up.

Notice that *Downscaling Promotion Thresholds* is an approximation to the authoritative execution for the TOL warm-up. For instance, downscaling the thresholds drastically reduces simulation time but it may influence region formation and optimization. As an extreme case, imagine a configuration in which the real threshold value of 100 is reduced to 5 to stress promotion. In this case, superblocks may be created differently given an *if-then-else* statement, since the profile information for 5 executions of the basic blocks can be totally different from the information once they have been executed 100 times. In addition, it could be the case that converting biased branches to asserts requires profiling information with accuracy of a 1%-2%, something possible with a threshold of 100 but not with a threshold of 5, leading to completely different optimization regions in both cases.

5.5.2 Predictive Model

As we show in Section 5.7, an *oracle selection* of the best $\{Th, WU_{length}\}$ pair gives good *error / simulation time* trade-off. However, identifying the oracle version is not practical as it would require a large number of simulations. In practice, it is necessary to *predict* the most appropriate threshold and warm-up length pair $\{Th, WU_{length}\}$ for each application sample based on high-level guest application statistics. We propose a predictive model which for each sample *chooses* the pair $\{Th, WU_{length}\}$ which minimizes the error. The predictive model is solely based on the guest PCs execution counters.

Referring to Figure 5.6-a, the optimal pair should be such that the scaled warm-up execution (warm-up period WU1 and WU2) has similar behavior to the complete execution. Thus, the predictive model tries to identify the warm-up period that gives execution distribution (*i.e.* number of times each PC is executed) which, when scaled, minimizes the differences with respect to the distribution of AUTHORITATIVE execution. This can be seen in Figure 5.6-b. The X-axis shows different PCs and the Y-axis their execution

counts. The chart shows three different scenarios, the AUTHORITATIVE execution and the execution distributions of two different warm-up periods, WU1 and WU2. In this case, the predictive model will select WU1 because when scaled properly, it will better match the distribution of AUTHORITATIVE execution.

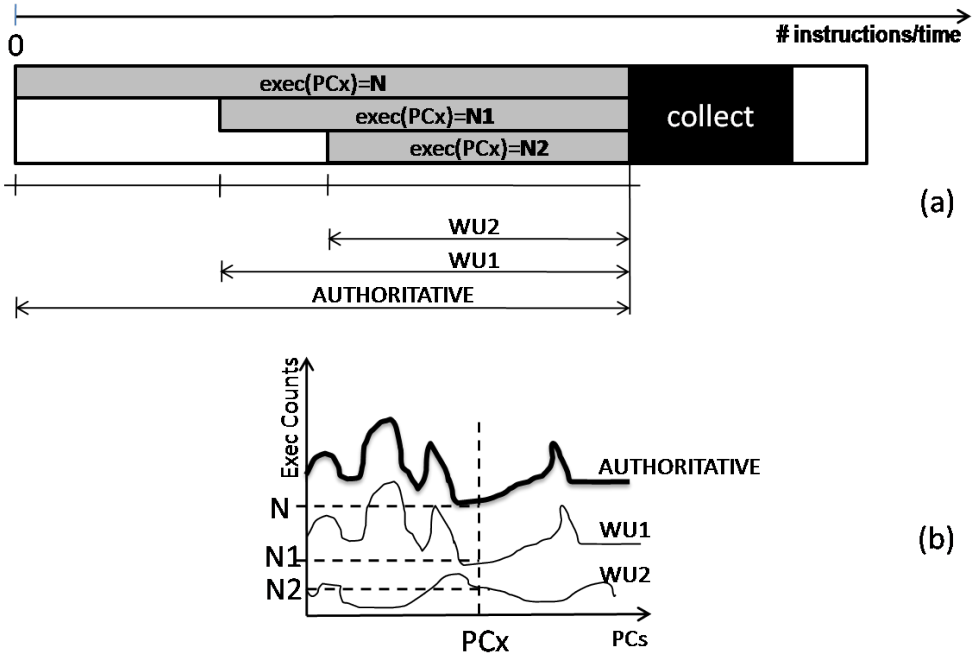

Figure 5.6: Distribution of PC execution counts for different scenarios.

The predictive model searches for the pair $\{Th, WU_{length}\}$ that minimizes the mean square error function defined as follows:

$$F(Th,wu) = \frac{1}{M}\sum_{1}^{M}\left(\frac{\frac{TH_{original}}{Th}N_{wu}^{PCx} - N_{auth.}^{PCx}}{N_{auth.}^{PCx}}\right)^2 \tag{5.1}$$

where M is the amount of the static PCs in the collect period, Th is the investigated downscaled threshold, $Th_{original}$ is the original threshold, N_{wu}^{PCx} is the execution counter of *guest* PC_x for the investigated warm-up period and N_{full}^{PCx} is the AUTHORITATIVE execution counter. Such statistics can be obtained with off-line tools[1] such as PIN [3]. Referring to Figure 5.6-b, when scaled WU1 gives lower mean square error than the scaled version WU2 and therefore is the one selected. Notice that all these are *guest* application statistics, which are TOL independent.

We bounded the simulation time such that $WU_{length} \leq 1B$. Since the main goal studied in this thesis is to reduce the simulation time, having warm-up periods longer than 1B, will still keep simulation time very long. In addition, as it is going to be shown later in this section, our experiments show that such a condition is big enough to catch reasonable small errors.

One might notice that we need an authoritative execution with the instrumentation tool (PIN) for the

[1] By *off-line* we mean tools that do not require the TOL to be active and simply execute the guest application.

predictive model. However, this is not a problem since authoritative execution is necessary in order to create the architectural guest state of the sample either way. Moreover, notice that collecting these statistics is a one-time process.

5.6 Experimental Setup

For the experiments in this chapter, we used the SPEC2006 benchmark suite only. The reason behind is the fact that MediaBench and PhisicsBench are very small benchmarks. This means that authoritative simulation in these cases can be performed in reasonable amount of time and therefore the warm-up for these applications is not credible simulation scenario.

The experimental setup used to evaluate all warm-up techniques is presented in Figure 5.7. Notice that all intervals (*i.e.* Y_{TOL}, Y_{hw} and Z) are measured in number of *guest* instructions (in our case, x86 instructions), as this number of guest instructions is independent of the TOL implementation.

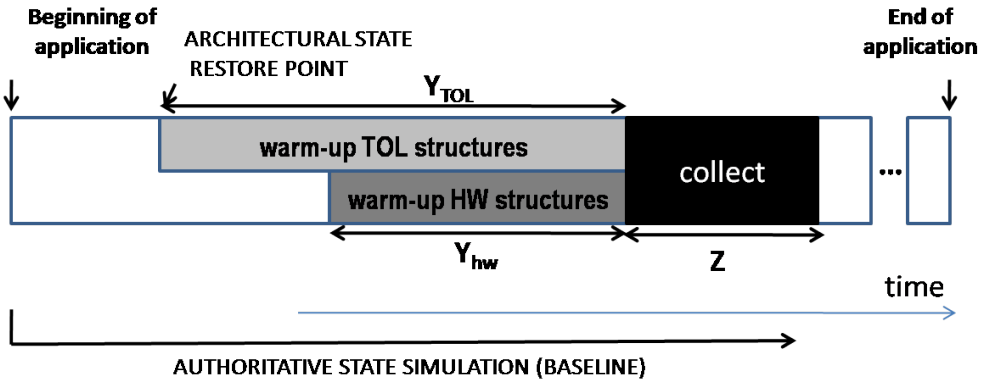

Figure 5.7: Experimental setup for warm-up simulation technique.

In all experiments, we set the *collect* period length as:

$$Z \in \{100K, 1M, 10M, 20M\}. \tag{5.2}$$

For NAIVE, CP and TH, the lengths for the *warm-up* periods are set as:

$$Y_{TOL} \in \{1M, 10M, 100M, 500M, 1B\} \tag{5.3}$$

and

$$Y_{hw} \in \{1M\}. \tag{5.4}$$

The values Y_{hw} and Z are consistent with previous work for HW-only processors [75]. Given the simplicity of the CPU core (relatively small cache - 32KB), we found that a 1M for Y_{hw} was enough [73].

For FASS, the TOL *warm-up* overlaps with the hardware warm-up:

$$Y_{hw} \in \{1M, 10M, 100M, 500M, 1B\}. \tag{5.5}$$

This is done in order to enable comparison between HW-only warm-up and TOL warm-up.

We use 5 samples for every benchmark, where the collect period starts at 1B, 2B, 3B, 4B and 5B x86 instructions from the beginning of the benchmark. In order to guarantee that the comparison is done across the same guest instructions, the collect period starts from a fixed point for all techniques and for all different parameters. The techniques proposed in this work however are applicable to any sample. Therefore, the way by which representative samples are selected is beyond the scope of this chapter and it will be studies in Chapter 6.

In this chapter we define and present the error of each warm-up technique as the maximum error of three metrics: the gCPI (gCPI refers to the guest CPI), the number of host instructions and the SB coverage, *i.e.* the percentage of guest code executed at the highest optimization level, compared to the Authoritative Simulation (AS):

$$error^{tecnhique} = max(error^{tecnhique}_{gCPI}, error^{tecnhique}_{\#instr}, error^{tecnhique}_{SB\%}). \tag{5.6}$$

Definitions of $error^{tecnhique}_{gCPI}$, $error^{tecnhique}_{\#instr}$ and $error^{tecnhique}_{SB\%}$ are given in Section 3.4. The reason is the fact that the accuracy of the gCPI is not enough to express the accuracy of the simulation technique.

In HW/SW co-designed processors, it is possible for a warm-up technique to achieve a gCPI very similar to the authoritative execution, but for the behavior of the TOL to be very distinct between the two. For example, imagine a scenario in which block A is promoted to the highest optimization level *during* the collect period for the authoritative execution. This is depicted in Figure 5.8-a. In this case, the TOL statistics for the collect period would show an important amount of cycles executing the optimizer. On the contrary, imagine that using a warm-up technique, code block A is not promoted to the highest optimization level, as depicted in 5.8-b. In both configurations, gCPI could be similar if the amount of cycles spent in the optimizer in the authoritative execution is compensated by the amount of cycles executing A in a less optimized version in the other simulation. In this scenario, we could have two executions with similar gCPI metrics, but very different TOL metrics. Such a difference would require designers to spend more time improving the optimizations when the real benefit would come from concentrating on the TOL overheads.

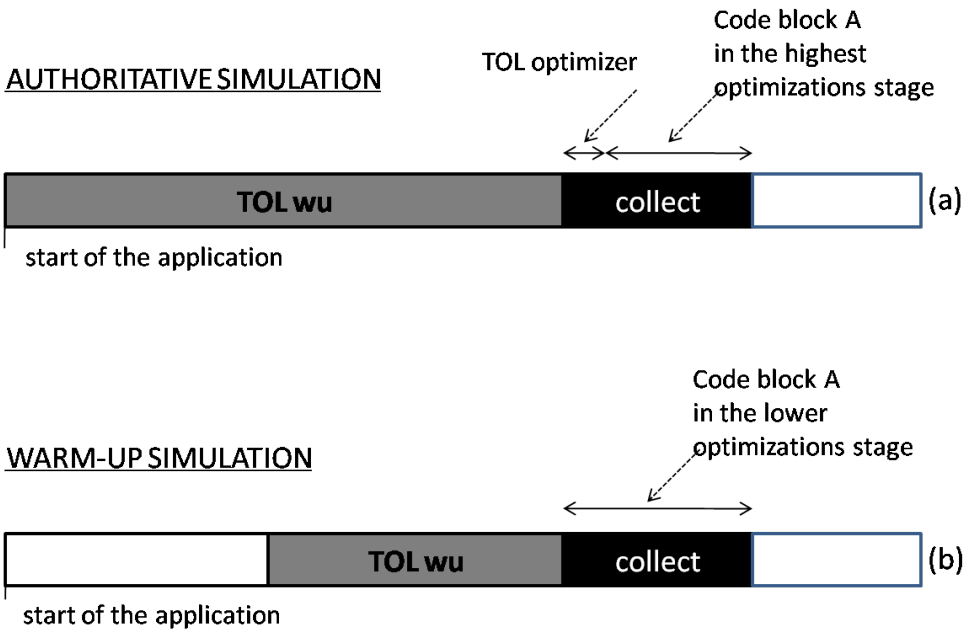

Figure 5.8: (a) Authoritative simulation; (b)Warm-up simulation.

Figure 5.9 explores how common this scenario is. The chart plots the error for two metrics, the superblock coverage and the number of host instructions for the experiments for which the gCPI error is less than 2.5%. As our experimental data show, there are numerous examples for which bounding the gCPI error does not guarantee an equally small error for the SB coverage or for the host instruction count. In particular, 15 of 116 samples have an error for the other statistics that is over 2.5%.

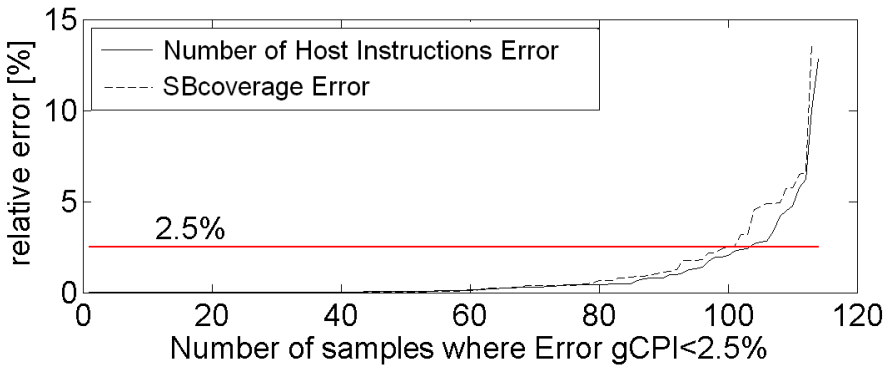

Figure 5.9: Cases when the error gCPI error is limited to 2.5% the number host instructions and SB coverage might be higher.

The simulation time of each technique is measured in terms of the number of *host* instructions. By the simulation time we refer to the total simulation time, including the warm-up period and the collect period. Since our baseline is AS, we report simulation time reduction defined as:

$$timeReduction^{technique} = \frac{timeSimulation^{AS}}{timeSimulation^{technique}}. \tag{5.7}$$

As explained earlier, the simulation time is not measured in time, but in the number of host instructions that are simulated with cycle-accurate host simulator. The TOL functional simulation is not counted as the cycle-accurate simulation is more expansive than the TOL functional simulation. This is done in favor of the baseline.

The further the sample is from the beginning of the application the greater the simulation time reduction is. Since our samples are close to the beginning of the application (maximum 5B instructions far away), the simulation time reduction results shown in this chapter should be regarded as a lower bound.

The *average error* and the *average simulation time reduction* are calculated across *all* samples using geometric mean.

$$errorAVG = \sqrt[number\_Of\_Samples]{\prod_{allsamples} error^{technique}} \tag{5.8}$$

$$timeReductionAVG = \sqrt[number\_Of\_Samples]{\prod_{allsamples} timeReduction^{technique}} \tag{5.9}$$

5.7 Results

This section presents the experimental results and compares the accuracy and the simulation time of different techniques. The purpose of this section is to back up the claim that TH is not only the solution which satisfies both requirements, but it also gives better results than CP.

5.7.1 FASS and NAIVE

Figure 5.10 presents the average error in logarithmic scale for the Fast authoritative state simulation (FASS) and the NAIVE TOL warm-up configurations when compared to the Authoritative State (AS) simulation. The X-axis represents the number of x86 instructions used for warm-up. Each subfigure presents the results for a different collect period Z. For clarity, results are presented using boxplots for the NAIVE technique and just with a trend line for the FASS.

The error of FASS is minimal; actually the maximum error never exceeds 2%. The trend of the error is to decrease as the microarchitectural state gets closer to the state of the authoritative execution. Expressed in term of the warm-up period length this would mean the following: the longer warm-up period is, the

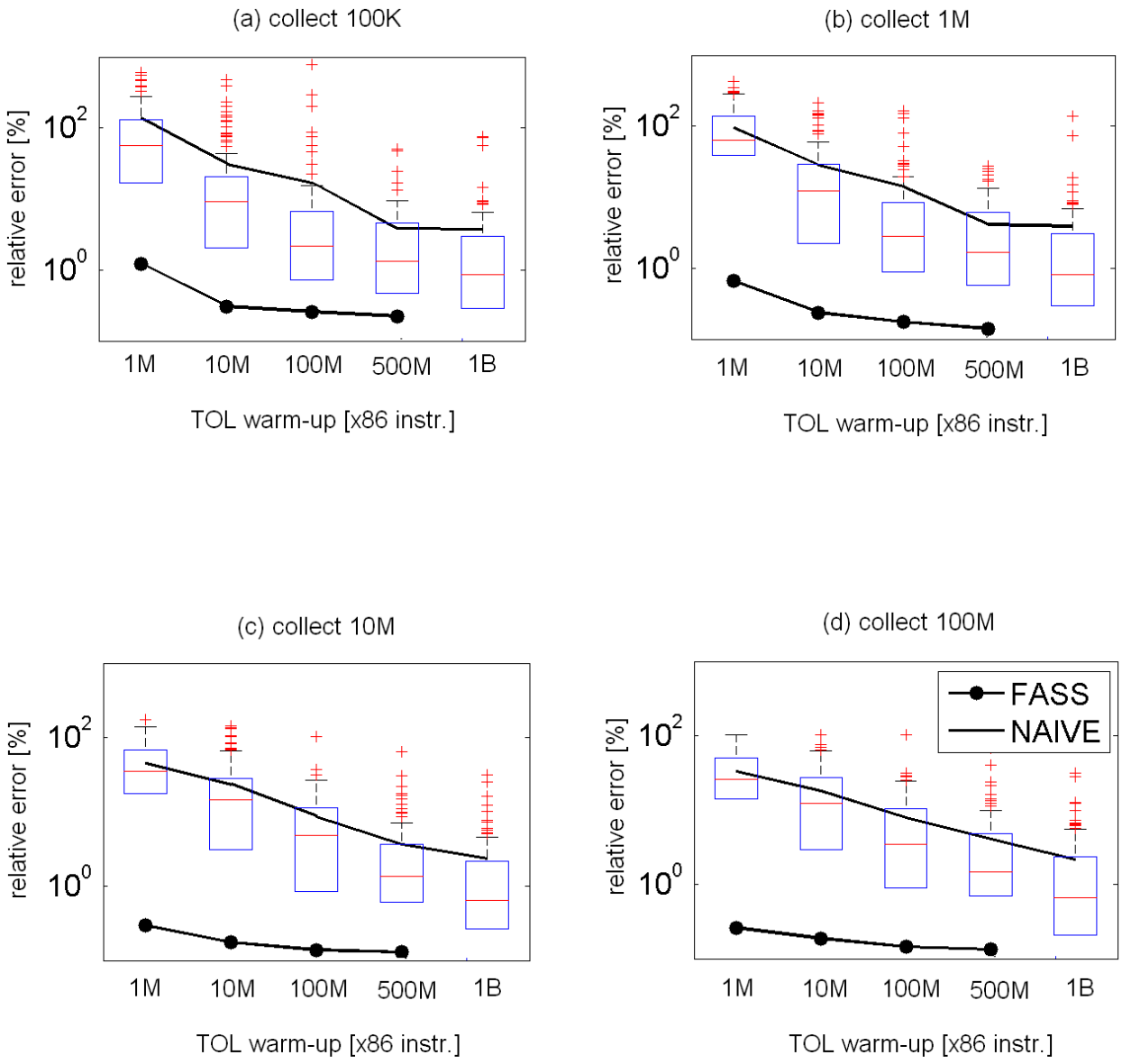

Figure 5.10: Average error for different collect periods for: Fast authoritative state simulation (FASS) and Naive TOL warm-up simulation (NAIVE). The data for the NAIVE is shown with boxplots as well. Notice that the middle line of the boxplot shows the median and not the geometric mean.

smaller inaccuracy. The same trend applies for the collect period; the longer it is, the smaller the inaccuracy. However, FASS is not feasible due to its simulation time. Compared to AS, FASS only suffers from errors that are due to inaccuracies in the HW warm-up. As for the TOL, it is identical to the state it would have for the AS.

NAIVE, on the other hand, provides a trade-off between accuracy and simulation time. Similarly to FASS, the accuracy of the technique improves as length of the warm-up or of the collect period increase. However, even for a long warm-up period or a long collect period, NAIVE is so inaccurate that its applicabil-

ity is extremely limited. Even with the longest warm-up period and the longest collect period (1B and 20M x86 instructions respectively) this technique delivers errors that grow up to 50% due to the poor warm-up of the TOL.

Whereas the FASS suffers only from microarchitectural inaccuracies, NAIVE suffers from the TOL inaccuracies as well. Given that NAIVE has a HW warm-up period of 1M x86 instructions, the difference in accuracy between NAIVE and FASS is due to the inaccuracies in TOL warm-up. As seen from the experimental data in Figure 5.10, the error due to the TOL warm-up inaccuracies is 3-4 orders of magnitude larger than the error introduced by the HW warm-up.

The results clearly indicate that NAIVE is not appropriate for simulation of HW/SW co-designed processors, since it is not able to deliver neither a good average error nor a low upper bound for the error. Therefore, techniques that are able to deliver high accuracy at limited simulation time are necessary. For clarity and simulation time needed to evaluate TH technique, in the rest of this section the results will be presented for collect period of 10M x86 instructions. According to our experimental data, the other collect periods show similar behavior.

5.7.2 Compilation Plan TOL Warm-Up Simulation (CP)

CP is based on restoring the profiling information prior to the warm-up period (Section 5.4.4). The experimental results for CP using a collect period of 10M x86 instructions are presented in Figure 5.11. The average error for NAIVE is shown in the figure in order to ease the comparison.

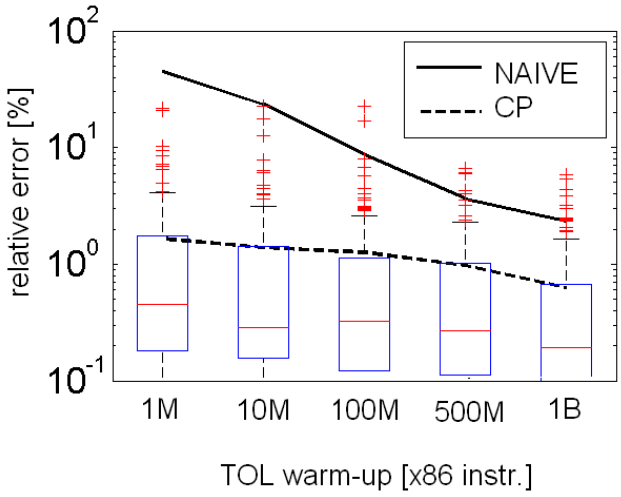

Figure 5.11: Average error for collect period of 10M x86 instructions: Compilation Plan (CP) and NAIVE TOL warm-up simulation. In addition to the trend lines, the data for CP are presented using a boxplot as well.

The experimental data show that CP performs much better than NAIVE. The average error is in the range of 1-2% even for very short warm-up periods, because the TOL is warmed-up more accurately. The maximum error (worst case scenario error) reaches up to 20-30%, because SBs are created in a different order than for the authoritative execution. Although the maximum error can be high, this technique performs significantly better than NAIVE. However, CP has other issues that limit its applicability (*general applicability concept*), explained in Section 5.4.4.

5.7.3 Downscaling Promotion Thresholds (TH)

This section compares the TH with NAIVE and CP and shows that the former is better suited.

For simplicity, we downscale only the promotion threshold from basic block to superblock (TH_{SB}). For the evaluation, we have used 9 different threshold values for TH_{SB}: 10, 20, 50, 100, 200, 500, 1K, 2K and 5K. The original promotion threshold is 10K. These 9 values are combined with the 5 warm-up periods (1M, 10M, 100M, 500M and 1B) in the simulation runs (including the default threshold) for each sample.

Section 5.7.3 shows the potential of oracle selection of the best pair $\{Th, WU_{length}\}$ and Section 5.7.3 the results of the predictive model. The predictive model delivers results very close to the oracle (which guarantees very low error) and at the same time low simulation time. Finally, in Section 5.7.3 we show that TH works well for different TOL and microarchitectural configurations.

Oracle Configuration

The oracle configuration looks off-line for the pair of $\{Th, WU_{length}\}$ across all simulated combinations such that the warm-up error is lower than a predefined, *acceptable error* ERR_{accept} (set to 1%). Let us assume that the pair $\{Th, WU_{length}\}$ gives a particular *error*, *simulation time* pair. We define the oracle pair ("o") as the one that gives the minimum simulation time given an acceptable error among all $\{Th, WU_{length}\}$ pairs:

$$(o) : timeSimulation^{(o)} = \min_i \left\{ timeSimulation^i : error^i \leq ERR_{accept} \right\}. \tag{5.10}$$

If two pairs have an error smaller than 1% (ERR_{accept} = 1%), we select the one that has the smallest simulation time. The goal is to avoid paying the extra cost if the benefit in accuracy is negligible (for example going from an error or 0.1% to 0.09% with a 10X extra simulation time).

Figure 5.12 depicts the trade-off between simulation time and error of the evaluated techniques for a collect period of 10M. The X-axis represents the average simulation time reduction with respect to AS, while Y-axis represents the average error.

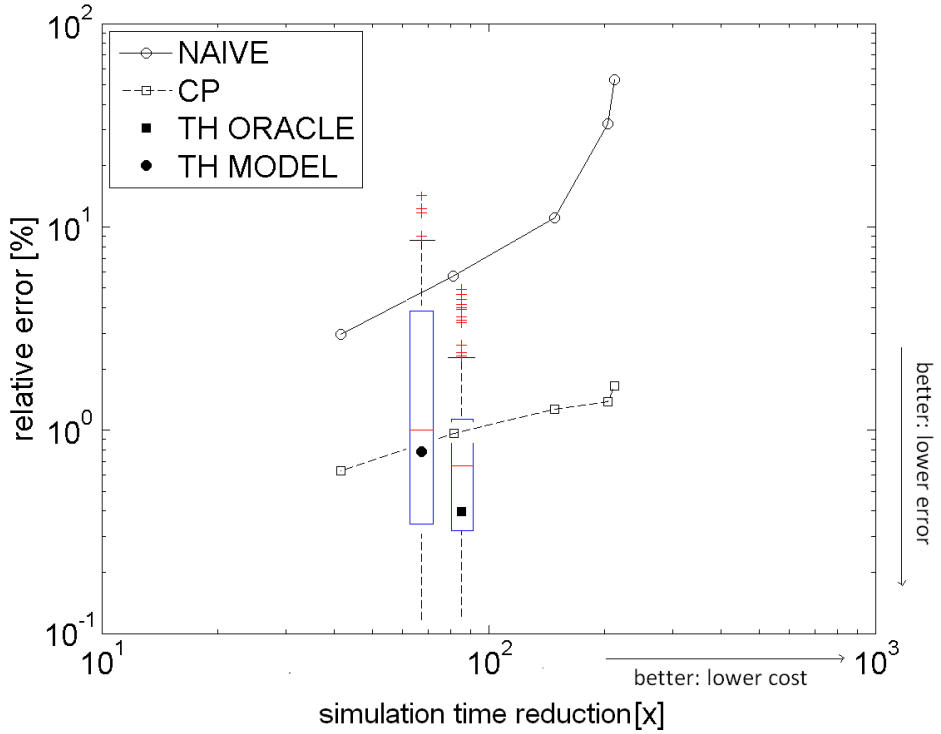

Figure 5.12: Error vs. simulation time reduction curve for NAIVE, CP and TH (ORACLE and MODEL based warm-up configurations). Boxplot is shown for TH ORACLE and TH MODEL.

In the light of experimental data, TH Oracle bounds the maximum error to 4.7% with an average error of 0.35%. NAIVE, on the other hand, has an average error of 3% with a maximum value error 50%[2]. Regarding the simulation time, TH Oracle is at least 90X faster than AS. Notice that the further the sample is from the beginning of the application the greater the simulation time reduction is.

Predictive Model

The previous section showed that the TH Oracle delivers very small average (0.35%) and maximum (4.6%) error at a high low simulation time reduction (90X). For the technique to be applicable in practice it is necessary to identify the most appropriate threshold and warm-up length pair $\{Th, WU_{length}\}$ for each application sample before simulation.

The experimental results for the predictive model, which was presented in Section 5.5.2, are depicted in Figure 5.12. The predictive model is able to achieve most of the benefits of the oracle. In particular, it achieves an average error of 0.75% with a maximum error of 15% compared to 0.35% and 4.6% for the oracle. The average simulation time reduction is 65X compared to 90X for TH Oracle. Notice that just

[2] Note that the maximum error is calculated per sample. In many papers in the literature, the maximum error is presented per benchmark and not per sample. Doing so, our results would be even better.

16 out of 120 samples have an error of more than 5%. Compared to CP (which does not meet the *general applicability concept*), TH predictive model gives better *error / simulation time* trade-off. As an example, CP with $Y_{TOL} = 500M$ has a simulation time reduction of 80X, while the average and the maximum errors are 0.9% and 20%.

Evaluation using different configurations

We have also explored the behavior of the predictive model in the case of TOL variations. In particular, we tried 2 additional TOL configurations: (*i*) TOL without performing optimizations ($TOL_{woOptimiz}$) and (*ii*) TOL without code region linking ($TOL_{woLinking}$). The average and worst case scenario errors for $TOL_{woOptimiz}$ and $TOL_{woLinking}$ are 0.56% (maximum error is 16%) and 0.37% (maximum error is 12%) respectively. The results are presented with the form of a boxplot by Figure 5.13 together with the default TOL. Notice that the number of samples with error more than 5% is 14 and 6 (out of the 120) for the $TOL_{woOptimiz}$ and the $TOL_{woLinking}$ respectively.

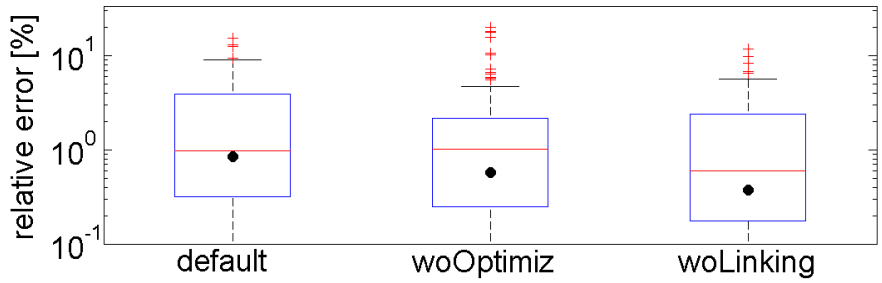

Figure 5.13: Boxplot data and average error for predictive model based warm-up configuration for different TOL configurations.

In addition to TOL variations, we also explored the effect of microarchitectural variations. In particular we simulated configurations in which we modified the memory hierarchy, the component that has the strongest effect on performance. The explored configurations are following: double and half D$, double and half L2$. Due to the simulation time limitations, we have studied just the subset of the applications (403.gcc, 434.zeusmp, 458.sjeng, 462.libcuantum, 471.omnetpp and 483.xalancbmk) in order to observe the general trend. Our results show consistency in the benefits. In comparison with the default memory configuration, on average, TH provides 1.5X bigger errors for bigger caches, while 1.5X smaller errors for smaller caches. For instance, for a configuration with the data cache having half the size we obtain an average error of 0.5% vs. an error of 0.75% for the configuration that is reported in this chapter (the simulation time reduction remains the same, 65X). Similar results were obtained for other configurations.

Based on the experimental results, we conclude that the predictive model meets all the requirements we have set for a warm-up technique. In particular:

1. it has a very low error and very low simulation time,
2. it is tolerant to TOL variations, *i.e.* it delivers equally accurate results for different TOL setups
3. it is tolerant to HW variations and
4. it does not need any TOL specific statistics which makes its applicability more general.

5.8 Conclusions

As it was shown in the previous chapter, the TOL has to be simulated on the top of the host cycle accurate simulator in order to have accurate results. This however implies additional issue, which is the TOL warm-up. Simulation in computer architecture is mostly based on restoring the state from a checkpoint or trace and warming-up the microarchitectural structures for a few million instructions before collecting statistics. Such techniques are not suitable for HW/SW co-designed processor, because the TOL state, mainly the contents of the code cache and the profiling information, needs to be warmed-up as well. Consequently, traditional simulation methodologies do not meet the accuracy vs. simulation time trade-off and they are not tolerant to different TOL and microarchitectural configurations either.

The main contributions of this chapter are the following: (*i*) we show that traditional warm-up techniques require a prohibitive simulation time to achieve low errors when applied to HW/ SW co-designed processors, (*ii*) we show that researchers should not evaluate the accuracy using just the error of the CPI for HW/SW co-designed processors but as a vector of errors containing TOL statistics as well and (*iii*) we propose a novel simulation technique for HW/SW co-designed processors that achieves an average error of 0.75% and reduces simulation time by 65X.

Chapter 6

Off-line Phase Classification for HW/SW Co-Designed Processors

This chapter proposes a technique for accurate off-line phase classification for HW/SW co-designed processors. Off-line phase classification is done in order to select the samples to be simulated. The chapter is organized as follows. Section 6.2 discusses the related work, while Section 6.3 gives relevant background. Section 6.4 proposes the TOL Description Vector (TDV) phase classification scheme. Experimental methodology is described Section 6.5, while evaluation results are presented in Sections 6.6. Finally, Section 6.7 summarizes the chapter.

6.1 Introduction

In Chapter 4 we pointed out to the need of cycle-accurate simulation, without ignoring the TOL. After that, in Chapter 5 we introduced the techniques to warm-up the state for HW/SW co-designed processors. In this chapter we focus on choosing representative samples to be simulated. As explained in Chapter 2, the idea of sampling is to represent the complete application with a subset of application snippets which are called *samples*. Ideally, each of these selected samples represents a different application's phase. The process that classifies the similar phases is based on high-level statistics and it is called *Off-line Phase Classification*. In the case of HW-only processors the statistics typically used is the basic block (BB) execution frequency [66], and the phase classification is called Basic Block Vector (BBV) phase classification. BBV is based on the assumption that each basic block of a given application behaves similarly each time executed.

In this chapter we analyze BBV for the case of HW/SW co-designed processors. Due to the nature

of HW/SW co-designed processors (the code regions are dynamically translated and they can be executed in different optimization stages), the execution time of each basic block has a significantly wider variance compared to HW-only processors. In order to back up this statement, Figure 6.1 compares the range of execution time in terms of number of cycles for a randomly chosen basic block in the 400.perlbench application for HW-only and HW/SW co-designed processor. The results clearly indicate that the execution time varies significantly more for HW/SW co-designed processors. More specifically, for HW/SW co-designed processors the execution time for this basic block ranges from 20 to 1000 cycles, whereas for HW-only processors it ranges from 15 to 100 cycles. Consequently, BBV phase classification may classify dissimilar samples as similar in the case of HW/SW co-designed processors, because the assumption that each basic block behaves similarly each time executed is not valid. On the other hand, in the case of HW-only processors each basic block behaves similarly almost always.

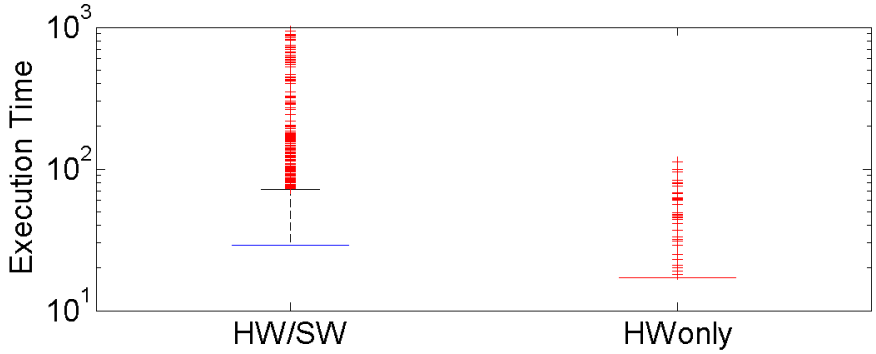

Figure 6.1: Execution time boxplot chart of the one basic block (staring from address 0x80c6d87) in 400.perlbench in two different architectures: HW-only and HW/SW co-designed.

To overcome this limitation of BBV phase classification, we propose a novel off-line phase classification scheme that targets HW/SW co-designed processors. The proposed scheme focuses on both the TOL and the code cache execution and it is called TOL Description Vector phase classification (TDV). TDV outperforms BBV not only in the accuracy / simulation time reduction trade-off, but also in profiling effort (*i.e.* it requires fewer statistics to be profiled).

In particular, the main contributions of this chapter are:

1. We show that the traditional phase classification used for traditional HW-only processors has a prohibitively low accuracy when applied to HW/SW co-designed processors.

2. We propose a novel phase classification scheme that targets the particularities of the HW/SW co-designed processors. The accuracy achieved by this technique is significantly better (3X on average) compared to the BBV phase classification, which is the traditional phase classification scheme for

HW-only processors, for the same number of samples.

3. We show that the different TOL configurations produce higher variability in simulation errors than differences in the microarchitectural configurations.

6.2 Related Work

Off-line phase classification schemes have been widely studied for HW-only architectures. The authors in [40] and [48] list, describe and compare these techniques. In all of these techniques different high-level statistics are used to describe the similarity of samples: instruction mix, loop detection, memory access addresses etc. However, the traditional phase classification is based on the Basic Block Vector (BBV) clustering. Recent studies show that BBV does not behave well in the presence of frequent L2 misses [8, 21]. What differentiates our work from the previous contributions is that we perform phase classification analysis for HW/SW co-designed architectures, where the effects of the dynamically execution and the staged optimization cause different application behavior.

The phase behavior of JIT compilers was also well targeted in the literature. The main issues are however different. In particular, these papers try to identify on-line phase changes [53, 30, 36]. JIT compilers use phase change information to improve the performance of the system. On the other hand, in this chapter we are focused on the off-line phase classification. This problem is not relevant to the current JIT compilers, since research in this area is typically preformed using real hardware and not simulators, like in the case of HW/SW co-designed processors.

Sampling methodologies have also been widely researched in the past. SimPoint [66] is probably the highest dominant technique according to which the most representative samples are determined by off-line phase classification. Besides SimPoint, there are other sampling methodologies such as SMARTS [73, 33] and COTSon [9]. SMARTS sampling methodology considers the architecture as a black box and does not analyze samples' similarity. It simply assumes random sampling and only determines the parameters for this random sampling. On the other hand, similar to SimPoint, COTSon simulates only phases, but it predicts them based on the on-line classification, using the internal simulation events. However, it is not clear how these internal simulation events apply in the case of HW/SW co-designed architectures. In this thesis we will use SimPoint as our baseline since it is the most used technique.

6.3 SimPoint and Basic Block Vector Phase Classification

This section presents the necessary background behind this work. We explain the basics and the terminology of SimPoint; and the phase classification based on Basic Block Vector (BBV), which is used by SimPoint. We also briefly describe the reason why BBV does not perform well when applied to HW/SW co-designed architectures.

6.3.1 SimPoint

To speed up the simulation process researchers typically use only few samples of a particular application, instead of simulating the whole application. The samples are usually chosen such that they present dissimilar phases [66]. This approach is followed by SimPoint and it is illustrated in Figure 6.2. Across all the samples, only n samples, out of M, are selected for the simulation.

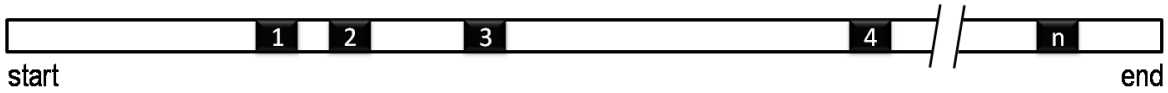

Figure 6.2: Sampling simulation approach. Only n samples (out of M) are selected for cycle-accurate simulation.

In this case, the Cycles Per Instruction (CPI) in application execution is estimated as:

$$CPI_{est} = \Sigma_{i=1}^{n} \alpha_i \cdot CPI_i^{sample}$$

$$\Sigma_{i=1}^{n} \alpha_i = 1, \tag{6.1}$$

where CPI_i^{sample} is the CPI value of the sample and α_i is the weighted coefficient of that sample. On the other hand, the real CPI value is:

$$CPI_{real} = \Sigma_{i=1}^{M} (1/M) \cdot CPI_i^{sample}. \tag{6.2}$$

Selected samples $(1, 2, ..., n)$ are usually chosen as the ones which show the most phases' dissimilarity between each other and they are weighted with different coefficients (α_i) depending on their contribution. The similarity / dissimilarity between the samples is estimated by employing the *off-line phase classification*. The entire process is illustrated in Figure 6.3.

Off-line phase classification is the process of selecting similar samples and does not require cycle-

accurate simulation results. It predicts which samples will have similar cycle-accurate statistics between each other analyzing only the microarchitectural independent statistics (*e.g.* instruction mix, BB execution frequency etc.), as depicted in Figure 6.3-a. These statistics are usually called *high-level binary* statistics and the vector which contains them is attached to every sample. For instance, in Figure 6.3-a CPI, D$ miss rate and I$ miss rate (which are microarchitectural dependent statistics) are mapped to the number of the integer instructions, the number of the loads and the number of the stores (which are microarchitectural independent statistics).

After collecting the high-level binary statistics across all samples, the phase classification algorithm is applied in order to identify the samples that are similar (Figure 6.3-b). Computer architects usually use the algorithm which contains the following phases: normalization, dimensionality reduction, clustering and representative members choosing (Figure 6.3-b).

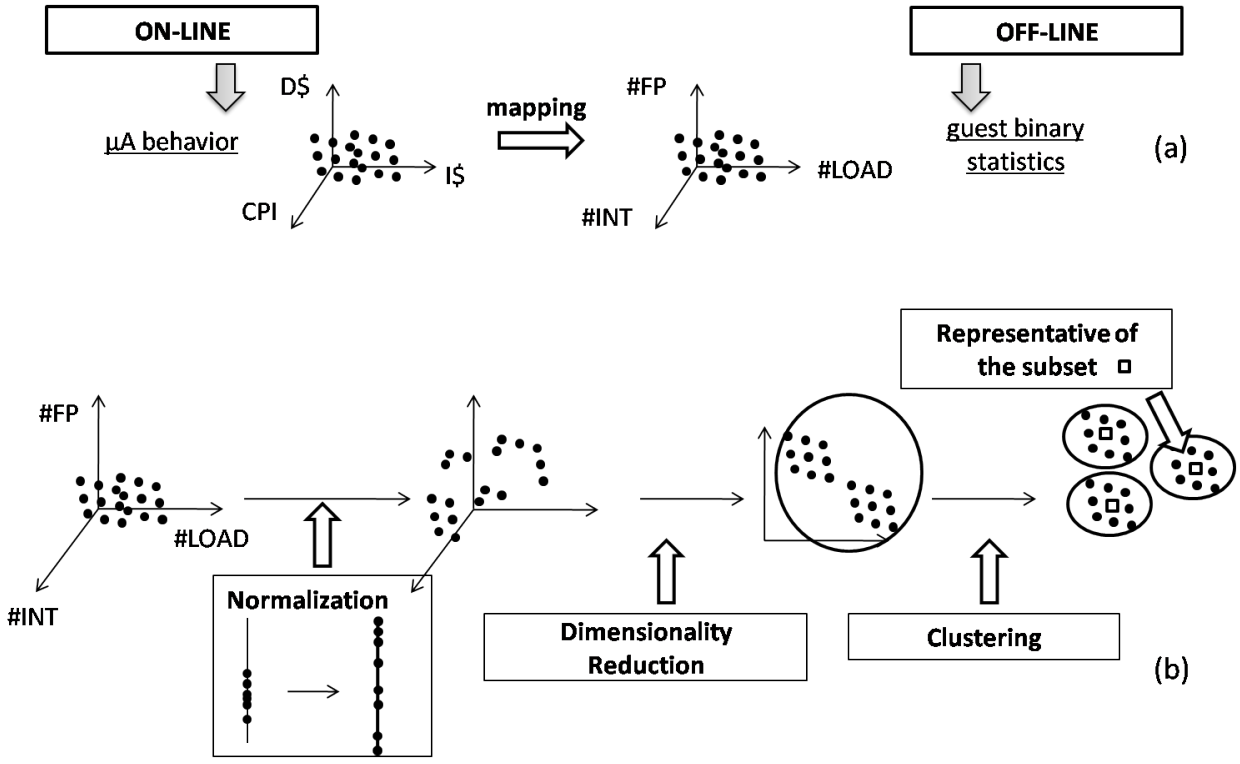

Figure 6.3: Phase Classification process in computer architecture. It contains two steps: (a) mapping between microarchitectural statistics and high-level statistics and (b) clustering process of high-level statistics.

Normalization is the process of shrinking or expanding each dimension of the off-line vector to the range [0, 1]. This is done in order to avoid favoring one dimension over the others. Figure 6.3-b depicts this process as compacting the values to the range [0, 1].

The next step is *Dimensionality Reduction*. This is needed to limit the execution time of the clustering algorithm itself; the initial dimensionality of the off-line vector can be in the order of thousands. We use

Random Linear Projection (RLP), the same scheme as the one used by SimPoint. In Figure 6.3-b this is illustrated as reducing the dimensionality from 3 dimensions to 2 dimensions.

The *Clustering* is performed at the end of the classification process. The samples are grouped according to the Euclidean distance between them. Once the groups are formed, the *representative member* of each subset is chosen as the one which has the smallest Euclidean distance to all group members (centroid approach). The representative members are selected as the samples for the simulation, while the population of each group determines the contribution weight of that sample (α_i). Hierarchical Clustering is the most used clustering approach.

Note that the goal of this chapter is not to propose a new phase classification algorithm, nor to analyze a particular phase classification algorithm. The goal of this chapter is to propose the off-line vector that is suitable for off-line phase classification of HW/SW co-designed processors.

6.3.2 Basic Block Vector based Phase Classification (BBV)

The SimPoint methodology originally performs Basic Block Vector (BBV) phase classification in order to find similar samples. BBV is a vector which contains information of the execution frequency of each basic block in a given sample. The length of the BBV is equal to the number of static basic blocks. We use BBV as the ***baseline*** in our experiments.

BBV gives accurate phase classification in the cases where each basic block has similar behavior each time executed. It lays down on the assumption that the CPI of one sample can be represented as:

$$CPI^{sample} = \Sigma_{bb=1}^{N} \omega_{bb} \cdot CPI^{bb}, \tag{6.3}$$

where CPI^{bb} is the CPI of the basic block bb, ω_{bb} is the contribution of basic block bb to the total execution time of the given $sample$ and N is the number of the basic blocks. In this case the BBV is defined as the following vector:

$$BBV = [\omega_1, \omega_2, ..., \omega_N]. \tag{6.4}$$

However, for the cases with a significant number of non-deterministic long latency events (like for example L2 misses) CPI^{bb} varies widely across executions [8, 21]. Consequently, BBV phase classification in this case will give inaccurate results.

Similar to that, in the case of HW/SW co-designed architectures, the execution of a particular basic block will not be similar every time. Notice that for HW/SW co-designed processors the CPI^{bb} metric is

observed as $gCPI^{bb}$ (guest CPI^{bb}). The main reason for high $gCPI^{bb}$ variance is the overhead cycles which correspond to TOL execution. Such an overhead is 2-3 orders of the magnitude higher than the highest penalties in HW-only architectures (*e.g.* L2 cache miss). Moreover, this overhead is not constant and totally unpredictable (as shown in Section 4), so it requires additional off-line statistics to express it.

Another reason that implies different behavior of a particular basic block is the staged optimization. One basic block can be translated through different optimizations stages (*e.g.* interpretation, basic block translation, superblock optimization) which leads different amount of cycles needed for a completion of that basic block; in lower optimization stages completion needs more cycles than for higher optimization stages. In addition, one basic block can be part of more than one superblock, which can lead to different optimizations applied for the instructions of that basic block. However in this chapter we are focused only on variable TOL behavior as a main reason for a different behavior of a particular basic block

In order to better understand the TOL behavior we need a look into its particularities to remind the reader about its operation. As explained in Section 2.1, the execution flow in HW/SW co-designed processors switches between instructions from the TOL and instructions from the code cache. In the steady state, there are two main TOL tasks: *Translation* and *TOL Look Up*. The first *translates* and stores code regions in the code cache. The next time these code regions are encountered, they are executed directly from the code cache. The second refers to the cases when the code regions are already translated and stored into the code cache. For these cases TOL intervention is needed to guarantee forward progress. This happens in the cases when the previous code region ends with an indirect branch. When an indirect branch occurs, the address of the next code region is not known and the TOL is employed. In particular, TOL performs look up in a table of translated code regions in order to find the starting address of the code region in the code cache.

TOL Look Up task is one of the main reason for non-similar execution time of a specific basic block. This is illustrated in Figure 6.4. Imagine three basic blocks: BB1, BB2 and BB3 such that the branch BB1→BB3 is a direct branch, while the branch BB2→BB3 is an indirect branch. Imagine also that the translations of these basic blocks are already stored into the code cache. In this example we are focused on the execution time of BB3. In the cases when control flow goes from BB1 to BB3, the execution time of BB3 contains only the code cache execution time. On the other side, when the control flow goes from BB2 to BB3, the execution time of BB3 contains the code cache execution time plus the *TOL Look Up* execution time, since BB2→BB3 is an indirect branch. For this reason BB3 will not have the same execution time. Although this example is related to indirect branches, similar behavior can be observed in the cases of indirect calls and returns.

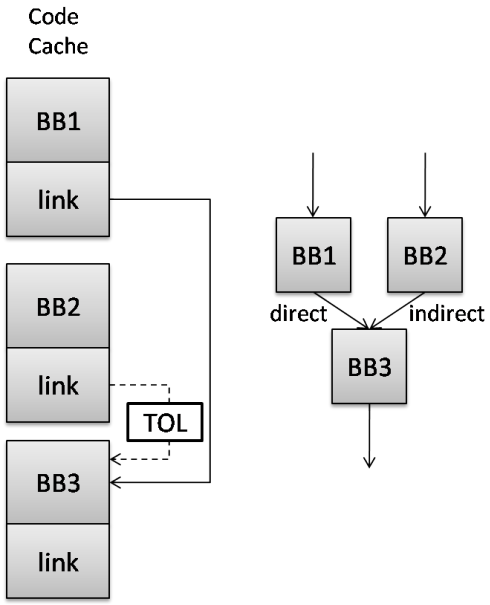

Figure 6.4: Example which illustrates the main reason for high gCPI variance per basic block in HW/SW co-designed architectures.

6.3.3 Path Profiling based Phase Classification

Another approach that can be used to find similar phases is Path Profiling [11], as the most accurate profiling. This approach is similar to BBV, with the only difference that instead of the contribution of the basic blocks path profiling profiles the contribution of the each path. In theory path profiling should give better sampling accuracy since it can distinguish scenarios which BBV cannot. For example, the execution stream *a-b-c-a-b-c-a-b-c* is different than the execution stream *a-a-a-b-b-b-c-c-c*, where *a*, *b* and *c* are different basic blocks. From BBV's perspective these two executions are the same, whereas in reality as explained they are not. One of the big issues of path profiling is to define a path. In general the path is any sequence of basic blocks executed from the beginning to the end of the application. However, this definition brings the paths that are very long and that requires very big profiling overhead. In addition these paths most probably lead to wrong conclusions because two paths may slightly differ (just few basic blocks different), which in reality means similar behavior but in path classification point of view not. Because of these issues, we tried the classifications of the paths that contains up to six basic blocks. We found that this definition of a path gives only slightly better accuracy than BBV but not better than solution presented in this chapter. The main reason is that the number of different paths grows exponentially in comparison with basic blocks, and therefore the focus of clustering is lost. Therefore in the rest of the chapter we do not study path profiling and we leave it as a part of the future work.

6.4 TOL Description Vector based Phase Classification (TDV)

This section proposes and explains the novel and simple off-line phase classification which being based on the TOL Description Vector (TDV) makes it suitable for HW/SW co-designed processors. TDV basically contains information about the static / dynamic instruction ratio (many different perspectives) and information about the instructions mix.

6.4.1 The Contents of TDV

The TOL Description Vector (TDV) has the following form:

$$TDV = [sd_1, sd_2, sp_0, sp_1, ..., sp_9, ind, im_1, im_2, ..., im_5]. \tag{6.5}$$

where the fields are statistics that can be measured by instrumentation tools. We found that with just 19 statistics TDV can lead to very accurate off-line phase classifications. In contrast BBV consists of as many elements as the number of static basic blocks (which is in the order of several thousand for some applications). Each such element counts the dynamic contribution of the particular basic block to the execution of the application. We classify these 19 statistics into four subgroups, named as *sd*, *sp*, *ind* and *im*.

Whereas BBV's view of the system is limited to the execution count of the basic blocks, TDV has a more descriptive view. It is based on the estimation of the main execution sources in HW/SW co-designed processors, which are Translation (*Transl.*), Look Up (*LUP*) and Code Cache execution (*CodeCache*). Execution time of each sample is represented as the summation of these main execution sources:

$$cycles^{sample} = cycles_{LUP} + cycles_{Transl.} + cycles_{CodeCache}, \tag{6.6}$$

where $cycles_{LUP}$, $cycles_{Transl.}$ and $cycles_{CodeCache}$ stand for the cycles spent during the *Look Up*, *Translation* and *Code Cache* execution respectively. Equation 6.6 can be written similar to Equation 6.3, in order to express $gCPI^{sample}$:

$$gCPI^{sample} = gCPI_{LUP} + gCPI_{Transl.} + gCPI_{CodeCache}. \tag{6.7}$$

$gCPI_{LUP}$, $gCPI_{Transl.}$ and $gCPI_{CodeCache}$ stand for the cycles spent during the *Look Up*, *Translation* and *Code Cache* execution per guest instruction respectively. Each of these summands is estimated using different high-level statistics. The parts that follow explain how each of these factors is estimated using high-level binary statistics.

Look Up estimation

Based on the previous studies [58], the most important contributor to the TOL is the ***Look Up (LUP)***, the module needed to guarantee forward progress of the translated application. As explained in Section 2.1 it performs look up in a table of translated code regions in order to find the starting address of the code region in the code cache. As we discussed in Section 6.3, the absence of the linking between the code regions in the code cache is the reason for LUP. Among many sources of the possible cases in which the linking is not possible, the branch indirection is predominated. Thus LUP is estimated by the number of the indirect branches, indirect calls and returns (*ind* field in Equation 6.5).

Translation estimation

The second TOL task is ***Code region translation and optimization***. The TOL invokes this process whenever a code region is encountered for the first time; the result of this process is to create and store an optimized version of the specific code region. The metric which is used in the literature to estimate this task is *static / dynamic instruction ratio* (*sd*). Higher ratio implies higher translation overhead. The rationale behind this is very straight forward. Lower ratio means that code regions are repeated more often and so the relative contribution of the translation overhead is smaller.

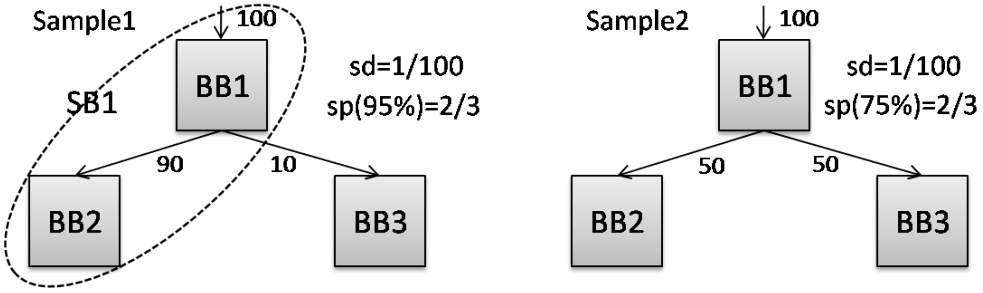

Figure 6.5: Example which illustrates same static / dynamic ratio, but different code region formation.

However, the previous metric estimates the translation overhead only when TOL is constructed as a one-stage optimizer. In case when we have more optimization stages, for instance superblocks, this might be very inaccurate because some of the basic blocks can be a part of one superblocks, some of them can be part of more than one superblocks, while some of them are not part of any superblock at all. Previous metric treats all basic blocks equally and therefore it is not enough to estimate translation and optimizations overhead that comes from superblocks. To target this we introduce a metric called *static percentage* (*sp*). Static percentage expresses the percentage of static code needed to cover *X%* of the dynamically executed code. Such a metric is important to differentiate the behavior between two samples with same static /

dynamic ratio, but different code region formation. In order to explain this situation, consider a scenario in Figure 6.5 where two samples have only three basic blocks ($BB1$, $BB2$, $BB3$) and each basic block has the same amount of instructions (*e.g.* 5). In the case of the first sample BB1 is executed 100 times, BB2 90 times and BB3 10 times; while in the case of the second sample BB1 is executed 100 times, BB2 50 times and BB3 50 times. Focusing only on *sd* metric these two samples are the same, which is wrong because in the case of the first sample branch $BB1 \rightarrow BB2$ is biased and the superblock $SB1$ is constructed, whereas in the second case that superblock is not constructed. On the other hand *sp* metric differentiates these two samples. The formal definition of *sp* is as follows:

$$sp(sample, X) = max_{S_X \subseteq S_{total}} \frac{|static(S_X)|}{|static(S_{total})|}$$

$$\frac{|dynamic(S_X)|}{|dynamic(S_{total})|} \leq X, \tag{6.8}$$

where S_{total} is the set of all instructions in a given sample, while functions $static()$ and $dynamic()$ respectively corresponds to number of static and dynamic instructions executed by the sample. According to our experiments, the set of values for X that optimizes the estimation of $gCPItransl$ is:

$$X = \{90\%, 80\%, 70\%, 60\%, 50\%\}. \tag{6.9}$$

The *sd* and *sp* metrics are measured in two different ways: in terms of guest instructions and in terms of basic blocks. The number of the guest instructions estimates how costly translation overhead will be per code region, while the number of basic blocks estimates how many code regions will be constructed. TDV includes both these fields, sd1 regards the static / dynamic instructions ratio whereas sd2 the static / dynamic basic block metric. Similar applies for *sp* (*sp0*, *sp1*, ...*sp9*).

Code Cache estimation

The translated application is stored into the code cache. Although the behavior of the ***Code Cache*** can be estimated by the BBV (because TOL overhead is the main reason for non similar behavior of a basic block), we found that using just the instruction mix gives higher accuracy. Instructions are classified into five main types: integer (INT), floating point (FP), load (LD), store (ST) and branches (BR) and for each type the TDV includes the number of dynamically executed instructions (*im1, im2, im3, im4, im5*).

6.4.2 Correlation between TDV and gCPI

Figure 6.6 shows the correlation between the summands in Equation 6.7 ($gCPI_{LUP}$, $gCPI_{Transl.}$ and $gCPI_{CodeCache}$) and each of the statistics used for TDV (X-axis). The higher the correlation is, the strongest the dependency between the particular summand and the particular field in TDV. For example, the dependency between $gCPI_{Transl.}$ and static / dynamic instruction ratio is quite strong with a correlation of 0.48.

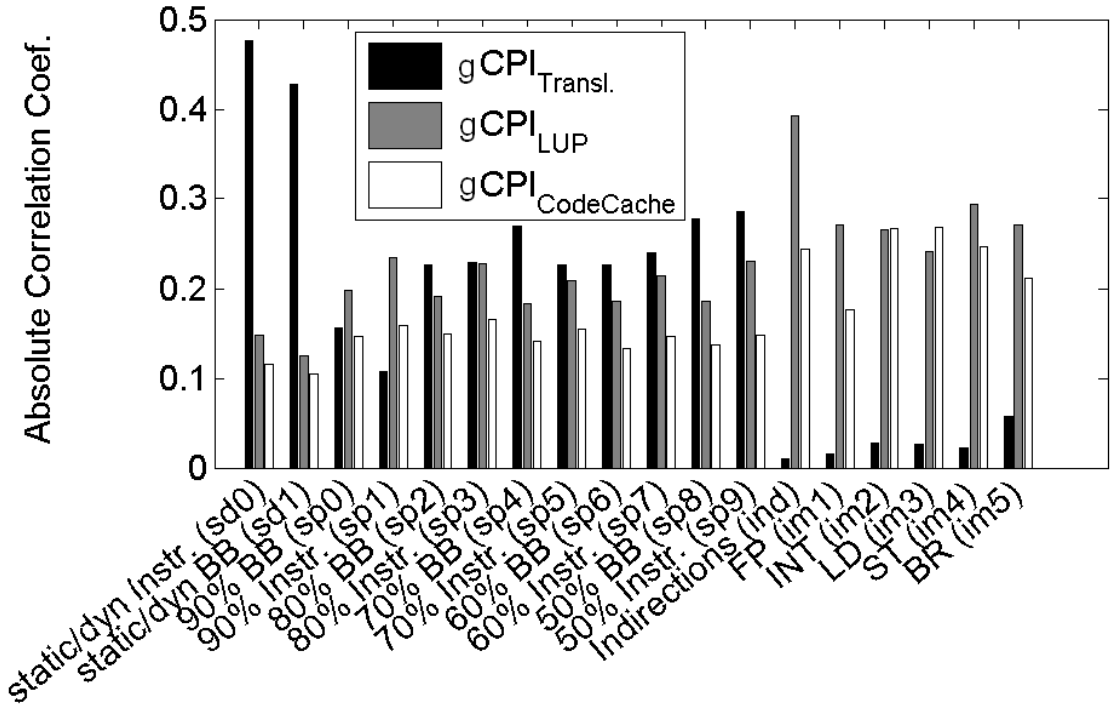

Figure 6.6: Correlation of the each field of TDV with three main components of gCPI in HW/SW co-designed processors.

As can be observed the $gCPI_{LUP}$ correlates the most with the number of indirect branches, indirect calls and returns (correlation of 40%) which confirms the assumption from Section 6.4.1. On the other hand, $gCPI_{Transl.}$ highly correlates with static / dynamic ratio (*sd0* and *sd1*). Other metrics introduced in Section 6.4.1 ($sp0 - sp9$) also show high correlation with $gCPI_{Transl.}$. Finally, $gCPI_{CodeCache}$ correlates the most with Instruction Mix ($im1 - im5$), as assumed in Section 6.4.1.

6.4.3 TDV vs. BBV Example

Table 6.1 shows an example where TDV clearly outperforms BBV. It contains three samples (**A**, **B**, **C**), which come from the 410.bwaves application, in three different cases (spaces): (a) TDV, (b) BBV and

Table 6.1: Three samples (A, B, C) of 410.bwaves application represented in different statistics space: (a) TDV, (b) BBV and (c) gCPI. The shaded lines refer to the similar samples for a given statistics space.

(a) TDV

	sd1	sd2	sp0	sp1	sp2	sp3	sp4	sp5	sp6	sp7	sp8	sp9
A	691	131	0.57	0.72	0.42	0.60	0.30	0.47	0.22	0.39	0.16	0.29
B	739	131	0.54	0.72	0.37	0.58	0.25	0.43	0.17	0.27	0.11	0.20
C	721	133	0.56	0.73	0.39	0.58	0.28	0.45	0.20	0.36	0.13	0.24

	ind	im1	im2	im3	im4	im5
A	0.22	0.01	0.23	0.42	0.13	0.00
B	0.24	0.01	0.59	0.08	0.08	0.00
C	0.23	0.01	0.38	0.28	0.09	0.02

(b) BBV

	ω_1	ω_2	ω_3	ω_4	ω_{10}	ω_{11}	ω_{12}	ω_{13}	ω_{16}	ω_{18}	ω_{19}
A	0.01	0.00	0.01	0.00	0.17	0.54	0.21	0.01	0.00	0.01	0.02
B	0.03	0.03	0.04	0.02	0.14	0.47	0.20	0.01	0.02	0.01	0.02
C	0.11	0.09	0.14	0.06	0.10	0.30	0.11	0.01	0.06	0.00	0.01

(c) gCPI

	gCPI
A	1.72
B	2.27
C	1.96

(c) gCPI. This application is chosen as the one which has relatively low number of static basic blocks, so the BBV has a reasonably small length and it is easy to manually compare all the fields in BBV, whereas at the same time it has a relatively high TOL overhead compared to other applications.

Table 6.1 shows that samples **A** and **C** are similar in the TDV and gCPI space. Similarity between the samples is calculated as Euclidean distance. For instance, if samples **A** and **B** have smaller Euclidean distance between each other than samples **A** and **C**, we say that **A** and **B** are *more similar* than **A** and **C**. Having a careful look at TDV we can notice that among all fields, sample **C** is closer to sample **A** than to sample **B**. Across all TDV fields the values collected for the sample **C** are closer to the values of the sample **A** than the values of the sample **B**.

On the contrary, in the BBV space samples **A** and **B** are similar, which is against the gCPI space. Consequently BBV will give worse similarity analysis than TDV in this case. The reason why in BBV space samples **A** and **C** are not similar is that BBV considers all basic blocks equally, while for HW/SW co-designed processors basic block which ends with an indirect branch is more important than the others. In particular in Table 6.1 basic blocks with indirect branches are BB4, BB12 and BB16, with the BBV coefficients ω_4, ω_{12} and ω_{16} respectively.

6.5 Experimental Setup

Similar to previous chapter, we used only SPEC2006 benchmark suite to evaluate different phase classification schemes. Different phase classifications are evaluated using 100 samples per each application

spread uniformly up to 200B x86 instructions (Figure 6.7). This would mean that dissimilar phases are chosen among those samples. Each sample is 10M x86 instructions long. This amount of samples is used for the simulation time reasons. SPEC2006 is a very large benchmark suite, at least an order of magnitude bigger than SPEC2000, so cycle-accurate simulation of whole dynamic application stream is almost not feasible.

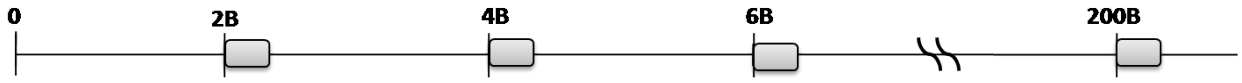

Figure 6.7: Experimental setup - studied samples.

In order to study how the different clustering schemes apply to HW-only vs. HW/SW co-designed processors, we modeled the former as a HW/SW co-designed processors with only BB translation and without feeding the cycle-accurate simulator with TOL instructions. By having only the BB translation the stage optimizations effect, one of the main particularities of HW/SW co-designed processors, which is staged optimizations, is removed from the modeled architecture. Another important particularity of HW/SW co-designed processors is TOL overhead and this particularity is removed by not sending TOL instructions to the cycle-accurate simulator. Therefore such a simulation setup very closely models a HW-only processor, because the main particularities of HW/SW co-designed processors are removed from the design.

The accuracy of the simulation technique is expressed by the average error of the estimation of gCPI among all applications. The accuracy of gCPI is calculated as the relative error between the estimated gCPI (Equation 6.1) and the real gCPI (Equation 6.2): Finally, the average error is computed across all applications as the arithmetic mean:

$$error = avg_{application}(error_{gCPI}^{tecnhique}) \tag{6.10}$$

6.6 Results

This section shows the results. Firstly the results of BBV phase classification for HW-only and HW/SW co-designed processors are shown. After that, we compare BBV and TDV phase classification in the case of HW/SW co-designed processors. Finally the error trends for different microarchitectural configurations and others statistics are discussed.

6.6.1 BBV: HW/SW vs. HW-only

Figure 6.8 shows the simulation error of BBV phase classification for two cases: HW-only and HW/SW co-designed processors. X-axis presents the normalized number of samples selected for simulation, while the Y-axis presents the average relative error. As the figure shows, for the BBV phase classification scheme, the HW/SW co-designed processors suffer from significantly higher error compared to the HW-only designs. Moreover it backs up the initial assumption that BBV phase classification does not provide a good trade-off between accuracy and the number of samples.

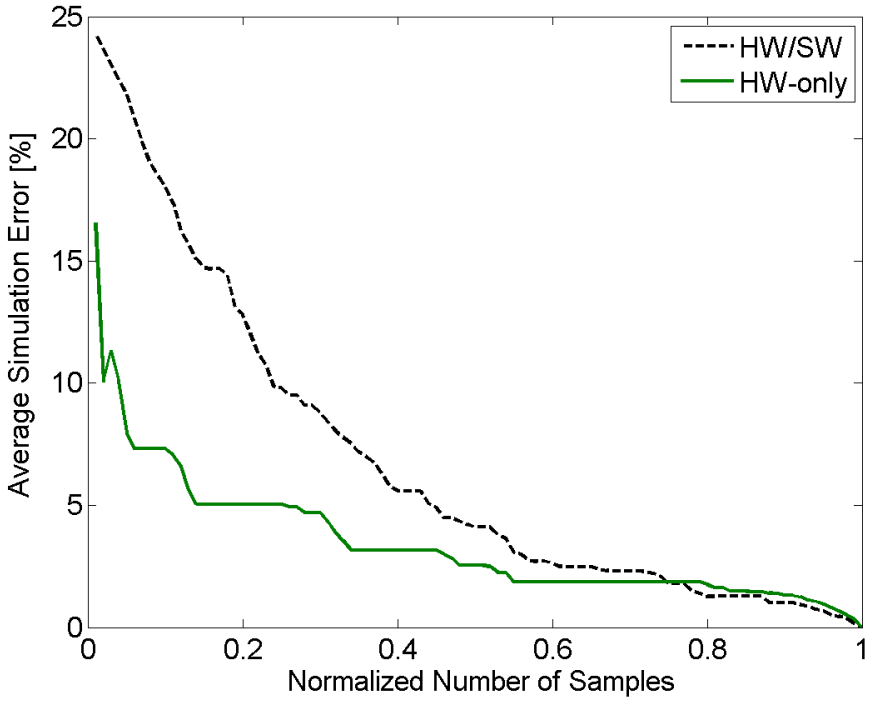

Figure 6.8: Average relative error vs. number of selected samples for BBV phase classification in two different scenarios: HW-only and HW/SW co-designed processor.

For instance, for 20 samples per application (which makes normalized the number of samples equal to 0.2), we have an average simulation error of 13% for HW/SW co-designed processors and 5% for HW-only architectures; a difference of 2.5X. On the other hand, in order to have a simulation error of 5% HW/SW co-designed processors need 2X more number of samples than the HW-only processors. As already mentioned, the main reason for such behavior is the TOL overhead. Our results also show that the applications with the highest simulation error are the ones with the high TOL overhead (higher than 10% of the entire execution).

To eliminate the Random Linear Projection (Section 6.3) from being the reason behind the high inaccuracy of the BBV we performed studies with different lengths for the Random Linear Projection (RLP)

vector. In particular we studied the effect of increasing the default value of the RLP length from 15 to 20, 30, 50 and 100 respectively. On the contrary to the expected behavior, having bigger length of the RLP vector introduces higher errors and thus cannot be the reason for high error in the case of BBV phase classification. Such behavior is due to the nature of RLP, where the lower dimensionality of RLP makes statistical space more spherical and easier for clustering.

6.6.2 HW/SW: BBV vs. TDV

The results presented in Figure 6.9 compare the BBV and the TDV phase classification in the case of HW/SW co-designed processor. Similar to Figure 6.8, X-axis presents the normalized number of samples, while the Y-axis presents the average relative error.

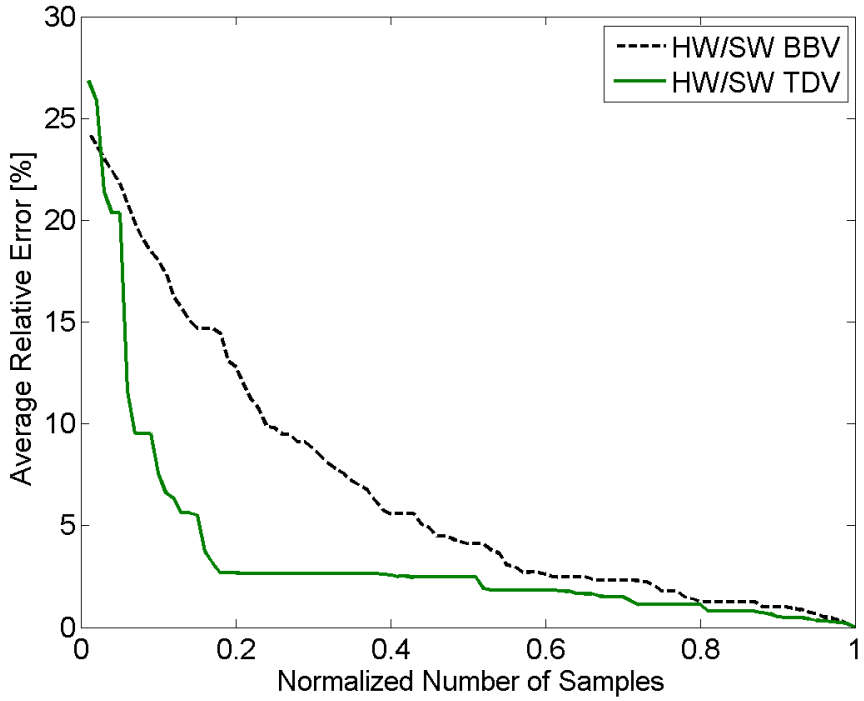

Figure 6.9: Average relative error vs. number of selected samples for BBV and TDV phase classification in the case of HW/SW co-designed processor.

The first observation from Figure 6.9 is that TDV has errors smaller than BBV. For 20 samples per application, TDV delivers a simulation error of 3%, whereas BBV delivers an error of 13%, an improvement of more than 4X. As can be observed, when the number of samples is between 10 and 40 (between 0.1 and 0.4 in Figure 6.9) the average simulated error in the case of TDV is around 2-4X smaller than in the case of BBV. The benefit of TDV can be also seen from a different perspective, which is the simulation time reduction. In order to reach the relative error of 3%, which is the typical threshold used in the literature,

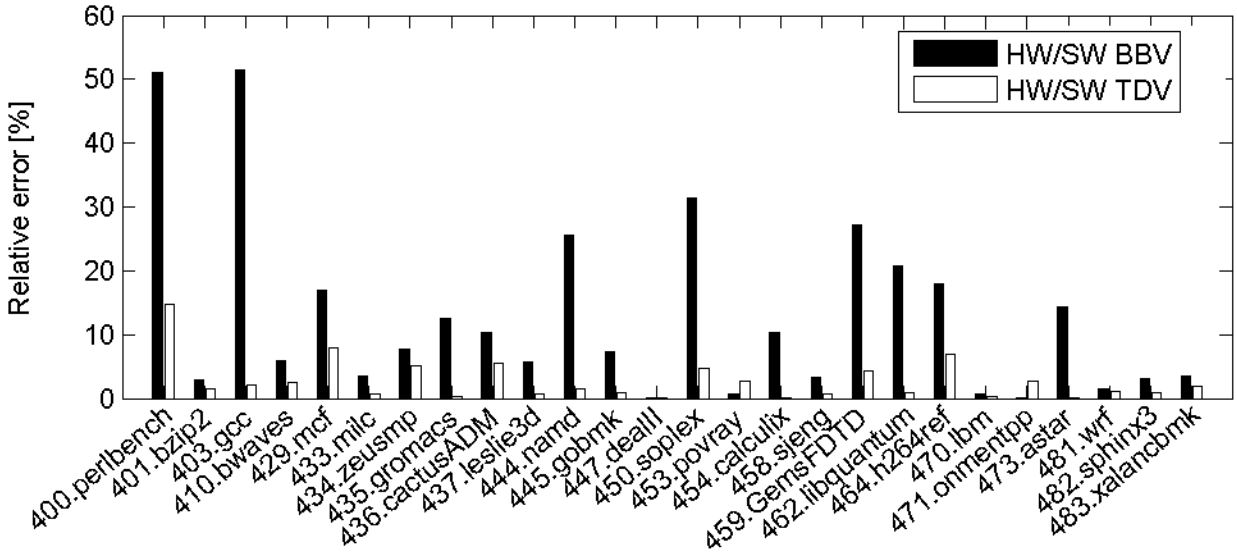

Figure 6.10: Relative error per application (SPEC2006) for BBV and TDV phase classification in the case of HW/SW co-designed processor for a fixed (k=20) samples per application.

TDV requires 3X lees number of samples than BBV.

Figure 6.10 shows the simulation error across all applications from SPEC2006 benchmark suite in the case of 20 samples selected for each application. Across all samples TDV phase classification is almost always better than BBV phase classification. Only in two cases (453.povray and 471.onmentpp) BBV gives better accuracy than TDV. However, these cases are cases with small error (up to 3%). In these cases the TOL overhead is minimal and the behavior of the application executed on the HW/SW co-designed processors is similar to the execution on the HW-only processor.

Moreover, notice that BBV has a maximum error of 50%, whereas TDV bounds the maximum error to 16%. For these applications in order to have acceptable error, more samples have to be simulated. For instance for 400.perlbench, simulating 35 samples limits the error for TDV to 4%. Finding the number of the samples needed to deliver a given simulation error is out of the scope of this thesis. However the approach used in SimPoint, based on Bayesian Information Criterion [66], can be used.

There are 11 applications with simulation error which is higher than 10% for case of BBV phase classification. It is important to notice that for all of these cases TDV significantly outperforms BBV. Moreover, only for one application (400.perlbench) TDV gives an error more than 10%. The reason behind it is based on the fact that 400.perlbench has high number of static superblocks which makes its behavior very unpredictable under a HW/SW co-designed execution scenario, as explained in Chapter 4.

6.6.3 HW/SW: Different Configurations

This subsection explores the accuracy of the TDV phase classification for different configurations. A successful classification scheme should be tolerant to such variations; especially for the HW/SW co-designed architectures where the design space is wider compared to HW-only architectures.

We study the variations for both components: microarchitectural (HW) and TOL (SW). For the variations of TOL, we studied the following configurations:

- default
- without Optimizations
- without Linking

For the variations of microarchitecture we studied:

- double data cache
- half data cache
- double second level cache
- half second level cache
- four times smaller branch predictor history

The results for TOL variation are presented in Figure 6.11. Two major observations can be made. First, for every TOL configuration and when less than 40 samples are selected, TDV still has 2-3X smaller error than BBV. The second observation is that the accuracy depends on the specific configuration. For instance TOL *without Linking* has around 3-4% bigger error than default TOL. This is the case for both, BBV and TDV phase classification. For BBV it is caused by the fact that disabling linking increases the execution time, which makes the range of gCPI values for particular block even wider than in the case where linking is enabled. On the other side, the Look Up behavior is drastically changed (since there is no linking, the Look Up is employed after every code regions is executed), so the TDV is misspredicts the Look Up behavior. Similar happens when optimizations are disabled. Therefore, these studies show that for different configurations different number of samples is needed to deliver particular error.

The error of the variation in the case of different microarchitectural configurations is negligible. In other words, the error curve is the same like the one presented in Figure 6.9. Such a behavior is expected,

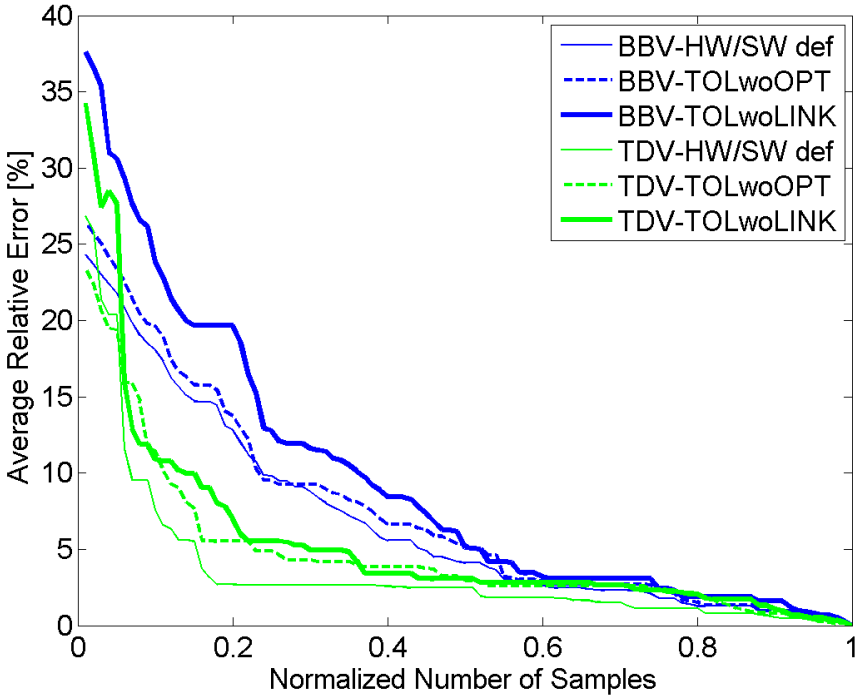

Figure 6.11: Average relative error vs. number of selected samples for BBV and TDV phase classification in the case of different TOL in HW/SW co-designed processor: (*i*) default (HW/SW def), (*ii*) without Optimizations (TOLwoOPT) and (*iii*) without Linking (TOLwoLINK).

since the host hardware is a simple in-order processor, so for different microarchitectural configurations, for a particular application the amount of cycles is just scaled similarly up or down across all samples. Therefore, the error remains almost the same.

6.6.4 HW/SW: Other Statistics

Researchers usually do not pay attention only to gCPI as it is mentioned in Section 3.3, but also to some other relevant statistics. There are many scenarios in which two simulations may give similar gCPI, but different behavior. These are similar to ones presented in Section 5.6. Imagine the following scenario. The real application behavior is such that big part of the code is in the highest optimization level, but it is not linked efficiently due to a large number of indirect branches. Then after applying the sampling technique, we choose the samples which have less code regions in the highest optimization level, but these code regions are efficiently linked. The amount of cycles lost by having worse code generation is compensated by avoiding the TOL LUP execution, so summarizing we would have the similar number of cycles for both simulations (authoritative and sampled simulation). Just based on gCPI accuracy, we could say that sampling is accurate, whereas in reality it is not.

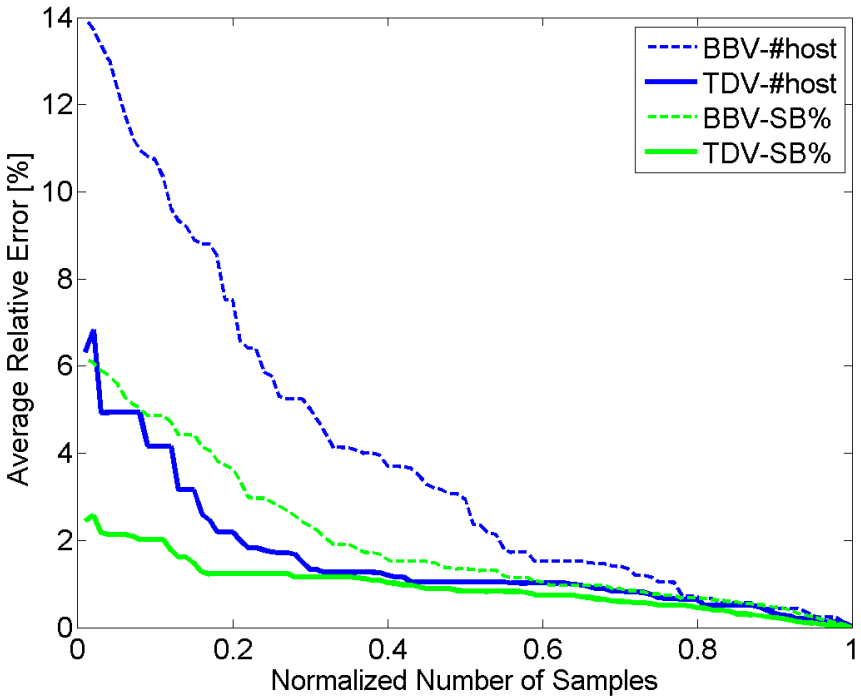

Figure 6.12: Average relative error vs. number of selected samples for BBV and TDV phase classification in the case of HW/SW co-designed processor for number of host instructions (#host) and SB coverage (SB%).

Figure 6.12 shows the error of other statistics such as: the number of the host instruction and the SB coverage, for TDV and BBV phase classification. This shows that the errors are smaller than the error of gCPI, and that TDV has around 2X better accuracy than BBV. The results also show that although BBV in some cases estimates accurately the code cache behavior, it does not express accurately SB coverage. This means that $sp0$ - $sp9$ fields in TDV express more accurately the SB coverage than BBV. This is due to the fact that BBV does not have any information about superblock estimation.

6.7 Conclusions

Besides a warm-up simulation technique, which was discussed in previous chapter, the simulation of traditional microprocessor is also based on simulating a few samples across whole application. Samples are usually specified by off-line phase classification. However, traditional off-line phase classifications do not perform well for HW/SW co-designed processors, in which a Transparent to the entire software stack Optimization Layer (TOL) dynamically translates and optimizes guest instructions to an internal host ISA.

In short the contributions of this chapter were the following: (*i*) we show that traditional Basic Block Vector (BBV) phase classification is not suitable for HW/SW co-designed processors, (*ii*) we propose a novel phase classification called TOL Description Vector (TDV). On average, TDV phase classification

reaches the same error like BBV with 3X less number of samples. Such a trend does not depend neither on different TOL nor on microarchitectural configurations.

Chapter 7

Conclusions and Future Work

This chapter exposes the main conclusions of this thesis and also discusses the future work. In a nutshell, this thesis proposed techniques that provide accurate simulation results at the low simulation cost for HW/SW co-designed processors. HW/SW co-designed processors require different simulation techniques in comparison with traditional HW-only processors due to the different nature of the systems, which includes dynamic binary translation, staged optimizations etc.

7.1 Conclusions

HW/SW co-designed processors are a new vehicle for low-power architectures, without sacrificing performance. In order to do that, they are equipped with a software layer that dynamically analyzes, profiles, translates and optimizes instructions from a guest ISA to an internal microarchitecture with its own customized host ISA. The software layer, which is usually organized as a staged optimizer, in this thesis was called Transparent Optimization software Layer (TOL). Translated and optimized code regions are stored in a *code cache* and in the steady state most of the code is fetched and executed from it. Solutions for HW/SW co-designed architectures range from an entire hardware implementation to an entire software implementation, with many of them requiring support from both components. In order to evaluate all these trade-offs, *fast and accurate* cycle-accurate simulation of the entire HW/SW co-designed processor is required.

In this thesis we proposed simulation techniques to overcome the issues related to simulation and evaluation of HW/SW co-designed processors. We first discussed and analyzed the current simulation techniques which assume a constant or ignored TOL behavior and therefore simulate only code cache behavior (Chapter 3). We demonstrated that simulation errors using these techniques can be unacceptably high and

consequently the simulations results may cause wrong conclusions. This huge simulation error mostly comes from non-constant and not-predictable microarchitectural behavior of Transparent Optimization software Layer of the HW/SW co-designed processor which leads to the conclusion that the cycle-accurate simulation of TOL is absolutely necessary, including the microarchitectural interaction between the TOL and the code cache as well.

Having in mind that HW/SW co-designed processors require the cycle-accurate simulation, what this thesis studies next is warm-up and sampling simulations techniques for HW/SW co-designed processors. These techniques are used for a simulation of HW-only systems in order to have accurate simulation results with a reasonable simulation time. We demonstrated that such techniques deliver an unacceptable error when they are performed for HW/SW co-designed processors. In the case of the warm-up we showed that the acceptable simulation error requires 3 orders of magnitude longer warm-up period. In addition to that, when sampled simulation technique is performed, similar error requires 3X more samples. Therefore in this thesis we proposed novel warm-up and sampled simulation techniques utilized for HW/SW co-designed processors.

Firstly, in Chapter 5, we proposed novel warm-up methodology based on downscaling promotion thresholds that are used to promote code regions to higher optimization stages. During the warm-up period the promotion thresholds that are used to promote code regions to higher optimization stages are downscaled to the value which enables faster promotion. We demonstrated that such a technique has a high potential because the best selection of the downscaled threshold value and the warm-up length delivers the error of only 0.4% with the 90X simulation time reduction in comparison with authoritative simulation. In addition to that, we proposed the scheme to predict suitable values for downscaled thresholds and warm-up length. The scheme is based on searching for the values which scales the best execution histogram of instructions contained into collect interval.

Secondly, in Chapter 6, we presented sampling methodology that is based on describing the HW/SW co-designed particulates separately: the TOL and the code cache. In order to find the best suitable sampling methodology (or off-line phase classification as it is called in this thesis) we first searched for the statistics which show some level of correlation with statistics related to HW/SW co-designed processors. Our solution, called TOL Description Vector (TDV) gives similar simulation error comparing with methodologies used in HW-only systems, but using 3X less simulation time.

7.2 Future Work

Since this is the first work that points to the simulation for HW/SW co-designed processors, in this thesis we studied only the *performance* simulation for *simple* (uni-core) architecture. However, the power dissipation is also very important metric and an architecture design cannot be fully evaluated without it. Power simulation, similarly to performance simulation, when performed with traditional techniques might suffer from inaccuracies in the case of HW/SW co-designed processors. Therefore, the simulation techniques presented in this thesis, the sampling and the warm-up, have to be verified in the case of a power simulator as well. At the end of this thesis, many reliable power simulators were published, such as McPAT [49], and its incorporation to DARCO tool seems to be a feasible process. However, for a first estimation power can be roughly estimated by the number of dynamically executed host instruction, which was already reported in this thesis. As the results show, the error of the number of dynamically executed host instructions is acceptable in both techniques proposed in this thesis.

Furthermore, as it was explained in this thesis only uni-core architectures are discussed. Another interesting and very important question is to know how the techniques presented in this thesis are going to be scaled for the case of multi-core processors. This is important because the general trend in a processor design is oriented to multi-core solutions. For this to be studied, it would need an additional step in development of DARCO infrastructure. However we believe that general trend shown in this thesis will not change and that it is only the question how big will be the gap between the state of the art techniques and the techniques proposed in this thesis.

HW/SW co-designed processors are based on simple in-order host cores, so that kind of host core was studied in this thesis. However it might be interested to observe how the effects described in this thesis are going to apply for out-of-order host cores. In that case the TOL overhead penalty will still be order of magnitudes bigger than the last level memory cache miss, so we believe that the trend of results will stay the same. This is important because if we take into account many types of cores, we can expand the scope of techniques presented in this thesis to the wider spectrum of architectures such as virtual machines, dynamic binary translators etc.

The work presented in this thesis also has some interesting extensions. Although we covered a wide spectrum of different aspects of the simulation of HW/SW co-designed processors, there are still some open questions and issues that require further analysis. For instance, in this thesis we have demonstrated that Transparent Optimization Layer (TOL) does not have constant microarchitectural behavior and therefore it

has to be simulated together with the code cache. We have also showed that simple prediction model that is based on high-level statistics is not suitable. An extension of this work can focus on analysis of some more sophisticated estimating algorithms or even propose a novel estimating algorithm. The procedure in this is case is similar to the one presented in this thesis.

Another issue to explore is the further optimization of the prediction model used in Downscaling Promotion Thresholds technique which is used as the warm-up approach. Although we showed that the current model gives high fidelity of accurate estimation compared to the oracle, there are some cases where the model is not optimal in comparison with it. The further analysis may lay on the approach in which the scaled PC execution histogram is differently interpreted because it over- or under- estimates the histogram of full execution.

We also showed in this thesis that simulation can lead to misleading conclusions if results are evaluated using only the guest CPI (gCPI) metric. Therefore we proposed a simple methodology where other errors, such as errors of the number of host instructions and dynamical coverage of superblocks, are also taken into consideration. In order to have more reliable mechanism to judge the correctness of a simulation methodology, some broader analysis of this phenomenon has to be explored. Basically it is necessary to find the set of the metrics that are enough to bias the errors of other metrics. These analyses could be based on brute force search or some smarter solutions such as genetic algorithm.

Although the benchmarks are usually developed such that they do not show redundancy in microarchitectural behavior, many publications show that actually they do. Many characterization studies are done in order to find the set of benchmarks that behave similarly for HW-only processors. Researchers use these studies for HW/SW co-designed processors as well, even though HW/SW co-designed processors might show different behavior as it is shown in this thesis. Therefore, it would be interesting to analyze these characterizations for HW/SW co-designed processors in comparison with HW-only processors.

Finally, another open question for the evaluation of HW/SW co-designed processors in general, is related to hardware counters. Hardware counters are defined as a set of special-purpose registers built into processors in order to store the counts of hardware related activities. Different processors have different number of hardware counters and different hardware activities can be measured. There are two major issues for usage of hardware counters in the case HW/SW co-designed processors. The first one is related to the fact that instruction mix of guest application is different from the instruction mix executed on host hardware, due to different ISAs. This would mean, for example, that the number of the memory access instructions will be different for host and guest ISA and therefore it is difficult to define a cache miss from the guest

point of view. Same applies for other hardware events such as branch misspredict ion etc. The second issue is related to the set of statistics that have to be supported by hardware counters. In other words we have to find a set of statistics that characterizes the most HW/SW co-designed processors. For instance in the case of HW-only processors these statistics are cache misses, branch prediction misses, whereas for HW/SW co-designed processors they are still not known. Therefore it would be interesting to have some studies that address aforementioned issues.

Bibliography

[1] DynamoRIO (http://dynamorio.org/).

[2] Mediabench II Benchmark (http://euler.slu.edu/ fritts/mediabench/).

[3] Pin intrumentation tool (http://www.pintool.org/).

[4] Quick emulation tool (http://http://www.qemu.org/).

[5] Standard Performance Evaluation Corporation. SPEC CPU2006 benchmarks. (http://www.spec.org/cpu2006/).

[6] A. Aho and J. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[7] E. Altman, M. Gschwind, S. Sathaye, S. Kosonocky, A. Bright, J. Fritts, P. Ledak, C. Agricola, and Z. Filan. BOA: The Architecture of a Binary Translation Processor. *IBM Research Report RC*, 2000.

[8] M. Annavaram, R. Rakvic, M. Polito, J. Bouguet, R. Hankins, and B. Davies. The Fuzzy Correlation between Code and Performance Predictability. In *Proceedings of the International Symposium on Microarchitecture*, ISPASS 37, pages 93–104, 2004.

[9] E. Argollo, A. Falcon, P. Faraboschi, M. Monchiero, and Daniel Ortega. COTSon: Infrastructure For Full System Simulation. *Operating Systems Review*, pages 52–61, 2009.

[10] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, 2000.

[11] T. Ball and J. R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, 1996.

[12] S. Blackburn, K. McKinley, R. Garner, C. Hoffmann, A. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Moss, and A. Phansalkar. Wake

Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Commun. ACM*, 51(8):83–89, August 2008.

[13] D. Boggs, G. Brown, B. Rozas, N. Tuck, and K. S. Venkatraman. NVIDIAs Denver Processor. In *Proceedings of a Symposium on High Performance Chips*, HOT CHIPS '14, 2014.

[14] E. Borin and Y. Wu. Characterization of DBT overhead. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '09, pages 178–187, 2009.

[15] A. Branković, K. Stavrou, E. Gibert, and A. Gonzalez. Analysis of CPI variance for DBTO overheads. In *Proceedings of the 5th Workshop on Architectural and Microarchitectural Support for Binary Translation, held in conjuction with ISCA-39*, AMAS-BT '12, 2012.

[16] A. Branković, K. Stavrou, E. Gibert, and A. González. Performance Analysis and Predictability of the Software Layer in Dynamic Binary Translators/Optimizers. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 15:1–15:10, 2013.

[17] A. Branković, K. Stavrou, E. Gibert, and A. González. Accurate Off-line Phase Classification for HW/SW Co-Designed Processors. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '14, pages 124–133, 2014.

[18] A. Branković, K. Stavrou, E. Gibert, and A. González. Warm-Up Methodology for HW/SW Co-Designed Processor. In *Proceedings of the International Symposium on Code Generation and Optimization: feedback-directed and runtime optimization*, CGO '14, pages 284–294, 2014.

[19] L. Breiman, J. H. Freidman, R.O. Olshen, and C. J. Stone. *Classification and Regression Trees*. Kluwer Publishers, 1984.

[20] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimizations. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '03, pages 265–275, 2003.

[21] T.E. Carlson, W. Heirman, and L. Eeckhout. Sampled Simulation of Multi-Threaded Applications. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, ISPASS '04, pages 2–12, 2013.

[22] A. Chernoff and R. Hookway. DIGITAL FX!32 Running 32-bit Applications on Alpha NT. In *Proceedings of the USENIX Windows NT Workshop*, 1997.

[23] A. Deb, J. M. Codina, and A. Gonzalez. SoftHV: A HW/SW Co-Designed Processor with Horizontal and Vertical Fusion. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 1:1–1:10, 2011.

[24] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '03, pages 15–24, 2003.

[25] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic Binary Translation and Optimization. *IEEE Transactions on Computers*, 50(6):529–548, 2001.

[26] K. Ebcioglu and E. R. Altman. Daisy: Dynamic Compilation for 100% Architectural Compatibility. In *Proceedings of the 24th annual International Symposium on Computer Architecture*, ISCA '97, pages 26–37, 1997.

[27] L. Eeckhout, Y. Luo, K. De Bosschere, and L. K. John. BLRL: Accurate and Efficient Warmup for Sampled Processor Simulation. *The Computer Journal, vol 48*, pages 451–459, 2005.

[28] A. Falcon, P. Faraboschi, and D. Ortega. Combining Simulation and Virtualization through Dynamic Sampling. In *Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '07, pages 72–83, 2007.

[29] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, 2007.

[30] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-Level Phase Behavior in Java Workloads. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 270–287, 2004.

[31] A. Georges, L. Eeckhout, and D. Buytaert. Java Performance Evaluation through Rigorous Replay Compilation. *SIGPLAN Not.*, 43(10):367–384, 2008.

[32] A. Guha, K. Hazelwood, and M. L. Soffa. Balancing Memory and Performance through Selective Flushing of Software Code Caches. In *Proceedings of the International Symposium on Compilers Architectures and Synthesis for Embedded Systems*, CASES '10, pages 1–10, 2010.

[33] 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. *SIGMETRICS Perform. Eval. Rev.*, 31(4):31–34, March 2004.

[34] J. Haskins and K. Skadron. Minimal Subset Evaluation: Rapid Warm-Up for Simulated Hardware State. In *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, ICCD '01, pages 32–39, 2001.

[35] J. Haskins and K. Skadron. Memory Reference Reuse Latency: Accelerated Warmup for Sampled Microarchitecture Simulation. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '03, pages 195–203, 2003.

[36] M. Hauswirth and A. Diwan. Phases in Branch Targets of Java Programs, 2004.

[37] J. D. Hiser and D. Williams et al. Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, 2007.

[38] S. Hu and J. E. Smith. Reducing Startup Time in Co-Designed Virtual Machines. In *Proceedings of the 33rd annual International Symposium on Computer Architecture*, ISCA '06, pages 277–288, 2006.

[39] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The Garbage Collection Advantage: Improving Program Locality. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 69–80, 2004.

[40] T. Huffmire and T. Sherwood. Wavelet-based Phase Classification. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 95–104, New York, NY, USA, 2006. ACM.

[41] H. Kim and J. E. Smith. Hardware Support for Control Transfers in Code Caches. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 253–264, 2003.

[42] A. Klaiber. The Technology Behind the Crusoe Processors. In *White paper*, January 2000.

[43] S. Kluyskens and L. Eeckhout. Branch History Matching: Branch Predictor Warmup for Sampled Simulation. In *Proceedings of the 2nd International Conference on High Performance Embedded Architectures and compilers*, HiPEAC'07, pages 153–167, 2007.

[44] K. Krewell. Transmeta gets more efficeon. *Micro-processor Report*, 2003.

[45] N. Kumar and N. Neelakantam. Indirect Branches in the Transmeta Efficeon Processor. In *Proceedings of the 2011 Workshop on Infrastructure for Software/Hardware Co-Design*, WISH '11, 2011.

[46] R. Kumar, A. Martínez, and A. González. Speculative Dynamic Vectorization for HW/SW Co-Designed processors. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 459–460, 2012.

[47] R. Kumar, A. Martínez, and A. González. Dynamic Selective Devectorization for Efficient Power Gating of SIMD Units in a HW/SW Co-Designed Environment. In *Proceedings of the 2013 25th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '13, pages 81–88, 2013.

[48] J. Lau, S. Schoemackers, and B. Calder. Structures for Phase Classification. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '04, pages 57–67, 2004.

[49] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N.. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480, New York, NY, USA, 2009. ACM.

[50] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the Programming language design and implementation*, PLDI '05, pages 190–200, 2005.

[51] M. Lupon, E. Gibert, G. Magklis, S. Samudrala, R. Martinez, K. Stavrou, and D. Ditzel. Speculative Hardware/Software Co-Designed FP Multiply-Add. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIX, pages 623–638, 2014.

[52] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[53] P. Nagpurkar and C. Krintz. Phase-based Visualization and Analysis of Java Programs. *Elsevier Science of Computer Programming, Special issue on Princples of programming in Java, volume 59,Number 1-2*, pages 131–164, 2006.

[54] IBM Microelectronics Division Research Triangle Park NC. The PowerPC 440 Core. *White Paper*, 1999.

[55] N. Neelakantam, D. Ditzel, and C. Zilles. A Real System Evaluation of Hardware Atomicity for Software Speculation. In *Proceedings of the fifteenth edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 29–38, 2010.

[56] G. Ottoni, G. Chinya, G. Hoflehner, J. Collins, A. Kumar, E. Schuchman, D. Ditzel, R. Singhal, and H. Wang. AstroLIT: Enabling Simulation-Based Microarchitecture Comparison between Intel and Transmeta Designs. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 21:1–21:2, 2011.

[57] S.J. Patel and S.S. Lumetta. replay: A hardware framework for dynamic optimization. In *EEE Transactions on Computers, 50(6):590-608*, 2001.

[58] D. Pavlou, A. Branković, R. Kumar, M. Gregori, K. Stavrou, E. Gibert, and A. Gonzalez. DARCO: Infrastructure for Research on HW/SW Co-Designed Virtual Machines. In *Proceedings of the 4th Workshop on Architectural and Microarchitectural Support for Binary Translation, held in conjuction with ISCA-38*, AMAS-BT '11, 2011.

[59] D. Pavlou, A. Branković, R. Kumar, K. Stavrou, E. Gibert, and A. Gonzalez. Quantitative Characterization of the Software Layer of a State-Of-The-Art Co-Designed Virtual Machine. *Technical report - UPC Barcelona*, 2012.

[60] D. Pavlou, E. Gibert, F. Latorre, and A. Gonzalez. DDGacc: Boosting Dynamic DDG-based Binary Optimizations through Specialized Hardware Support. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 159–168, 2012.

[61] Tirias Research. NVIDIA Charts Its Own Path to ARMv8. *White Paper*, 2014.

[62] R. Rosner, Y. Almog, M. Moffie, N. Schwartz, and A. Mendelson. PARROT: Power Awareness Through Selective Dynamically Optimized Traces. In *Proceedings of the Third International Conference on Power - Aware Computer Systems*, PACS'03, pages 196–214, 2004.

[63] N. Sachindran and J. E. B. Moss. Mark-copy: Fast Copying GC with Less Space Overhead. In *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 326–343, 2003.

[64] S. Sathaye, P. Ledak, J. Leblanc, S. Kosonocky, M. Gschwind, J. Fritts, A. Bright, E. Altman, and C. Agricola. BOA: Targeting Multi-Gigahertz with Binary Translation. In *Proceedings of the 1999 Workshop on Binary Translation, IEEE Computer Society Tech-nical Committee on Computer Architecture Newsletter*, 1999.

[65] K. Scott and J. Davidson. Strata: A Software Dynamic Translation Infrastructure. Technical report, Charlottesville, VA, USA, 2001.

[66] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 45–57, 2002.

[67] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. The Morgan Kaufmann Series in Computer Architecture and Design. 2005.

[68] S. Sridhar, J. S. Shapiro, and P. P. Bungale. HDTrans: A Low-Overhead Dynamic Translator. *SIGARCH Comput. Archit. News*, pages 135–140, March 2007.

[69] P. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of Java applications. In *Proceedings of the conference on Virtual Machine Research And Technology Symposium - Volume 3*, pages 5–5, 2004.

[70] G. R. Uh, R. Cohn, B. Yadavalli, R. Peri, and R. Ayyagari. Analyzing Dynamic Binary Instrumentation Overhead. In *Proceedings of the Workshop on Binary Instrumentation and Application*, 2006.

[71] C. Wang, Y. Wu, and M. Cintra. Acceldroid: Co-Designed Acceleration of Android Bytecode. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '13, pages 1–10, 2013.

[72] Y. Wu, S. Hu, E. Borin, and C. Wang. A HW/SW Co-Designed Heterogeneous Multi-Core Virtual Machine for Energy-Efficient General Purpose Computing. In *Proceedings of the 2011 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 236–245, 2011.

[73] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical sampling. In *Proceedings of the International Symposium on Computer Architecture*, ISCA '03, pages 84–97, 2003.

[74] T. Y. Yeh, P. Faloutsos, S. J. Patel, and G. Reinmann. ParallAX: An Architecture for Real-Time physics. In *Proceedings of the International Symposium on Computer Architecture*, ISCA '07, pages 232–243, 2007.

[75] J. Yia, S. Kodakara, R. Sendag, D. Lilja, and D. Hawkins. Characterizing and Comparing Prevailing Simulation Techniques. In *Proceedings of the International Symposium on High-Performance Architecture*, HPCA '05, pages 266–277, 2005.