

Divino César Soares Lucas

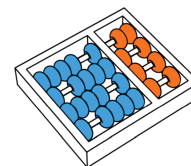
**“Modeling the Performance Impact of Hot Code  
Misprediction in Cross-ISA Virtual Machines”**

*“Modelagem do Impacto de Erros de Predição de Código  
Quente no Desempenho de Máquinas Virtuais”*

**CAMPINAS  
2013**







University of Campinas  
Institute of Computing

*Universidade Estadual de Campinas  
Instituto de Computação*

Divino César Soares Lucas

## “Modeling the Performance Impact of Hot Code Misprediction in Cross-ISA Virtual Machines”

Supervisor:  
*Orientador(a):* Prof. Dr. Guido Costa Souza de Araújo

Co-Supervisor:  
*Co-orientador(a):* Prof. Dr. Edson Borin

### *“Modelagem do Impacto de Erros de Predição de Código Quente no Desempenho de Máquinas Virtuais”*

MSc Dissertation presented to the Post Graduate Program of the Institute of Computing of the University of Campinas to obtain a Master degree in Computer Science.

*Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.*

THIS VOLUME CORRESPONDS TO THE FINAL VERSION OF THE DISSERTATION DEFENDED BY DIVINO CÉSAR SOARES LUCAS, UNDER THE SUPERVISION OF PROF. DR. GUIDO COSTA SOUZA DE ARAÚJO.

ESTE EXEMPLAR CORRESPONDE À VERSÃO FINAL DA DISSERTAÇÃO DEFENDIDA POR DIVINO CÉSAR SOARES LUCAS, SOB ORIENTAÇÃO DE PROF. DR. GUIDO COSTA SOUZA DE ARAÚJO.

---

Supervisor's signature / *Assinatura do Orientador(a)*

CAMPINAS  
2013

Ficha catalográfica  
Universidade Estadual de Campinas  
Biblioteca do Instituto de Matemática, Estatística e Computação Científica  
Ana Regina Machado - CRB 8/5467

L962m Lucas, Divino César Soares, 1985-  
Modelagem do impacto de erros de predição de código quente no desempenho de máquinas virtuais / Divino César Soares Lucas. – Campinas, SP : [s.n.], 2013.

Orientador: Guido Costa Souza de Araújo.

Coorientador: Edson Borin.

Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Sistemas de computação virtual. 2. Compiladores (Programas de Computador). 3. Arquitetura de computador. I. Araújo, Guido Costa Souza de, 1962-. II. Borin, Edson, 1979-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

**Título em inglês:** Modeling the performance impact of hot code misprediction in Cross-ISA virtual machines

**Palavras-chave em inglês:**

Virtual computer systems

Compilers (Computer programs)

Computer architecture

**Área de concentração:** Ciência da Computação

**Titulação:** Mestre em Ciência da Computação

**Banca examinadora:**

Guido Costa Souza de Araújo [Orientador]

Fernando Quintão Magno Pereira

Sandro Rigo

**Data de defesa:** 09-04-2013

**Programa de Pós-Graduação:** Ciência da Computação

## TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 09 de Abril de 2013, pela  
Banca examinadora composta pelos Professores Doutores:



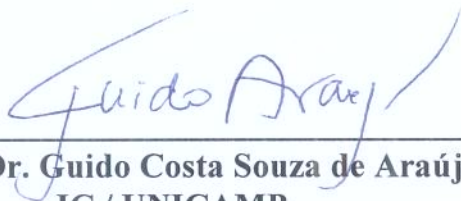
---

**Prof. Dr. Fernando Magno Quintão Pereira**  
DCC / UFMG



---

**Prof. Dr. Sandro Rigo**  
IC / UNICAMP



---

**Prof. Dr. Guido Costa Souza de Araújo**  
IC / UNICAMP



# Modeling the Performance Impact of Hot Code Misprediction in Cross-ISA Virtual Machines

Divino César Soares Lucas<sup>1</sup>

April 09, 2013

## Examiner Board / *Banca Examinadora*:

- Prof. Dr. Guido Costa Souza de Araújo (Supervisor / *Orientador*)
- Prof. Dr. Fernando Magno Quintão Pereira  
Department of Computer Science - The Federal University of Minas Gerais
- Prof. Dr. Sandro Rigo  
Institute of Computing - University of Campinas
- Prof. Dr. Alexandro Baldassin (Suplente)  
Institute of Geociences - Paulista' State University
- Prof. Dr. Rodolfo Jardim de Azevedo (Suplente)  
Institute of Computing - University of Campinas

---

<sup>1</sup>Financial support: Fapesp scholarship (process 2011/05028-5) 2011-2012.



# Abstract

Virtual machines are systems that aim to eliminate the compatibility gap between two, possibly distinct, interfaces, thus enabling them to communicate. This way, acting like a mediator, the VM lies at an important position that enable it to foster innovative solutions for many problems. Such systems usually rely on emulation techniques, such as interpretation and dynamic binary translation, to execute guest application code. In order to select the best emulation technique for each code segment, the VM typically needs to predict whether the cost of compiling the code overcome its future execution time. This problem, in the common case, reduce to predicting if the given code region will be frequently executed or not, a problem called *Hot Code Prediction*. Generally, if the predictor flags a given code region as hot the VM instantly takes the decision to compile it. However, a problem came out from this strategy, the predictor response is only a decision made by means of a heuristic and thus it can be incorrect. Whenever the predictor flags a code region that will be infrequently executed (cold code) as hot code, we say that it is doing a hotness misprediction. Whenever a misprediction happens it means that the technique the VM will use to emulate the code will not have its cost amortized by executing the optimized code and thus the VM will, in fact, spend more time executing its own code rather than the guest application code. In this work we measure the impact of hotness mispredictions in a VM emulating several kinds of applications.

In our analysis we evaluate the *threshold-based hot code predictor*, a technique commonly used to predict hot code fragments. To do so we developed a mathematical model to simulate the behavior of such predictor and we use it to estimate the impact of mispredictions in several benchmarks. We show that this predictor frequently mispredicts the code hotness and as a result the VM emulation performance becomes dominated by miscompilations. Moreover, we show how the threshold choice can affect the number of mispredictions and how this impacts the VM performance. We also show how the compilation, interpretation and steady state execution cost of translated instructions affect the VM performance. At the end we show that using SPEC CPU 2006 benchmarks to measure the performance of a VM using the threshold-based predictor can lead to misleading results.





# Resumo

Máquinas virtuais (MVs) são sistemas que se propõem a eliminar a incompatibilidade entre duas, em geral diferentes, interfaces e dessa forma habilitar a comunicação entre diferentes sistemas. Nesse sentido, atuando como mediadores, uma MV está em um ponto que a permite fomentar o desenvolvimento de soluções inovadoras para vários problemas. Tais sistemas geralmente utilizam técnicas de emulação, por exemplo interpretação ou tradução dinâmica de binários, para executar o código da aplicação cliente. Para determinar qual técnica de emulação é a ideal para um trecho de código geralmente é necessário que a MV empregue algum tipo de predição para determinar se o benefício de compilar o código supera os custos. Este problema, na maioria dos casos, resume-se a predizer se o dado trecho de código será frequentemente executado ou não, problema conhecido pelo nome de *Predição de Código Quente*. Em geral, se o preditor sinalizar um trecho de código como quente, a MV imediatamente toma a decisão de compilá-lo. Contudo, um problema surge nesta estratégia, a resposta do preditor é apenas a decisão de uma heurística e é, portanto, suscetível a erros. Quando o preditor sinaliza como quente um trecho de código que *não* será frequentemente executado, ou seja, um código que de fato é “frio”, ele está fazendo uma predição errônea de código quente. Quando uma predição incorreta é feita, ocorre que a técnica de emulação que a MV utilizará para emular o trecho de código não compensará o seu custo e portanto a MV gastará mais tempo executando o seu próprio código do que o código da aplicação cliente. Neste trabalho, foi avaliado o impacto de predições incorretas de código quente no desempenho de MVs emulando vários tipos de aplicações.

Na análise realizada foi avaliado o preditor de código quente baseado em limiar, uma técnica frequentemente utilizada para identificar regiões de código que serão frequentemente executadas. Para fazer esta análise foi criado um modelo matemático para simular o comportamento de tal preditor e a partir deste modelo uma série de resultados puderam ser explorados. Inicialmente é mostrado que este preditor frequentemente erra a predição e, como consequência, o tempo gasto fazendo compilações torna-se o maior componente do tempo de execução da MV. Também é mostrado como diferentes limiares de predição afetam o número de predições incorretas e qual o impacto disto no desempenho da MV.



Também são apresentados resultados indicando qual o impacto do custo de compilação, tradução e velocidade do código traduzido no desempenho da MV. Por fim é mostrado que utilizando apenas o conjunto de aplicações do SPEC CPU 2006 para avaliar o desempenho de MVs que utilizam o preditor de código quente baseado em limiar pode levar a resultados imprecisos.



# Contents

<b>Abstract</b>	<b>ix</b>
<b>Resumo</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	2
1.2 Contributions . . . . .	4
<b>2 Related Work</b>	<b>6</b>
2.1 Hot Code Prediction Approaches . . . . .	6
2.2 Hot Code Prediction Implementations . . . . .	10
2.3 Virtual Machines Overhead Analysis . . . . .	12
<b>3 Threshold-based Hot Code Misprediction Overhead</b>	<b>14</b>
3.1 The Emulation Cost . . . . .	14
3.2 Hot Code Prediction . . . . .	15
3.3 Hot Code Misprediction Overhead . . . . .	16
3.4 Model Extensions . . . . .	18
<b>4 Methodology</b>	<b>19</b>
4.1 The Model’s Input: Instructions Execution Frequency . . . . .	19
4.2 Model’s Parameters: Interpretation Cost . . . . .	20
4.3 Model’s Parameters: Translation Cost . . . . .	21
<b>5 Results</b>	<b>23</b>
5.1 Benchmark Characterization . . . . .	24
5.1.1 Footprint Characterization . . . . .	24
5.1.2 Prediction Accuracy . . . . .	26
5.1.3 Execution Coverage . . . . .	26
5.2 Estimation of the Model Parameters $\beta_I$ , $\alpha_T$ and $\beta_T$ . . . . .	29



5.3	The Effect of the Model's Parameters in the Emulation Overhead . . . . .	32
5.4	Misprediction Overhead . . . . .	34
5.5	Misprediction Overhead Characterization . . . . .	36
<b>6</b>	<b>Conclusions</b>	<b>39</b>
	<b>Bibliography</b>	<b>40</b>





# List of Tables

1.1	Example of emulation and misprediction overhead . . . . .	3
5.1	Benchmark characterization - footprint . . . . .	25
5.2	Benchmark characterization - prediction accuracy . . . . .	27
5.3	Benchmark characterization - cover set . . . . .	28
5.4	Measured model's parameters for $\beta_I$ , $\alpha_T$ and $\beta_T$ . . . . .	30
5.5	Range of the model's parameters used in experiments . . . . .	32



# List of Figures

3.1	Prediction scenarios for the threshold-based predictor . . . . .	17
5.1	Translation cost for SPEC CPU 2006 benchmarks' functions . . . . .	31
5.2	The effect of model's parameters in misprediction overhead . . . . .	33
5.3	Misprediction and emulation overhead for Sysmark:Office, SPEC-2006:GCC and Windows 7 Boot . . . . .	35
5.4	Impact of misprediction overhead during the VM maturing cycle . . . . .	36
5.5	Maximum and minimum misprediction overhead for SPEC CPU 2006, Sys- mark 2012 and OS Boots . . . . .	37
5.6	The components of the misprediction overhead . . . . .	38



# Chapter 1

## Introduction

Virtual machines (VMs) are systems that act like mediators, enabling the communication between two possible distinct interfaces. In such sense they are like protocols, however, the main goal of a VM [42] is to enable a given guest software to execute on a platform, the **host** environment, other than that which it was originally meant to run on, the **guest** environment. This is accomplished by supporting the guest environment interface using the host environment. There are plenty of virtual execution environments [2, 3, 5, 6, 13, 14, 16, 19, 24, 46, 48], some targeted at emulating a sole application (process VMs) and others at emulating a whole system (system VMs). Besides portability [6, 24, 43], virtual machines can be used for several others purposes like: support to legacy code execution [16, 48], dynamic program optimization [5, 13, 46], program shepherding [35, 38] and dynamic program instrumentation [32].

Virtual machines are very versatile, but they come with an inherent price: the emulation cost. Following the VM taxonomy proposed by Smith and Nair [42], regarding emulation technique, there are two main classes of virtual machines: those that use **interpretation** and those that use **translation**.

Interpretation is the simplest and most direct approach. In this kind of emulation, the VM has routines to emulate the behavior of each instruction in the guest instruction set architecture (ISA). Since every instruction of the guest software triggers an emulation routine, this method is typically slow.

Translation is a more sophisticated approach that aims at improving the emulation performance at the price of the VM portability. In this technique, the source representation of the guest software is translated into code that runs natively on the host ISA. The performance of the translated code is comparable to native execution and is much faster than interpretation. However, the cost to perform the translation is high and thus it is only profitable on code regions that have an execution frequency high enough to offset the translation overhead.

In order to maximize performance, it is necessary to increase the amount of optimized code which is executed, thus it is important to prevent translation of infrequently executed code (**cold code**) and to translate frequently executed code (**hot code**) as soon as possible. To do so, state-of-the-art VMs [5, 16, 24] rely on hot code prediction to select the best emulation technique for each code segment being emulated. The virtual machine is typically provided with a hot code predictor that employs some kind of heuristic to predict whether a given code region will be frequently executed or not. Until the predictor flags a region as hot, the VM uses an emulation technique with low startup overheads, generally interpretation or a quick form of translation, and postpones any code optimization. When some code region is flagged as hot, the VM switches to a second stage. In this stage, the VM constructs a code region (e.g. a trace [5, 6, 24, 46]) containing the hot code and optimize this region by doing an optimizing translation or further optimizing previously translated code. After optimization the code is stored in a code cache for future re-execution.

The aforementioned approach is called a two-stage VM and can be easily extended to work with multiple stages (multiple levels of hotness) depending on the predictor being used. A widely used approach to predict hot code is based on execution frequency thresholds [3, 5, 6, 13, 17, 24, 46, 48]. This predictor is very simple: it flags a region as hot if and only if it reaches a fixed execution frequency threshold, which we will call in this work  $T_P$ . The rationale behind it is: infrequently executed code regions do not reach the threshold. We call **warm code** those program regions that do not satisfies this hypothesis. These are regions that have an accumulated execution frequency sufficiently high to reach the prediction threshold but its final execution frequency is not high enough to compensate for the compiling overhead. When the predictor flags a warm code as hot we call this a **hot code misprediction**. From now on we will call this predictor the threshold-based predictor (or simply TBP), and its behavior and how the mispredictions affect the performance of its corresponding VM are the object of study of this work. In this work we are only concerned with cross-ISA VMs (a cross-ISA VM is one where the guest and host interfaces are different), since same-ISA VMs the translation cost can be negligible because in general only a copy of the guest code is done to “emulate” it. In the following sections we give an example on how mispredictions can affect the VM performance and list the contributions of this work.

## 1.1 Motivation

As discussed before, performing translation or heavy optimizations on cold/warm code may add a significant overhead to the emulation process as their code execution frequency may not be sufficiently high to amortize the translation cost. This is typically the case

when the hot code predictor misses the prediction.

In the following example, we assume a two-stage virtual machine that employs interpretation to emulate cold code and dynamic binary translation to emulate hot code. The translator produces faster emulation code than the interpreter, at the cost of the translation overhead. Therefore, without lack of generality, assume that the VM translator takes 1,000 cycles to translate and one cycle to execute each instruction, while the interpreter takes 50 cycles to emulate each instruction. As shown in Table 1.1, the interpreter of this virtual machine (second column) would take 250 cycles to emulate an instruction that executes 5 times and 10,000 cycles to emulate an instruction that executes 200 times, a total of 10250 cycles. Meanwhile the translator (third column) would take 1005 cycles to emulate an instruction that executes 5 times and 1200 cycles to emulate an instruction that executes 200 times, a total of 2205 cycles and, 4.65 times faster than the interpreter.

Freq.	Interp. Cost	Trans. Cost	Opt. Cost	Pred. Cost, $T_P = 6$
5	$5 \times 50$	$1000 + 5 \times 1$	$5 \times 50$	$5 \times 50$
200	$200 \times 50$	$1000 + 200 \times 1$	$1000 + 200 \times 1$	$(6 \times 50) + (1000 + 194 \times 1)$
<b>Total</b>	10250	2205	1450	1744

Table 1.1: Example of emulation and misprediction overhead.

Despite the fact that translating all the instructions is better than interpreting, a faster approach would be to interpret the instruction that executes 5 times and to translate the instruction that executes 200 times (fourth column). This combination would take a total of 1450 cycles and would be 7.07 times faster than interpreting and 1.5 times faster than translating both instructions. According to this example the best emulation technique depends on the code execution frequency, and combining different techniques may allow us to improve the emulation time of code with different execution frequencies. However, since the execution frequency of instructions is not known beforehand, it is important to predict whether the code will be hot or cold in order to select the best emulation technique for each instruction (or code segment).

If we apply the TBP with threshold  $T_P = 6$  to the previous example (fifth column), the VM would spend 250 ( $5 \times 50$ ) cycles interpreting the instruction that executes 5 times, 1300 cycles interpreting and translating the instruction that executes 200 times and 194 cycles executing the translated code. The total emulation time, 1744 cycles, is 20% higher than the optimal case, in which the cold instruction is interpreted and the hot instruction is translated beforehand. This extra overhead occurs because, in this approach, hot code is emulated as if it was cold code (using interpretation) until it is predicted as hot. In cases where the execution frequency is slightly larger than the threshold, the predictor will mispredict the code as hot, causing extra overhead due to translation or optimizations on cold code. As an example, if  $T_P = 5$ , the total emulation time for the previous example

would be 2695:  $1250 (5 \times 50 + 1000 + 0)$  cycles for the instruction that executes 5 times and  $1445 (5 \times 50 + 1000 + 195)$  cycles for the instruction that executes 200 times. This is 1.86 times slower than the optimal case.

In the next section we present our main contributions in this work.

## 1.2 Contributions

One major contribution of this work is to show that the threshold-predictor’s hypothesis is not sufficiently strong to be used for large code footprint applications, particularly in those that run on interactive environments. Specifically, we show that the **maximum** expected overhead due to mispredictions is no more than 10% when executing applications from SPEC CPU 2006 [22], however, our experiments indicate that applications from Sysmark 2012 [15] are expected to suffer from **more than** 27% of misprediction overhead. As we will see in Chapter 2, this predictor has been used in several ISA VMs: Transmeta CMS [16], HP Dynamo [5], Mojo [13], IA32-EL [6], Star DBT [46], Aries [48] and others.

Other further contributions of this work are the following.

- An analytical model to estimate the misprediction overhead of threshold-based hot code predictors (Chapter 3). To the best of our knowledge this is the first work to analytically model the overhead of such predictors.
- A comprehensive evaluation of the TBP in productivity applications with large code footprint (Chapter 5). We show how using the SPEC CPU2006 benchmarks to measure VM performance can lead to misleading results. We are not aware of other work which has quantified how misleading can be the use of SPEC to measure VM performance.
- We show that as the VM employs more cycles to translate/optimize regions the misprediction overhead sharply increases (Chapter 5). This result, together with the next one, indicates that relying only in optimization effectiveness may not suffice to amortize the translation overhead.
- We show that for large code footprint applications, using TBP may lead to many mispredictions and, in many cases, the optimized code execution is not capable of amortizing the mistranslations overhead.

This text is organized as follows. Chapter 2 discusses related work. Chapter 3 presents the analytical model we use to quantify the hot code misprediction overhead. Chapter 4 presents the methodology we employed to produce our experiments. Chapter 5 shows the overhead results derived from our analysis of hot code misprediction for SPEC CPU



2006, Sysmark 2012 and some operating system boots. Finally, Chapter 6 presents our conclusions.

# Chapter 2

## Related Work

In this chapter we present a review of the literature of virtual machines, focusing on three aspects that are closely related to this work. Initially, we present a review of the current approaches used to address the prediction and/or detection of frequently executed code regions. Next, we present several virtual machines and the hot code detection techniques they use. The third aspect of our review discusses papers that analyze the emulation overhead in virtual machines.

In describing current and past virtual machines we are mainly interested in analyzing the role of the TBP in such VMs. Moreover, we intend to show that many current VMs uses CPU intensive benchmarks, such as SPEC CPU together with the TBP, to report performance results.

### 2.1 Hot Code Prediction Approaches

As we will soon see, there are a few hot code prediction techniques and many of them employ what we call a threshold-based approach to determine if a given code region will be frequently executed or not. As explained in Chapter 1 this approach consist in considering a region as hot only when (and if) a given execution frequency counter for that region reaches a threshold. This concept is usually implemented using execution sampling or instrumentation counters. In the following paragraphs we describe the most relevant research in this area.

The work of Duesterwald and Bala [17] settled the basis for a hot code prediction and region formation technique that would be used by many subsequent virtual machines. In this work they propose MRET (Most Recent Execution Tail) an instrumentation driven integrate approach to predict and construct a hot code region. MRET relies on two assumptions. First, that the target of backward branch instructions are almost always the start of frequently executed regions and second, that given a control flow graph (CFG) [1]

of a region, collecting an arbitrary path of execution in this CFG (during one of its executions) tends to capture the most frequently executed path in that region. Following these assumptions MRET assigns execution counters for every basic block that is target of a backward branch instruction. These counters are incremented every time the basic block is executed and when its value reaches a certain threshold a region formation algorithm is started. The constructed region consists of all basic blocks executed after the threshold has been reached up to the next backward branch instruction (or a maximum number of BBs). The authors in [17] compare MRET with a path-based approach (registers are assigned to paths instead of basic blocks) and show that much less space and instructions are required to store and update counters. As previously stated, this predictor has been used in several virtual machines, however it has some problems and other approaches have been proposed to overcome these problems. In the next paragraphs we describe two problems with MRET and the ways to mitigate such problems.

In some situations, MRET approach to consider the next executed tail (NET) as the most frequently executed path of a CFG does not hold. For example, in a CFG with two paths that alternate execution, choosing only one path will end with a path that executes only half the time. In order to improve the region selection of MRET, Chen et al. [45] proposed MRET<sup>2</sup>. The modifications are very simple. After MRET forms the trace the execution counter is reset and the algorithm wait until the threshold is reached a second time. When the counter reaches the threshold for the second time a new trace is created. The new version considers the trace formed by the intersection of the basic blocks of the two traces as the most frequently executed path. The authors in [45] compare MRET and MRET<sup>2</sup> using the SPEC CPU 2000 benchmarks and show that the path completion rate of MRET<sup>2</sup> is consistently larger than that of MRET.

Another way of implementing the concept of threshold-based hot code prediction is through the use of sampling [10]. By using such approach the system is enabled with a framework that periodically reads hardware registers (e.g. the Program Counter) to collect samples that identify what code regions the program was executing. After a reasonable amount of samples are collected the system employs statistical methods to identify what are the frequently executed code regions. Given the reduced frequency of counter updates and the smaller number of registers needed to store frequency counters. This approach results in less space and execution time overhead when compared to instrumentation-based approaches, However, this approach needs to periodically stop the application execution to apply statistical inferences to determine the most frequently executed regions.

Other techniques for predicting hot code exists but in general they use a threshold-based approach (based on sampling or execution frequency) and differ mainly in the way they propose to create the hot region.

In another effort to reduce the overhead incurred from application profiling, Merten

et al. [34] propose a hardware-driven profiling scheme. The proposed approach, called Branch Behavior Buffer (BBB) is an integrate two-step hot code detection and region formation technique. The proposed technique is based on branch instruction execution frequencies and hardware timers. In BBB a hardware table called branch behavior buffer maintains a summary of all branches executed during a time slice. Branches for which the execution exceeds a threshold are marked as candidate branches. At the end of each monitoring interval the set of candidate branches are further analyzed to see if they meet the requirements needed in order to form a hot code region: 1) be active for a minimum amount of time (a timer based on branch instructions execution is used) and 2) represent a minimum percentage of all branches execution. Once a hot spot has been detected, the hardware triggers a trap to the operating system warning the detection of the new hot spot. The authors performed experiments with several SPEC CPU 95 benchmarks and shows that the system is capable of capturing small code regions (hot spots) that represent a large portion of the dynamic instruction stream. The proposed approach has negligible profiling overhead, however, it lacks support for multiprocess/multicore systems, rely on OS support for region formation and its effectiveness is limited to the BBB size.

Loop regions (roughly speaking the basic blocks between a backward branch and its target) are frequently the source of hot spots due to its iterative nature. However in a given loop region many paths may be exercised across multiple iterations, and thus considering only one path may not be enough to capture all the execution iterations of the loop. To address this problem Baba et al. [4] proposed a hardware based two-level hot path detection approach. The two-level translates to two hardware tables. Initially, “loop” paths are formed using a bit-tracing algorithm. Each time a path is executed a corresponding entry is updated in a Filter Table incrementing the execution frequency of that path. Once the execution frequency of a path in the Filter Table reaches a fixed threshold the path turns to be persisted in a second table called Accumulator Table. This two level organization enables to capture local hot paths - those in the Filter Table for which the execution counter has reached a minimum threshold - and global hot paths - those in the Accumulator Table for which the associated execution counter exceeded a certain threshold. They show results using SPEC CPU 2000 comparing their technique with a one-level approach and a two-level unlimited resources approach. Their results indicate that the technique is superior to a one-level approach and can successfully capture the same top 5 hottest paths that the unlimited resources implementation does captures. Although their two-level approach seems effective at filtering infrequently executed paths, it introduces a further delay in detecting hot paths which consequently increases the missed opportunity cost [17], that is, the cost of interpreting hot regions until they are flagged as hot. Another problems with this approach is caused by indirect branches along the loop paths which may create a huge increase in the number of detected paths, code

duplication and resources trashing [39].

The techniques mentioned so far can be viewed as two-phase approaches. In the first phase the application is profiled to identify hot regions and when these regions are identified they are optimized. As we will see in the next section, most virtual machines use a similar approach if not the same. During its execution a program can go through many phases [18, 40, 41] where the application behavior can change drastically, what makes one questioning if the initial profiling phase used by these hot code predictors are representative of the whole program execution. To answer this question Wu et al. [47] used Intel IA-32 EL [6] to investigate how representative the initial profiling phase are of the entire program execution. They experiment with several execution count thresholds and also compare the profiling phase data with predictions made using the benchmarks training input set. They report results for SPEC CPU 2000 benchmarks. The results indicate that a profiling phase with thresholds ranging from 500 to 2k give results comparable to those of a profile-guided optimization using training input sets. However they noticed that no prediction threshold is good enough for all scenarios and that for several programs, due to phase changes, a single profiling phase does not capture the average program behavior accurately.

All the prediction techniques just described acts during the program execution and so they employ “simple” heuristics to make predictions. If we allow the predictions to be made offline we can use much more sophisticated techniques. In the next paragraphs we will comment some of these techniques, they are product of more recent research in hot code prediction and are a rupture with the ad-hoc approach based in thresholds.

Buse and Weimer [9] propose an approach based on Logistic Regression to identify frequently executed paths in a program. They claim that the program source code contains enough information to distinguish those paths that have greater probability to be executed from those that will be infrequently executed. Their technique transform the problem of detecting hot paths in a classification problem. Statically all paths are obtained and an information vector (IV) is generated for each path. This information vector contains static characteristics of each path including number of branch instructions, variable assignments, object allocations, throw statements, total number of instructions, and many others. After all paths are enumerated and the IV are collected the paths are feed to a logistic regression which return the probability of each path being in the set of highly executed paths. Given the probability of the path being hot, it is possible to determine if a given path is indeed hot (e.g. above a threshold) and what is the relative frequency among the paths. They evaluate their technique for SPEC JVM 98 using the *F-score* [11] measurement to quantify the “precision” which the technique identifies hot paths. Their experimental results report that around 86% of the hot paths are correctly classified. They also use Kendall’s tau [20] distance metric between ranked lists to measure how well is the relative frequency ordering

produced by their technique when compared with the real execution count ordering. The results indicate the technique has a Kendall Tau Distance of 0.25 which means both ordering are strongly correlated.

The work of Johnson and Valli [27] is very similar to the just described work of Buse and Weimer. As the aforementioned work, Johnson and Valli proposed an approach that uses static properties of the program source code to estimate the execution frequency of program hot spots. The proposed approach differ significantly in three aspects: 1) it uses Support Vector Machine as the classifier, 2) The code region considered is methods/functions and 3) the number and type of the static properties. This new approach rely solely on ten static features (e.g. number of loops, number of call instructions, number of call sites for the function, number of basic blocks in the function, etc) while the previous approach relies on more than 20. One disadvantage of this approach is that the inference machine used is only a binary classifier (outputs hot or cold) while Buse and Weimer approach returns probabilities and thus can be used to derive relative execution frequencies. The work is evaluated using SPEC-INT CPU 2000 and UTDSP [30] benchmarks. They report the precision (percentage of correct classified functions) for two sets: the hot functions set and the total set (hot and cold functions) for both benchmark suites. The results indicate that the method is capable, on average, to predict correctly 59% and 84% of the hot methods from SPEC and UTDSP benchmarks, respectively. For the total set the numbers are 65% and 70% respectively for SPEC-INT CPU 2000 and UTDSP.

Liu and Zhang [31] modeled the hot path prediction problem as a geometrical problem and used symbolic execution [28] and constraint solving techniques [33] to estimate path execution frequency. Their technique, like those of Johnson and Valli and of Buse and Weimer, initially enumerate all paths in the program - in general paths' scope are limited to a class or method. However, different from these previous approaches this new technique does not use static properties of the paths to predict execution frequency, instead they model each path with a set of constraints and uses constraint solving techniques to estimate the path execution frequency. Specifically, they use symbolic execution to identify a set of constraints that represent the conditions upon which a given path would execute. Given a path, the set of constraints for that path can be used to model a geometrical figure in a n-dimensional space. The work proposes using constraint solving techniques to estimate the volume of this figure, and to use this value as an estimative for the execution frequency of the given path.

## 2.2 Hot Code Prediction Implementations

There are many works describing the design, implementation and applicability of virtual execution environments [2,3,5,6,13,14,16,19,24,46,48]. In common, all these systems have

in common the fact that they do employ some sort of selective compilation to amortize the emulation overhead. In many cases, they combine interpretation and compilation and others involve a multilevel optimization approach.

The FX!32 [24] is a virtual machine that enables transparent execution of 32-bit x86 Windows NT applications on Alpha hosts running Windows NT. FX!32 first interprets the guest application code regions at the same time that inserts code to gather profile information. The next time the code is invoked the system uses the profile information to generate an equivalent Alpha binary code. The system was tested on a 500MHz Alpha machine running the BYTEMagazine benchmark suite, matching the performance of the same benchmark running on a 200MHz x86 machine. Since the translation is performed offline the system does not suffer from hotness misprediction overhead.

Bala *et. al* [5] describe the design and implementation of Dynamo, a software dynamic optimization system that is capable of transparently improving the performance of a HP PA-800 instruction stream as it executes on the processor. To attain this, Dynamo interprets the guest application code until a start-of-trace criteria is met at which point it employs Most Recent Executed Tail (MRET) [17] to form a trace, optimize and persist it for use in future executions. They evaluate the performance of the system with the SpecInt95 benchmark suite and show that Dynamo can leverage the execution time of a binary compiled with the HP production compiler at -O0 to that of a binary compiled with the same compiler at level -O4.

The IA-32 Execution Layer (IA-32 EL) [6] is a dynamic binary translator that enables the execution of IA-32 applications on Intel Itanium processor systems. The system employs a two-phase translation approach. Initially, the application code is translated on a basic-block basis using a minimal set of optimizations, and instrumentation code is used to detect hot spots. When the instrumentation counter of a basic block reaches a prefixed threshold it is marked as a candidate hot spot. After many basic blocks are marked as candidates the system use edge profiling information to form an instruction superblock that will be further optimized and cached. They measured the system performance using the Spec CPU 2000 benchmarks and showed that they can reach 65% of native execution performance.

StarDBT [46] is a multi-platform research binary translator capable of translating x86 32/64 bits applications to IA 32 bits binaries at a performance comparable to native execution. StarDBT uses a simple fast translator for cold code translation and once a workload hot spot is detected, it forms a trace around the code, applies optimizations and caches the trace. They evaluate the system using the SPEC CPU 2000 and Sysmark 2004 suites. Results show that the system runs comparatively well when compared to other state-of-the-art binary translators, however for large interactive Windows applications the overhead can be considerably high. The authors argue that optimizing infrequently



executed code regions causes the overhead.

As it is apparent from the above mentioned works, the use of a two-phase approach to binary translation and the SPEC benchmark to measure system performance is frequently employed. In this work we advance one step further on DBT overhead characterization by showing that such strategy can lead to misleading results. The SPEC benchmark, particularly SPEC CPU 2006, is a CPU intensive benchmark where the execution time of applications is dominated by just a few frequently executed hot spots. As we show below the use of such suite to measure DBT performance can produce misleading results because all code regions predicted as hot will be recurrently executed. However, for large code footprint applications, such as those used in interactive environments, the profile generated in the first phase can contain lots of false-positive hot spots - that is, code regions that were routinely executed in the first phase but will soon become rarely executed.

If we consider VM for high level languages [2,37], several other strategies for translation emerge, such as multi level optimization and many compilation threads. However, the Jikes RVM [2] uses a cost benefit model similar to the one we present in Chapter 3 to determine which level of compilation to apply to a given method. The model we propose differs from the one in Jikes fundamentally because our model is used to assess the misprediction overhead of the TBP. This model includes aspects not present in the Jikes model (e.g. the oracle predictor) and also we do not have any assumption regarding instructions execution frequency.

## 2.3 Virtual Machines Overhead Analysis

In this section we review several papers which focus on characterizing and mitigating the emulation overhead in virtual machines. Despite the large number of works using and proposing virtual machines, just a few papers [8, 12, 25, 47] are specifically focused on characterizing the overhead of such systems.

Borin and Wu [8] study the overhead of the Intel research dynamic binary translator StarDBT when emulating the SPEC CPU 2000 benchmarks [21]. They break the DBT functionality in five main operations: initialization - time spent loading the DBT, cold code translation - time spent translating code before its first execution, code profiling - time spent instrumenting and collecting profile information, hot trace building - hot trace construction using MRET<sup>2</sup>, and translated code execution - time spent executing the application binary. To measure the overhead of each of these aspects they develop new versions of the DBT tailored specifically to measure the overhead of each component. After a detailed analysis, using the SPEC CPU 2000 benchmarks, they found that return instructions handling (due to translation of RET instructions) and code duplication (due



aggressive trace formation and function inlining) represent more than 64% of the total StarDBT overhead. Their results also show that cold code translation and hot trace building together account for 34% of the DBT overhead.

Hu and Smith [25] use a Co-Designed virtual machine to study the overhead of an adaptive dynamic binary translation system. They use a two-phase DBT that performs simple basic block translation to initial emulation, and a superblock optimizer for emulating hot spot code - detected when the execution frequency reaches a prefixed threshold. Given their DBT characteristic (two-phase translation) they model the emulation cost in terms of the cost to do the initial translation of basic blocks and the optimization cost of the eventually detected hot spots. After applying their model to the Winstone 2004 benchmarks [44] they show that, in the environment under consideration, the initial translation of cold code is the major component of the emulation overhead. They propose two solutions to speed up cold code emulation, both hardware assisted. First is a modification of the decoding unit, by turning it to a dual mode decoder. In one of the modes it decodes the native instruction stream (guest application code, x86 in this case) and in other mode it decodes the internal co-designed VM representation (the VM code). The Virtual Machine Monitor (VMM) is responsible for switching between the two modes. The second approach aims at adding a new instruction to the host architecture. This new instruction would receive two pointers, one to the guest x86 instruction to be decoded and the other a pointer to where the decoder output is stored. After applying the strategy to the co-designed baseline virtual machine they show that the VM system startup performance is significantly improved.

Following the same direction as Hu and Smith research is the work of Chen et al. [12]. They use a binary translation simulator to characterize the overhead of the SPEC2000 integer benchmark suite and show that interpretation is responsible for over 42% of the overhead of the two-phase DBT simulated, which is in resonance with the result of Hu and Smith. To mitigate the problem they propose the utilization of a Decoded Instruction Cache (DICache). The proposed structure for the DICache is similar to an in hardware L1 cache, it contains a tag field, a field pointing to the interpretation routine and other two fields to store operand information. They simulated the use of various DICache configurations and the hit rate averaged at 98.70% reaching an speedup of 1.94x when running several SPEC CPU 2000 benchmarks.

# Chapter 3

## Threshold-based Hot Code Misprediction Overhead

In this chapter we formalize several aspects related to hot code prediction. Initially, we define cost functions that estimate interpretation and translation costs associated with the emulation of an arbitrary instruction in an abstract VM. Subsequently, we introduce the concept of an oracle hot code predictor and present a formalization of a TBP. Finally, we show all possible scenarios that may happen when using a TBP and propose a mathematical model to estimate the hot code misprediction overhead when using such predictor.

### 3.1 The Emulation Cost

We can estimate the cost to interpret an instruction  $I$  that executes  $n$  times using the following linear equation on  $n$ :

$$C_I(n) = \alpha_I + \beta_I n \quad (3.1)$$

where  $\alpha_I$  is the cost of any necessary preprocessing (e.g. pre-decoding) required to execute  $I$ . Once preprocessed, the interpreter takes  $\beta_I$  cycles to emulate the instruction every time it is executed. If no pre-decoding techniques are used,  $\alpha_I$  is equal to zero.

Similarly, we can estimate the cost to emulate an instruction  $I$  with dynamic binary translation using the following equation.

$$C_T(n) = \alpha_T + \beta_T n \quad (3.2)$$

where constant  $\alpha_T$  represents the non-recurrent cost to translate (compile), optimize and cache instruction  $I$ , and constant  $\beta_T$  is the cost paid each time the translated code

is executed to emulate the instruction  $I$ .

As we will see in Chapter 4, the parameters  $\alpha_I$ ,  $\beta_I$ ,  $\alpha_T$  and  $\beta_T$  are not constants along all benchmarks. However through the rest of this text, except otherwise noted, we will use average values for them.

One of the goals of a VM designer is to apply the most cost effective emulation technique for each code region. A common approach to achieve this is to use a two-phase strategy. In the first phase, the system uses a low-overhead startup technique (interpretation) but as soon as the code is predicted as hot it switches to a low-overhead steady-state technique, i.e. binary translation. Inequality 3.3 defines the point where translation have a lower cost than interpretation:

$$\begin{aligned}
 C_T(n) &< C_I(n) \\
 \alpha_T + \beta_T n &< \alpha_I + \beta_I n \\
 \beta_T n - \beta_I n &< \alpha_I - \alpha_T \quad (-1) \\
 \beta_I n - \beta_T n &> \alpha_T - \alpha_I \\
 n(\beta_I - \beta_T) &> \alpha_T - \alpha_I \\
 n &> \frac{\alpha_T - \alpha_I}{\beta_I - \beta_T}
 \end{aligned} \tag{3.3}$$

When the total execution frequency of the instruction is greater than  $\frac{\alpha_T - \alpha_I}{\beta_I - \beta_T}$ , it is better to emulate it with dynamic binary translation, rather than using interpretation. Instructions whose final execution frequency does not reach this point should be instead interpreted. We use  $T_H$  as an equivalent for  $\frac{\alpha_T - \alpha_I}{\beta_I - \beta_T} + 1$ , that is, the minimum number of times an instruction should execute to amortize its compilation cost.

Note that inequality 3.3 requires that we know in advance how many times each instruction will be executed in order to determine the best technique to emulate them. Therefore, it cannot be used by a VM monitor to choose one particular method prior to emulation, but only to assess performance losses after emulation. In order to choose which technique to use for emulation, as pointed out earlier, a mechanism that predicts whether a given instruction will be frequently executed is used. We discuss such predictors in the next section.

## 3.2 Hot Code Prediction

In order to define a baseline and exemplify what would be a perfect predictor we define an *oracle predictor*, i.e. a predictor that, before any execution of an instruction  $I$ , knows whether it is better to translate or to always interpret  $I$ . The *behavior* of such predictor can be formalized as:

$$Pred_{Ora}(I) = \begin{cases} \text{translate} & \text{if } I_n > T_H \\ \text{interpret} & \text{if } I_n \leq T_H \end{cases}$$

where  $I_n$  represents the instruction **final** execution frequency, and  $T_H$ , as stated before, is the execution frequency for which translation is cheaper than interpretation. As the oracle predictor knows *a priori* the instruction final execution frequency, the cost of emulating the instruction using this predictor will be the cost of always interpreting the instruction or the cost to translate and always execute the translated code. Thus the cost of emulating an instruction  $I$  using this predictor can be formulated as:

$$Cost_{Ora}(I) = \begin{cases} C_T(I_n) & \text{if } I_n > T_H \\ C_I(I_n) & \text{if } I_n \leq T_H \end{cases}$$

We can express the behavior of the TBP formally in terms of the following formula:

$$Pred_{Thr}(I) = \begin{cases} \text{interpret} & \text{if } I_n < T_P \\ \text{translate} & \text{if } I_n \geq T_P \end{cases}$$

where  $T_P$  is the prediction threshold and  $I_n$  means the **current** execution frequency of  $I$ . When using the TBP, we do not know anything about the instruction final execution frequency, thus every time the instruction is executed we must consult the predictor. When the code is flagged hot, we pay the cost to translate it, but until that happens (if it indeed happens) we are paying the interpretation cost. Thus the threshold-predictor cost to emulate an instruction  $I$  that have final execution frequency  $I_n$  is given by:

$$Cost_{Thr}(I) = \begin{cases} C_I(I_n) & \text{if } I_n < T_P \\ C_I(T_P) + C_T(I_n - T_P) & \text{if } I_n \geq T_P \end{cases}$$

In the next section we discuss all possible scenarios that can happen when using the TBP and how the cost of such predictor compares to the oracle cost.

### 3.3 Hot Code Misprediction Overhead

Figure 3.1 shows all six possible cases, based on the range of values the final instruction frequency  $n$  can assume (the horizontal black bar), when using the TBP. We model the misprediction overhead for each one of these cases below when compared to a perfect oracle predictor.

Notice that, depending on the values assigned to  $\alpha_T, \beta_T$ , and  $\alpha_I, \beta_I$ , the value computed for  $T_H$  (from equation 3.3) may become greater or smaller than  $T_P$ . Cases 1(a-c) cover the scenarios for which  $T_H$  is greater or equal than  $T_P$ , and cases 2(a-c) cover the scenarios for which  $T_H$  is smaller than  $T_P$ .

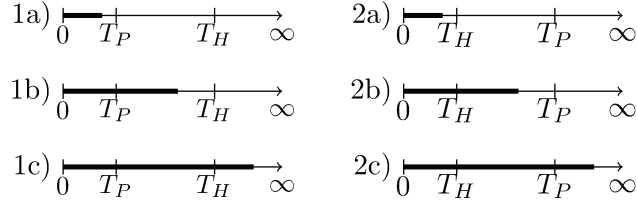


Figure 3.1: Prediction scenarios for the TBP. Black bar shows final instruction frequency.

**Case 1a:** No misprediction happens. The code is not flagged as hot and it is indeed cold.

**Case 1b:** Code is mispredicted as hot. Code is marked as frequently executed ( $n > T_P$ ), but the frequency of the instruction is smaller than  $T_H$  (thus it is cold). The overhead is calculated as follows:

$$Overhead_{1b} = (C_I(T_P) + C_T(n - T_P)) - C_I(n)$$

**Case 1c:** The prediction is correct, but during runtime, until the instruction frequency reaches  $T_P$  it will be mistakenly flagged as cold. Therefore, there is a cost incurred due to the delay for the correct prediction, as follows:

$$Overhead_{1c} = (C_I(T_P) + C_T(n - T_P)) - C_T(n)$$

**Case 2a:** No misprediction happens. The code is not flagged as hot and it is indeed cold.

**Case 2b:** Code is mispredicted as cold. The misprediction overhead is calculated as follows:

$$Overhead_{2b} = C_I(n) - C_T(n)$$

**Case 2c:** The prediction is correct, but during runtime, it was incorrectly predicted as cold for all values  $n < T_P$  before reaching the correct prediction. Therefore, there is a penalty calculated as follows:

$$Overhead_{2c} = (C_I(T_P) + C_T(n - T_P)) - C_T(n)$$

There are four scenarios in which the TBP may incur a misprediction cost. Notice that these equations are just another way to formulate the misprediction cost of the TBP

in relation to an oracle predictor. If we group the costs of all scenarios we can obtain the same result using  $Cost_{Thr}(I) - Cost_{Ora}(I)$ .

For a given benchmark  $B$ , we can sum the cost per instruction when using the TBP and calculate the total overhead in relation to the oracle-predictor using the following formula:

$$Overhead_{total} = \frac{\sum_{I \in B} Cost_{Thr}(I)}{\sum_{I \in B} Cost_{Ora}(I)} \quad (3.4)$$

We have used this model to characterize the overhead of several SPEC2006 [22], Sysmark 2012 [15] benchmarks and operating systems boot processes. In the next chapter we describe the methodology we used to estimate the parameters ( $\alpha_I$ ,  $\beta_I$ ,  $\alpha_T$  and  $\beta_T$ ) based on this model.

### 3.4 Model Extensions

Although it is known that some VM employ a multi gear and/or multi threaded compilation strategy, the overhead model just presented consider only one optimization gear and compilation thread. However note that such multi-gear/threaded virtual machines are more common for high level languages (e.g: Java [2]) and are an exception for ISA level virtual machines, which are the focus of this work. Nevertheless the model could be extended to consider multiple levels of optimization and multiple compilation threads, however we let this extension as a future work.

Another subtlety in our model is the choice of the granularity of the selected code region. We chose the instruction level granularity as a mean to abstract away the details of the region formation technique employed by the VM. But again, the model could be modified to consider a specific code region.

# Chapter 4

## Methodology

The input data of our analytical model is the execution frequency  $n$  for each instruction executed in the benchmark together with a set of parameters to specify the emulation costs.

To determine reasonable values for the model parameters,  $\alpha_I$ ,  $\beta_I$ ,  $\alpha_T$ , and  $\beta_T$  (equations 3.1 and 3.2), we used the Bochs emulator (version 2.5.1) [43] and the Low Level Virtual Machine (LLVM) (version 3.0) [29]. The Bochs x86 system emulator was used to collect the instructions' execution frequencies of the benchmarks and to estimate the parameters  $\alpha_I$  and  $\beta_I$ . LLVM was used to perform the compilation of several code fragments from which we estimated the translation cost of an instruction,  $\alpha_T$ . Finally, the translated instruction emulation performance,  $\beta_T$ , was estimated by running compiled code generated by LLVM.

Notice that we are not the only group that uses a combination of tools to start up our VM environment. HQEMU [23] and Harmonia [36] are examples of modern VMs that use QEMU [7] together with LLVM [29] to build up a VM infrastructure. Therefore, we expect the values we measure for translation to be very close to those of such systems, although the overhead model does not apply to them directly - as they employ multi threaded/geared compilation and do not use interpretation.

### 4.1 The Model's Input: Instructions Execution Frequency

We used the Bochs infrastructure to profile the execution frequency for instructions of the SPEC CPU2006 [22], Sysmark 2012 [15], and the boot process of Windows 7, Windows XP, and Debian 5 Linux. This information was used as input to our model, enabling us to study in details the impact of the misprediction overhead on each of those benchmarks.

We built a Bochs instrumentation code to collect the instruction execution frequency, where each instruction is identified using a combination of its linear address plus the first byte of its instruction opcode. This approach was used because only a given linear address is not enough to identify an instruction instance, as the guest operating system uses the same range of addresses for several processes. Notice that Bochs is a system VM capable of running a full guest operating system.

The Sysmark 2012 suite is composed of a set of scenarios, each one intended to simulate a typical PC-user session, ranging from photo edition to web design and system management. In each scenario many applications may execute concurrently.

Notice that Sysmark frequently restarts the computer in order to assure the system is in a known and stable state. Therefore, if we collect the instruction execution frequency during the execution of the entire scenario we also collect unwanted profile information from the operating system boot and idle periods within the execution of the scenarios. To address this problem, we collect profile information for each interval of 100 million executed instructions and attach to each profile a screenshot from the Bochs virtual screen. After the session is finished, we visually inspect the screenshots and group all intervals that actually belong to a Sysmark scenario.

**Reproducibility:** It is tricky to repeat the execution profile of an entire system. For the experiments that employed Bochs, we configured it to execute in a deterministic way. This was accomplished by using a volatile hard disk, in which all changes to the disk are discharged as soon as Bochs quits executing, and by carefully configuring the clock system, to prevent synchronization with the host system and to enforce the guest system to always boot with the same time and date. The emulated virtual machines were configured with 2GB of RAM and a virtual hard disk of 15GB. Sysmark benchmarks used Windows 7 and SPEC CPU2006 used Debian Linux.

## 4.2 Model’s Parameters: Interpretation Cost

**Estimating interpretation start-up cost  $\alpha_I$  and steady-state cost  $\beta_I$ :** Bochs is known for its high portability and mature code base, an ideal candidate for characterization of a virtual machine that uses interpretation as its emulation technique.

To measure the cost of instruction pre-decoding and interpretation,  $\alpha_I$  and  $\beta_I$ , we changed Bochs to report the number of instructions pre-decoded, the number of instructions interpreted and the total amount of cycles spent in the pre-decoder and interpreter routines. The number of host machine cycles spent in emulation was obtained with the help of Intel Core2 hardware performance counters via the RDTSC instruction [26]. The ratio of the number of *x86 cycles* spent in pre-decoder routines over the number of *pre-*



*decoded instructions* gives  $\alpha_I$ . The ratio of the number of *x86 cycles* spent in the interpreter routines over the number of *interpreted instructions* gives  $\beta_I$ . Each benchmark yields a different  $\alpha_I$  and  $\beta_I$ , providing a range of reasonable  $\alpha_I$  and  $\beta_I$  values for our model.

### 4.3 Model’s Parameters: Translation Cost

LLVM was chosen to estimate the translation cost parameters because it can be seen as a powerful VM that translates LLVM bit-codes into host binaries. LLVM bitcodes are a low-level program representation that is close to machine instructions and has its own ISA, the LLVM virtual ISA [29]. Therefore, it is a good candidate to estimate dynamic translation and optimization costs between different ISAs.

All SPEC CPU2006 benchmarks, with exception of benchmarks written in Fortran, were compiled to LLVM bitcode. Fortran is still not supported by the LLVM frontend. Two programs had its execution time measured. The first one was LLVM opt, responsible for reading an input LLVM bitcode, transforming the code using target independent optimizations and outputting optimized bitcode. The latter was the LLVM llc, the LLVM compiler backend that converts LLVM bitcode to x86 assembly language. This is usually the process a VM needs to perform to translate code using the source ISA to the target ISA. In this scenario, the source ISA is the LLVM bitcode and the target ISA is x86.

We do not perform these experiments for Sysmark 2012 and Boot processes since we do not have the source code for the benchmarks or because LLVM does not have support to compile the source.

**Estimating translation start-up cost  $\alpha_T$ :** We collected data for simulating two scenarios. First we measure the number of cycles needed to perform a crude compilation without applying any kind of optimization, as in a basic binary translation process between two different ISAs. Second, we show data representing another scenario of a virtual machine capable of applying several expensive optimizations, as an estimate of the overhead incurred in time-consuming JIT engines.

We use an auxiliary program also available in the LLVM suite, the LLVM extract, to separate a single function from the rest of a LLVM bitcode file. After generating a LLVM bitcode file for each one of the 71261 functions of all the selected SPEC CPU2006 benchmarks, we run all llc passes that are activated by using the “-O0” flag in opt command line, to collect data for the first scenario. To collect data for the second scenario we run all opt passes that are activated by the “-O2” flag.

Nevertheless, when measuring the cost of compiling each function, care must be taken to eliminate the time spent loading the LLVM tools into memory, initializing, and finalizing them, as this overhead is usually absent from a virtual machine system already loaded

into memory. We did not count this overhead because we timed each pass separately instead of measuring the whole run time of the compilation process. Also, the llc assembly printer pass was not counted because this pass is used to write the x86 code in memory to an assembly text form, a useless task in JIT. The bytecode writer pass on the LLVM opt program was also not counted for the same reason.

The ratio  $\alpha_T$  is then estimated by the number of *x86 cycles* required to compile a function over the number of *LLVM instructions* in this function. Each function has a different  $\alpha_T$ , providing a range of reasonable  $\alpha_T$  values for our model.

**Estimating translation steady-state cost  $\beta_T$ :** To estimate  $\beta_T$  (the number of host cycles spent per guest instruction to emulate the source program after binary translation), we measured the number of *LLVM instructions* executed by the selected SPEC programs using the SPEC reference input and also the number of *x86 cycles* needed to run SPEC x86 native programs using the same inputs. The  $\beta_T$  parameter is then estimated by the ratio of *x86 cycles* over the number of *LLVM instructions*. Each benchmark yields a different  $\beta_T$ , providing a range of reasonable  $\beta_T$  values for our model.

**Variability of  $\beta_T$ :**  $\beta_T$  can change depending on the optimizations used to generate the LLVM bytecode guest executable and the x86 native executable.

The more optimized is the guest program, the higher is  $\beta_T$  (lower performance gain with translation). This simulates the scenario in which a VM translates guest binaries that are already optimized. In this case, there is little performance gain by applying dynamic binary optimization, since most optimization opportunities were already explored. This rises  $\beta_T$  in comparison with a VM translating an unoptimized guest code with plenty of optimization opportunities.

The higher is the level of optimization used to generate the x86 native version, the lower is  $\beta_T$  (better performance gain with translation). This simulates the scenario in which a powerful dynamic binary translation and optimization engine is used to translate guest into native code.

We measured  $\beta_T$  using an optimized LLVM bytecode (“-O2”) as guest binary because, in general, programs are already optimized to a certain degree, illustrating a common situation for VMs. To generate the native binary, we used no optimizations. This simulates the scenario of VMs that are unable to apply optimizations when performing just-in-time compilation.

# Chapter 5

## Results

In this chapter we present several results we gathered from applying the aforementioned overhead model to three sets of benchmarks: SPEC CPU 2006, Sysmark 2012 and Linux/Windows boot processes. Initially, in section 5.1, we characterize the dynamic behavior of the three sets of benchmarks. Next, in section 5.2, we show the results achieved when using Bochs and LLVM to estimate the model’s parameters - these values are used in the remaining experiments. In section 5.3 we show how each model’s parameters individually affect the misprediction overhead. Following in the same line as section 5.3, section 5.4 shows how interaction between model’s parameters affect the emulation and misprediction overhead. We also show in section 5.4 how the VM performance is affected by hotness misprediction during its maturing cycle - these results helps the VM designer to understand which characteristic of the VM (translation speed, translated code quality) influence most the misprediction overhead. Finally, in section 5.5 we fix a reasonable VM configuration and quantify what would be the misprediction overhead of such virtual machine. Moreover, we also show how different values of the prediction threshold affect the VM performance and what is the major component of the VM overhead. These results illustrate how using SPEC CPU2006 benchmarks to measure VM performance can lead to misleading results and also how the VM optimizations effectiveness are limited by the miscompilations overhead, for large code footprint applications.

During the experiments three Sysmark 2012 scenarios did not complete their execution and we preferred to omit their partial results. Also, for all experiments in this chapter we used a zero pre-decoding cost given that in our results, it showed negligible impact on the misprediction overhead.

Although we used Bochs and LLVM to measure interpretation and translation costs, in fact we use these values only to determine a range of values to be considered in our experiments, see Table 5.5, and not to model a virtual machine built on basis of Bochs and LLVM. Therefore the results we show in this chapter were not specifically designed

to model a specific VM, but rather to provide insights on how the misprediction overhead can affect the performance of an arbitrary VM. The range of values we use to gather these results totalize over 125000 configurations, so we do expect that these configurations cover a large extent of all DBT design space. Please note, that even if the DBT costs ( $\alpha_T$  and  $\beta_T$ ) are out of this range, it is reasonable to expect that the trends shown in the graphs will not change.

## 5.1 Benchmark Characterization

### 5.1.1 Footprint Characterization

Table 5.1 shows the static and dynamic foot print of the three sets of benchmarks we studied. The second column of the table shows the number of static instructions touched (in terms of the number of entries in the profiling hash) <sup>1</sup> during the execution of the benchmark; The third column shows the number of executed instructions; The fourth column shows the average instruction reuse, that is the average number of times a static instruction is executed (Dynamic / Static).

In general, the three sets of benchmarks have well distinctive results. SPEC benchmarks have, on average, the smaller static code footprint however they show the largest dynamic footprint, specifically for floating point benchmarks (those below - including - GemsFDTD). As a consequence, SPEC is the suite with the greatest reuse rate, two and three orders of magnitude greater than Sysmark and Boot, respectively. These numbers show that SPEC benchmarks have a few kernels that are intensively executed.

Sysmark static code footprint is on average the largest among the three suites. However its dynamic footprint is around the half of the SPEC. This means that Sysmark has a “sparse” set of benchmarks, that is it exercises a large portion of code, but they execute only a few times, as it can be seen from its reuse rate.

The Boot processes have a relatively short static footprint when compared with Sysmark benchmarks. However, they have a footprint almost twice the average size of SPEC’s footprint. The average Boot dynamic footprint is the smallest between all suites, this was expected since they execute for just a few moments. These results represent the very nature of boot processes, which start many services that just do setup configurations and yield.

---

<sup>1</sup>Remember that the hash key are composed by instruction virtual address plus its opcode.

<b>Benchmark</b>	<b>Static</b>	<b>Dynamic</b>	<b>Reuse</b>
Boot-Debian5	5,647,259	8,430,140,502	1,492
Boot-Win7	2,877,855	8,761,034,864	3,044
Boot-WinXP	2,053,038	3,482,653,943	1,696
<b>Average:</b>	3,526,051	6,891,276,436	2,077
Sysmark-Dfa	11,859,140	928,600,000,000	78,302
Sysmark-Office	40,061,419	2,076,000,000,000	51,820
Sysmark-SM	6,435,727	659,800,000,000	102,521
<b>Average:</b>	19,452,095	1,221,466,666,667	77,548
Perlbench	565,866	1,174,498,915,877	2,075,577
Bzip2	329,669	622,138,498,199	1,887,161
Gcc	446,808	139,680,385,253	312,618
Mcf	87,515	397,198,364,446	814,740
Gobmk	422,433	632,742,235,219	1,497,852
Hmmer	260,448	1,070,579,320,160	4,110,530
Sjeng	2,938,619	2,522,292,164,736	858,325
Libquantum	3,251,529	3,019,153,761,811	928,533
H264ref	589,358	582,575,758,631	988,492
Omnetpp	863,018	777,865,080,056	901,331
Astar	229,132	830,206,756,540	3,623,268
Xalancbmk	642,284	1,227,732,404,699	1,911,510
GemsFDTD	2,906,935	2,365,916,358,814	813,886
Bwaves	3,181,906	2,812,226,149,567	883,818
CactusADM	4,087,989	3,010,658,688,745	736,464
Calculix	3,078,574	8,407,849,901,883	2,731,085
DealII	878,394	2,324,156,407,892	2,645,915
Gamess	3,785,573	3,759,850,605,357	993,205
Gromacs	3,726,997	2,287,675,495,659	613,812
Lbm	4,023,221	3,521,533,338,229	875,301
Leslie3d	2,513,050	1,464,027,332,954	582,569
Milc	1,639,467	1,431,939,793,309	873,417
Namd	401,790	2,889,914,269,920	7,192,598
Povray	1,218,271	1,251,910,800,210	1,027,612
Soplex	1,720,213	552,549,201,778	321,209
Sphinx3	977,453	2,891,495,518,759	2,958,193
Tonto	3,280,610	2,974,540,011,001	906,703
Zeusmp	3,760,594	2,530,006,867,800	672,767
<b>Average:</b>	1,850,276	2,052,604,085,268	1,597,803

Table 5.1: Benchmark Characterization - static and dynamic footprint size and code reuse

### 5.1.2 Prediction Accuracy

Table 5.2 shows the prediction accuracy when using the threshold based hot code predictor to emulate the three suites of benchmarks. For this experiment we have fixed  $\beta_I = 70$ ,  $\alpha_T = 150k$  and  $\beta_T = 1.5$  which produces an  $T_H = 2190$  according to Equation 3.3. The second column in Table 5.2 shows the number of correct predictions (i.e. when the instruction final execution frequency is greater than  $T_H$ .) for both cases we consider,  $T_P = 100$  and  $T_P = 1000$ . The third and fourth column shows the total number of predictions and the number (and percentage) of incorrect predictions made when  $T_P = 100$ , respectively. The fifth and sixth columns, have the same meaning of the third and fourth columns but for  $T_P = 1000$ .

The first interesting thing to notice is that the number of correct predictions does not change when we increase the prediction threshold from 100 to 1000. Since the predictions are correct only when the instruction final execution frequency crosses  $T_H$ , increasing  $T_P$  does not affect the number of correct predictions. So this number will be the same for whenever  $T_P < T_H$ .

Increasing the prediction threshold ( $T_P$ ), however, does in fact greatly decrease the number of predictions and the number of incorrect predictions made. With a prediction threshold greater, fewer instructions reaches the threshold so fewer predictions are made. Those instructions that do not reach the new threshold are instructions that were previously being mispredicted, so the number of mispredictions decrease with an increase in the prediction threshold. One may argue that we could increase the threshold so as to eliminate the misprediction overhead, however increasing the threshold exacerbate other sources of overhead. This point will be further explained in Section 5.3.

Another observation about these results is that the prediction accuracy for a given threshold seems to be independent of particular characteristics of the benchmarks. For example, considering  $T_P = 100$  the greatest difference between the average percentage of incorrect predictions is 8% (Boot and Sysmark). However considering only the percentage of correct/incorrect predictions as an estimate for misprediction overhead is misleading since the benchmarks have very different footprints.

### 5.1.3 Execution Coverage

Table 5.3 shows the size of the 85%, 90% and 95% coverage set for the three benchmarks suites we studied. More specifically, the second column of Table 5.3 shows the size of the static footprint of each benchmark, i.e. the number of different static instructions that executed at least once; the third column shows the number of static instructions (and its percentage) that is needed to cover 85% of the dynamically executed instructions; the fourth and fifth columns have the same meaning as the third, except that they show values

		$T_P = 100$		$T_P = 1000$	
<b>Benchmark</b>	<b>CPreds</b>	<b>TPreds</b>	<b>IPreds</b>	<b>TPreds</b>	<b>IPreds</b>
Boot-Debian5	167,468	683,339	515,871 (75%)	242,939	75,471 (31%)
Boot-Win7	181,166	621,491	440,325 (71%)	257,849	76,683 (30%)
Boot-WinXP	94,448	409,681	315,233 (77%)	146,004	51,556 (35%)
<b>Average:</b>			74%		32%
Sysmark-DFA	893,064	2,810,450	1,917,386 (68%)	1,204,290	311,226 (26%)
Sysmark-Office	3,982,460	11,312,255	7,329,795 (65%)	5,299,491	1,317,031 (25%)
Sysmark-SM	791,084	2,213,594	1,422,510 (64%)	1,068,006	276,922 (26%)
<b>Average:</b>			66%		26%
Perlbench	69,442	134,384	64,942 (48%)	78,434	8,992 (11%)
Bzip2	19,607	59,455	39,848 (67%)	24,314	4,707 (19%)
Gcc	123,750	208,911	85,161 (41%)	143,284	19,534 (14%)
Mcf	18,621	74,101	55,480 (75%)	25,300	6,679 (26%)
Gobmk	62,709	131,311	68,602 (52%)	73,048	10,339 (14%)
Hmmer	20,123	52,242	32,119 (61%)	27,940	7,817 (28%)
Sjeng	75,590	356,978	281,388 (79%)	110,602	35,012 (32%)
Libquantum	70,100	384,090	313,990 (82%)	111,931	41,831 (37%)
H264ref	39,497	112,126	72,629 (65%)	51,885	12,388 (24%)
Omnetpp	43,155	132,950	89,795 (68%)	57,469	14,314 (25%)
Astar	14,194	43,029	28,835 (67%)	18,416	4,222 (23%)
Xalancbmk	68,211	134,523	66,312 (49%)	77,942	9,731 (12%)
GemsFDTD	82,833	375,398	292,565 (78%)	130,069	47,236 (36%)
Bwaves	71,233	380,712	309,479 (81%)	109,872	38,639 (35%)
CactusADM	99,608	499,112	399,504 (80%)	156,277	56,669 (36%)
Calculix	127,485	433,755	306,270 (71%)	169,292	41,807 (25%)
DealII	70,408	170,849	100,441 (59%)	86,721	16,313 (19%)
Games	134,824	493,494	358,670 (73%)	182,693	47,869 (26%)
Gromacs	101,579	458,498	356,919 (78%)	146,627	45,048 (31%)
Lbm	99,879	493,582	393,703 (80%)	155,865	55,986 (36%)
Leslie3d	66,288	308,826	242,538 (79%)	97,444	31,156 (32%)
Milc	55,914	217,155	161,241 (74%)	77,018	21,104 (27%)
Namd	33,126	81,967	48,841 (60%)	39,271	6,145 (16%)
Povray	54,866	174,242	119,376 (69%)	74,130	19,264 (26%)
Soplex	67,526	234,885	167,359 (71%)	88,785	21,259 (24%)
Sphinx3	53,950	147,622	93,672 (63%)	65,797	11,847 (18%)
Tonto	164,143	493,792	329,649 (67%)	215,116	50,973 (24%)
Zeusmp	116,052	489,899	373,847 (76%)	166,381	50,329 (30%)
<b>Average:</b>			68%		25%

Table 5.2: Benchmark Characterization - prediction accuracy of the threshold based predictor with  $T_P = 100$  and  $T_P = 1000$



Benchmark	Static	Cov-85%	Cov-90%	Cov-95%
Boot-Debian5	5,647,259	35,300 (0.63%)	63,242 (1.12%)	128,181 (2.27%)
Boot-Win7	2,877,855	42,603 (1.48%)	65,150 (2.26%)	127,953 (4.45%)
Boot-WinXP	2,053,038	29,932 (1.46%)	49,032 (2.39%)	102,568 (5.00%)
<b>Average:</b>		1.19%	1.92%	3.92%
Sysmark-DFA	11,859,140	3,910 (0.03%)	10,083 (0.09%)	37,433 (0.32%)
Sysmark-Office	40,061,419	115,604 (0.29%)	203,876 (0.51%)	447,792 (1.12%)
Sysmark-SM	6,435,727	6,261 (0.10%)	12,851 (0.20%)	37,452 (0.58%)
<b>Average:</b>		0.14%	0.26%	0.67%
Perlbench	565,866	1,533 (0.27%)	2,710 (0.48%)	5,439 (0.96%)
Bzip2	329,669	520 (0.16%)	670 (0.20%)	1,061 (0.32%)
Gcc	446,808	5,967 (1.34%)	10,291 (2.30%)	20,352 (4.55%)
Mcf	87,515	206 (0.24%)	299 (0.34%)	451 (0.52%)
Gobmk	422,433	4,513 (1.07%)	5,914 (1.40%)	8,410 (1.99%)
Hmmer	260,448	68 (0.03%)	72 (0.03%)	76 (0.03%)
Sjeng	2,938,619	1,902 (0.06%)	2,442 (0.08%)	3,486 (0.12%)
Libquantum	3,251,529	32 (0.00%)	42 (0.00%)	90 (0.00%)
H264ref	589,358	1,460 (0.25%)	2,198 (0.37%)	4,450 (0.76%)
Omnetpp	863,018	1,773 (0.21%)	2,184 (0.25%)	2,963 (0.34%)
Astar	229,132	219 (0.10%)	263 (0.11%)	323 (0.14%)
Xalancbmk	642,284	1,510 (0.24%)	2,368 (0.37%)	3,677 (0.57%)
GemsFDTD	2,906,935	2,220 (0.08%)	3,117 (0.11%)	4,373 (0.15%)
Bwaves	3,181,906	289 (0.01%)	501 (0.02%)	1,195 (0.04%)
CactusADM	4,087,989	1,555 (0.04%)	1,647 (0.04%)	1,738 (0.04%)
Calculix	3,078,574	155 (0.01%)	270 (0.01%)	899 (0.03%)
DealII	878,394	612 (0.07%)	1,256 (0.14%)	2,527 (0.29%)
GameSS	3,785,573	1,450 (0.04%)	2,261 (0.06%)	4,904 (0.13%)
Gromacs	3,726,997	1,000 (0.03%)	1,666 (0.04%)	3,095 (0.08%)
Lbm	4,023,221	1,923 (0.05%)	2,089 (0.05%)	2,256 (0.06%)
Leslie3d	2,513,050	351 (0.01%)	372 (0.01%)	393 (0.02%)
Milc	1,639,467	649 (0.04%)	852 (0.05%)	1,287 (0.08%)
Namd	401,790	2,453 (0.61%)	2,730 (0.68%)	3,006 (0.75%)
Povray	1,218,271	1,699 (0.14%)	2,519 (0.21%)	4,564 (0.37%)
Soplex	1,720,213	1,039 (0.06%)	1,341 (0.08%)	2,194 (0.13%)
Sphinx3	977,453	175 (0.02%)	280 (0.03%)	565 (0.06%)
Tonto	3,280,610	6,889 (0.21%)	10,037 (0.31%)	15,662 (0.48%)
Zeusmp	3,760,594	7,520 (0.20%)	8,905 (0.24%)	10,480 (0.28%)
<b>Average:</b>		0.20%	0.29%	0.47%

Table 5.3: Benchmark Characterization - size of the static instruction set covering 85%, 90% and 95% of the dynamic instruction stream.



for 90% and 95%, respectively.

The first thing we noticed is that a small percentage of static instructions covers 95% of executed instructions. As we noticed in other experiments (not shown here), this is mainly due to numerous instructions that seldom execute, and to other instructions that are highly executed.

As it can be seen from the averages, the boot processes are those that require a large footprint to cover the execution stream. This is due to the way the boot works, that is, spanning multiple services and thus causes many hash table entries to be filled. SPEC benchmarks are those that require fewer static instructions to cover the dynamic stream. Also interesting to notice that as we step from 85% to 90% and 95%, the SPEC benchmarks are those where the static footprint increases the less - which shows that SPEC has many instructions that are highly executed.

In terms of percentage, Sysmark benchmarks need a comparatively short static footprint to cover the dynamic stream, however it has the largest static footprint among the benchmark suites considered. For example, Office requires 1.12% of its static instructions to cover 95% of its execution, however quantitatively this is greater than the whole static footprint of a few SPEC benchmarks.

## 5.2 Estimation of the Model Parameters $\beta_I$ , $\alpha_T$ and $\beta_T$

**Steady-state interpretation cost  $\beta_I$ :** Column  $\beta_I$  of Table 5.4 shows the average cost, in cycles, for interpreting instructions of the SPEC CPU 2006 benchmarks. For example, when running the benchmark *400.perlbench*, Bochs took, on average, 48 host cycles to interpret each guest instruction. The average number of cycles to interpret instructions of floating point benchmarks is higher due to the elevated cost of emulating floating point instructions via software. In order to accommodate for these discrepancies, our overhead model assumes that the interpretation cost  $\beta_I$  varies from 30 to 220 host cycles.

**Start-up translation cost  $\alpha_T$ :** Column  $\alpha_T$  of Table 5.4 shows the average cost, in cycles, to translate each LLVM bytecode instruction from C and C++ SPEC CPU2006 benchmark programs to x86 code. For example, LLVM took, on average, 91952 host cycles to translate (compile) each guest LLVM instruction of the *400.perlbench* program to x86 instructions.

However, the parameter  $\alpha_T$  suffers a variability that is more complex than the other parameters in the model. The execution times ( $\alpha_T$ ) of the algorithms used for code optimization and generation run a number of steps that is far from trivial. The algorithms'

Integer Benchmarks				Floating Point Benchmarks			
Benchmark	$\beta_I$	$\alpha_T$	$\beta_T$	Benchmark	$\beta_I$	$\alpha_T$	$\beta_T$
400.perlbench	48	91952	1.02	470.lbm	85	41315	2.11
401.bzip2	40	94429	1.40	482.sphinx3	117	60468	1.31
403.gcc	48	72546	1.58	433.milc	141	102610	3.50
429.mcf	42	26652	3.18	444.namd	144	96285	1.51
445.gobmk	48	66671	1.83	447.dealII	64	177565	–
456.hmmer	95	53780	1.47	450.soplex	65	148678	2.80
458.sjeng	40	89039	1.56	453.povray	92	73341	2.25
462.libquantum	31	115724	1.64	434.zeusmp	122	–	–
464.h264ref	40	56832	1.29	435.gromacs	173	–	–
471.omnetpp	59	175220	5.28	436.cactusADM	209	–	–
473.astar	44	110029	2.48	437.leslie3d	213	–	–
483.xalancbmk	38	173994	4.38	454.calculix	128	–	–
-	-	-	-	459.GemsFDTD	133	–	–
-	-	-	-	465.tonto	94	–	–
-	-	-	-	481.wrf	108	–	–
-	-	-	-	410.bwaves	103	–	–
-	-	-	-	416.gamess	106	–	–

Table 5.4: Measured cost for ( $\beta_I$ ) interpretation, ( $\alpha_T$ ) compilation and ( $\beta_T$ ) native execution. Dashes mark Fortran benchmarks for which we do not have  $\alpha_T$  and  $\beta_T$  values, with exception of dealII for which profiling failed to produce correct output and thus to extract  $\beta_T$ . All costs are in cycles per instruction.

complexity in time does not always depend solely on the number of instructions, but also on the program control flow information and data dependences among instructions. For worklist-based algorithms, the worst case time and average case time may differ greatly, making it hard to have an analytical model to compute its run time. For this reason, we model the translation cost ( $\alpha_T$ ) by a wide range of values.

Our goal in measuring  $\alpha_T$  is to determine the minimum number of cycles per instruction, using an LLVM-based VM, to translate the program fragment. We show that, even in this situation, there is still a considerable misprediction overhead. Nevertheless, we also provide information on a virtual machine geared at optimizing code.

For each one of the 71261 C and C++ SPEC CPU2006 functions compiled, we calculated  $\alpha_T$  using the ratio of the number of cycles necessary to compile the whole function over the number of LLVM instructions in the function, in order to estimate good lower and upper bounds for the  $\alpha_T$ . Figure 5.1 presents a histogram where each bin shows the percentage of functions in a given range for  $\alpha_T$ , considering a first scenario where no optimizations are enabled. That is, the abscissa represents different costs to translate a function while the Ordinate represent the percentage of functions which had the translation cost between two adjacent (to the left) ticks of the x-axis.

The histogram confirms that  $\alpha_T$  varies significantly, depending on several param-

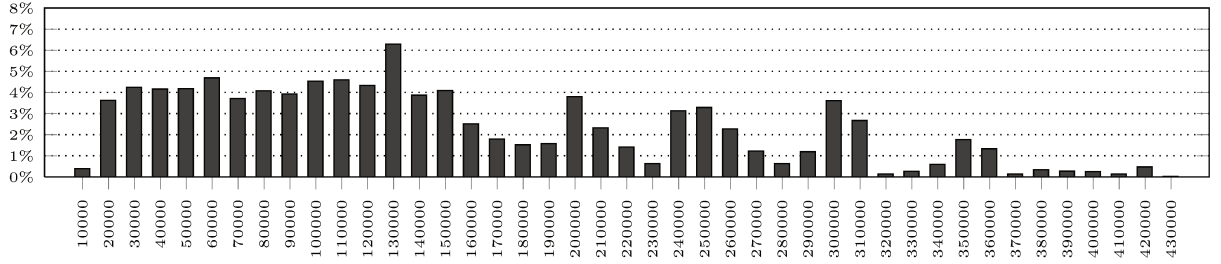


Figure 5.1: Histogram showing the percentage of total SPEC CPU2006 functions sharing a given range of the cost  $\alpha_T$ , in cycles. The Abscissa represent the cost to translate the function.

ters of the compiled function. For instance, the LLVM instruction selection pass, which dominates the compilation time in this scenario for several functions, was very fast in a perlbench function composed entirely of 35 stores. This function had one of the lowest  $\alpha_T$ . On the other hand, functions with a single instruction often have the highest  $\alpha_T$  because it pays a high price to prepare data structures for compiling a single LLVM instruction.

In this first scenario (compiling with no optimizations), 90% of all functions have  $\alpha_T$  greater or equal to 36,000 cycles. The average  $\alpha_T$  was 154,000 cycles. Therefore, if code regions are assumed to include entire functions, we expect that a VM that translates a guest ISA to a different host ISA will pay at least 36,000 host cycles per translated instruction to translate the majority of the hot regions. The third column of Table 5.4 shows the averages for this first scenario, for each benchmark.

The second scenario enables all “-O2” optimizations. In this case, LLVM takes at least 145,000 cycles for 90% of all functions. The average  $\alpha_T$  was 1,073,000 cycles. In the next section, we present a study using  $\alpha_T$  in the range of 30,000 cycles up to 850,000 cycles because we focus in the fastest cases with respect to both scenarios. For greater values, the misprediction overhead is even bigger.

**Steady-state translation cost  $\beta_T$ :** Column  $\beta_T$  of Table 5.4 shows the average cost, in cycles, for emulating each LLVM bytecode instruction using C and C++ SPEC CPU2006 benchmark programs after translating them to native code. For example, the host machine took, on average, 1.02 cycles to emulate each guest LLVM instruction in the *400.perlbench* program. In contrast, the cost of emulating the same benchmark with interpretation in Bochs was 48 host cycles, on average, showing the benefits of paying a high start-up cost for translation.

As explained in Chapter 4, there are two scenarios for measuring  $\beta_T$ , but Table 5.4 shows only the first and more important one, namely the one for a VM which does not apply optimizations. In this case, the generated code quality is poorer and the average

Parameter	Start	End
Prediction Threshold ( $T_P$ )	25	3000
Interpretation ( $\beta_I$ )	30	220
Compilation ( $\alpha_T$ )	30,000	850,000
Execution ( $\beta_T$ )	0.5	3.0

Table 5.5: Range of interpretation, compilation and execution cost experimented. All costs are in cycles per instruction.

$\beta_T$  measured among selected SPEC CPU2006 benchmarks is 2.25 host cycles per target instruction. The second scenario illustrates a VM that generates good quality native code by optimizing it, and the average  $\beta_T$  measured in this situation is 1.11 host cycles per target instruction. The code is, on average, slightly more than twice faster.

Based on the aforementioned results, we selected three ranges of for the  $\beta_I$ ,  $\alpha_T$  and  $\beta_T$  parameters, which are summarized on Table 5.5.

### 5.3 The Effect of the Model's Parameters in the Emulation Overhead

Figure 5.2 shows how the misprediction overhead vary in the three sets of benchmarks for each model's parameter. In each row of figures the varying parameter is shown on the x-axis. Except for the respective parameter on the x-axis, the other model's parameters are set this way:  $T_P = 1000$ ,  $\beta_I = 70$ ,  $\alpha_T = 150000$  and  $\beta_T = 1.5$ .

As it can be noticed from Figure 5.2 the parameter that most affects the misprediction overhead is the prediction threshold ( $T_P$ ) followed by  $\alpha_T$ ,  $\beta_I$  and  $\beta_T$ . We do not show results for  $\alpha_I$  since it showed little impact on the overhead. Although these last three parameters does not affect the prediction behavior, they are used for calculating the hotness threshold ( $T_H$ ) that is used by the oracle predictor to predict what is and what is not a hot region.

Analyzing the graphics of the effect of the threshold predictor on the misprediction overhead, one can clearly see that by increasing the threshold we can drastically reduce the misprediction overhead in some cases (e.g. Sysmark and Boots). When we increase the prediction threshold it becomes closer to the hotness threshold ( $T_H$ ) and thus it is more likely that it will cross  $T_H$ , also the number of predictions greatly decrease and thus the number of mispredictions. However increasing  $T_P$  also increases the amount of time we need to wait until flag a region as hot and thus the system expends more time executing unoptimized code. We see that, at some point, if we further increase the threshold the misprediction threshold start to increase. This is a consequence of the emulation process

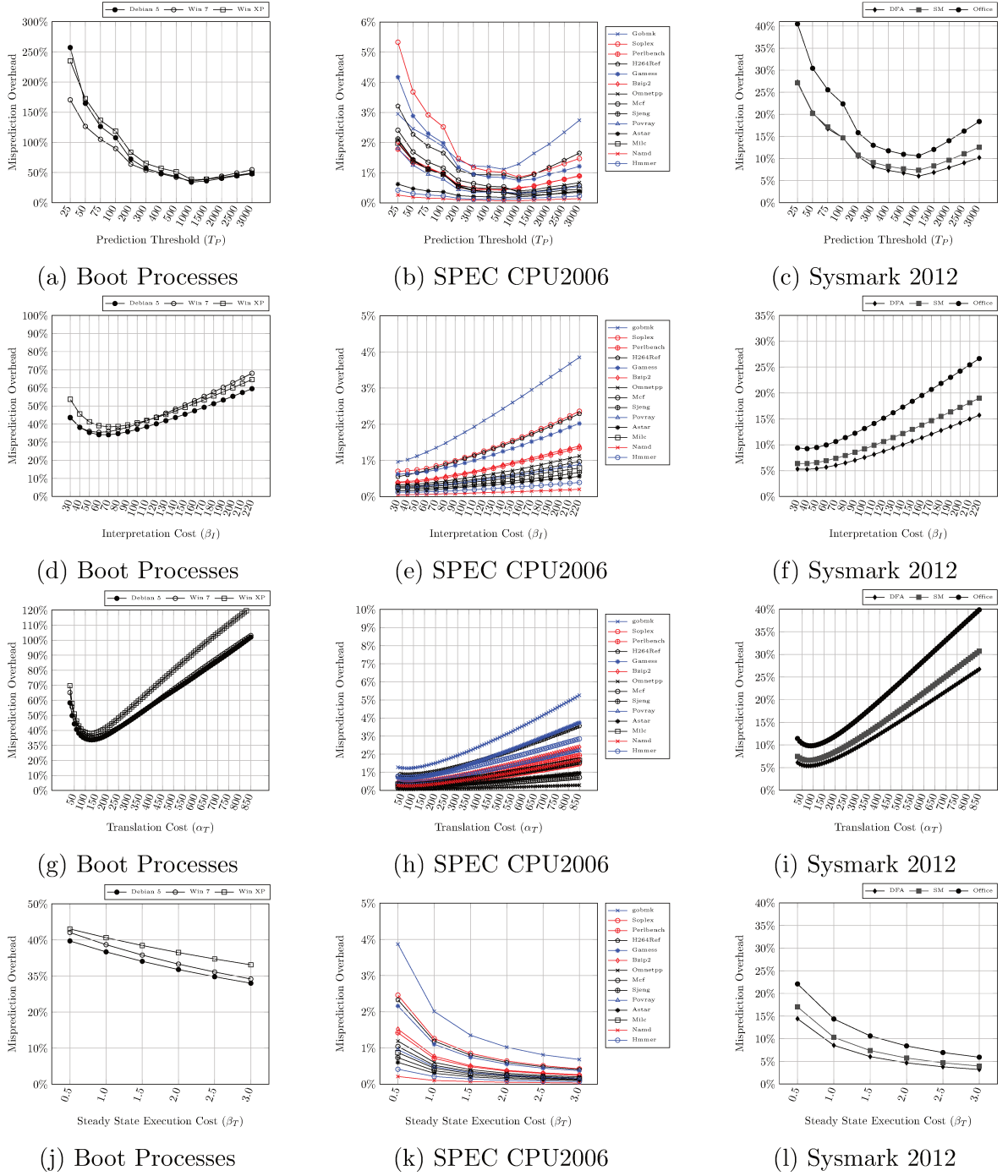


Figure 5.2: Misprediction Overhead for Sysmark 2012, SPEC CPU2006 and Windows 7 Boot in Function of Each Model Parameter

being dominated by interpretation rather than translation. We will show further results on this last point in Section 5.5.

From Figures 5.2g-5.2i and Figures 5.2d-5.2f we can see the effect of  $\alpha_T$  and  $\beta_I$  on the misprediction overhead, respectively. In all benchmarks increasing any of these parameters also increase the **emulation** cost. For the misprediction overhead, we see that in general the overhead increases as the parameter values increase, however there is a region that the overhead in fact decreases! To understand why this happens notice that both parameters do not affect how the threshold-based predictor predicts code, they only affect (linearly) the amount of cycles required for emulation. However these parameters do affect the behavior of the oracle predictor since they are used to calculate  $T_H$  and thus to decide which region is in fact hot! The effect of these parameters in the oracle emulation cost is far from linear since it is dependent on the amount of instructions which have the execution frequency in a given range  $(0 - T_H)$ . In other words, the slope is due to increase/decrease of the number of instructions predicted as hot/cold when the hotness threshold changes.

The effect of the parameter  $\beta_T$  on the misprediction overhead is show in Figures 5.2j-5.2l. As shown, the misprediction overhead decreases when we increase  $\beta_T$ , this is due to the same reason we explained for the  $\alpha_T$  and  $\beta_I$ . The lines are smooth because  $\beta_T$  only slightly affect the calculation of  $T_H$ . The decrease in the misprediction overhead may suggest, at first, that we have a more efficient emulation process, however this is misleading; the emulation cost increases when we increase  $\beta_T$ , leading to slow emulation.

## 5.4 Misprediction Overhead

Figures 5.3a-5.3c show the minimum misprediction overhead for the Windows 7 boot, Sysmark 2012 (Office Productivity scenario) and SPEC CPU 2006 *403.gcc* (with reference input), respectively. These figures show how the overhead changes with the translation start-up cost ( $\alpha_T$ ), and the translation steady-state cost ( $\beta_T$ ). For all points in these graphs, the parameters  $\beta_I$  and  $T_P$ , are unconstrained inside their respective ranges (Table 5.5). Thus these figures reveal the minimum misprediction overhead regardless of the specific values of these parameters. For example, consider a VM emulating the Windows 7 boot process enabled with a TBP, moreover assume that the translation start-up cost ( $\alpha_T$ ) is 300,000 host cycles (per translated instruction) and the steady-state cost ( $\beta_T$ ) is 1.5 host cycles (per emulated guest instruction). In this scenario even using the best threshold value and Interpretation cost, the execution would still suffer from 40% misprediction overhead.

Notice that the three figures have the same pattern, and, in fact, all benchmarks we experimented present this same behavior. As it can be seen, if the compilation cost  $\alpha_T$  increases, the misprediction overhead also increases. This is an intuitive trend, since the

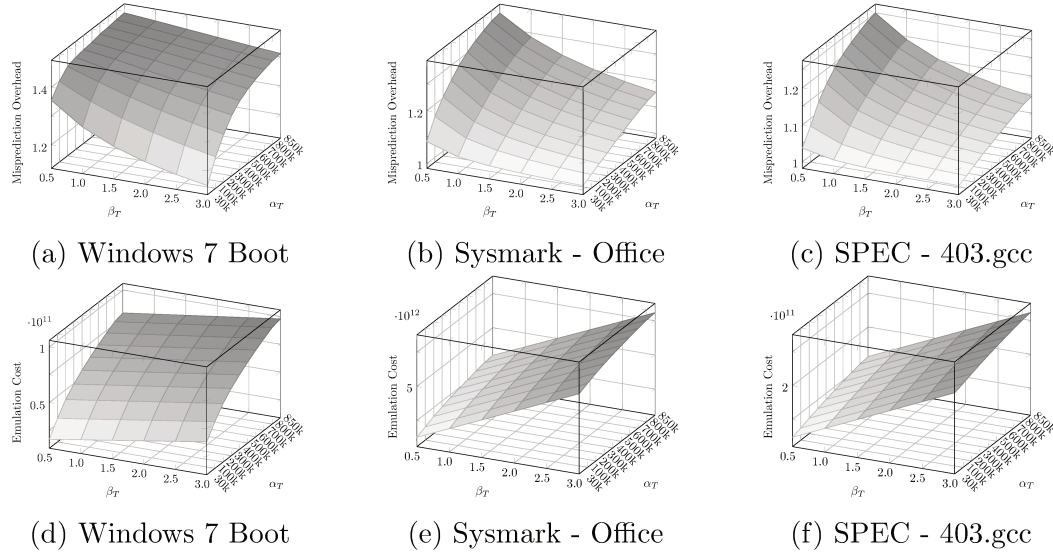


Figure 5.3: Misprediction and Emulation Overhead for Sysmark 2012 (Office), SPEC CPU2006 (403.gcc) and Windows 7 Boot.

main source of overhead of the TBP in relation to the oracle predictor is due to warm code translation; once the translation cost increases the overhead also increases. Also notice that the misprediction overhead decreases as the steady-state execution cost ( $\beta_T$ ) increases. This is not unexpected, the higher is the cost to translate, the higher is the total emulation cost, even for the oracle predictor. Hence, the misprediction overhead becomes a smaller share of the total emulation time.

Figures 5.3d-5.3f show surfaces representing the minimum emulation cost, in host cycles, for a VM emulating the aforementioned benchmarks and parameter values. Here notice that, as expected, for the three benchmarks, the emulation cost is minimal when the translation and execution costs are minimum. These surfaces are shown to illustrate how the emulation cost contrasts with the misprediction overhead. Notice that although smaller values of  $\alpha_T$  and  $\beta_T$  gives the minimum emulation cost, this is not a common case scenario for a VM. In the next paragraph we present two scenarios to illustrate how the misprediction overhead can severely affect the VM performance.

Figure 5.4 shows an example of how the development of a virtual machine can be severely affected by the misprediction overhead. We use our measured values for  $\beta_T$  and  $\alpha_T$  to build two scenarios where the VM is improved by adding more sophisticated optimizations. The first scenario considers the best cases, in other words, the lowest values for  $\beta_T$  and  $\alpha_T$  of our experiments. The second case considers the average measured values, as described in Chapter 4. The figure shows the trajectory on the overhead surface when the VM progressively supports more sophisticated optimizations and better code quality



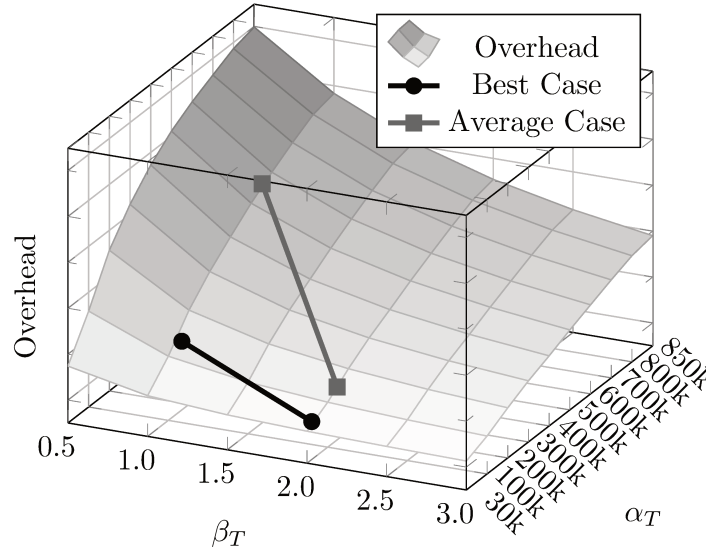


Figure 5.4: Trend lines for the evolution of VM in terms of generated code quality and their consequent misprediction overhead.

is generated. We presented a range of  $\alpha_T$  values in this work, but here two trend lines are presented. The first uses our measured average values and the second uses  $\alpha_T$  values for which 90% of all measurements are guaranteed to be greater than. The latter is a conservative estimate, since there is a high probability the VM will have higher overheads than those delimited by this curve. These curves explain how a good predictor increases in importance as the VM quality improves. In the second scenario, our results suggest that in a virtual machine that uses time-consuming optimizations to produce faster code the misprediction overhead is more relevant, since the mistranslation cost becomes more expressive in relation to the faster translated code.

## 5.5 Misprediction Overhead Characterization

Figure 5.5 shows the misprediction overhead of all three sets of benchmarks if we consider a system that can produce a reasonably fast code in a moderate amount of time. The parameters used to draw these results were:  $\alpha_I = 0$ ,  $\beta_I = 70$ ,  $\alpha_T = 150,000$ , and  $\beta_T = 1.5$  and two prediction thresholds  $T_P = 25$  and  $T_P = 1,000$ . There is a noticeable discrepancy between the results of SPEC and the other benchmarks. Considering a prediction threshold of 25 the maximum overhead measured in SPEC was achieved by *403.gcc* with nearly 10% of misprediction overhead, with the same threshold the minimum overhead among the OS boots and Sysmark benchmark was 270% and 27%, respectively.

The boot processes and interactive applications (such as those of Sysmark 2012) ex-



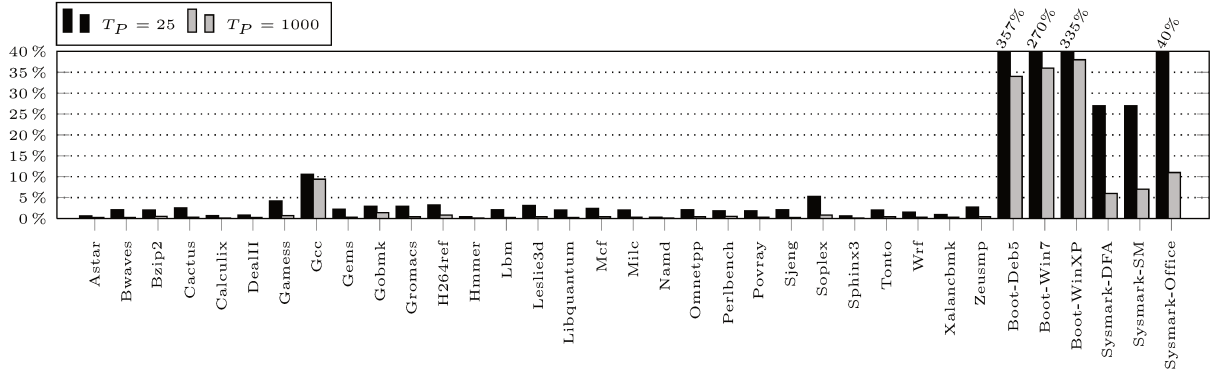


Figure 5.5: Maximum misprediction overhead for SPEC CPU2006 and minimum misprediction overhead for several OS boots and Sysmark 2012 benchmark.

exercise a larger code footprint when compared to SPEC CPU2006 scenarios. This characteristic leads to an increase in the number of warm code regions and, consequently an increase in the hot code misprediction overhead.

One could argue that this overhead can be reduced if a greater value for the prediction threshold is used. Figure 5.5 shows the maximum (for SPEC) and minimum (for OS boot and Sysmark 2012) misprediction overhead if we use a prediction threshold of 1000. The overhead is reduced for all benchmarks, notably for OS boots and Sysmark, this result support the argument that the overhead seen when  $T_P = 100$  is due to a large amount of warm code. *However, more important is to note that even with a  $T_P = 1000$  the average minimum overhead for OS boots and Sysmark 2012 is 36% and 8%, respectively!*

This huge difference among the results illustrates that using only SPEC CPU2006 benchmarks when measuring the performance of a VM that employs the TBP may lead to misleading results.

To support our previous arguments, we show in Figure 5.6 the misprediction overheads quantified in Figure 5.5, in terms of two components: the **Warm Code** and **Hot Code** overhead, which refers to cases 1b and 1c of Figure 3.1, respectively. We notice that due to the parameters we used, the cases 2(a)-(c) of Figure 3.1 do not occur.

For  $T_P = 25$  the overhead is predominantly due to warm code translation (over 99%) and the delay caused by late hot code detection is minimum. However, when  $T_P = 1000$  is used, the increase in the number of interpreted regions together with a decrease in the number of warm code translation, makes the interpretation cost the major component of the overhead - for all benchmarks the warm code translation cost is below 50% of the overhead.

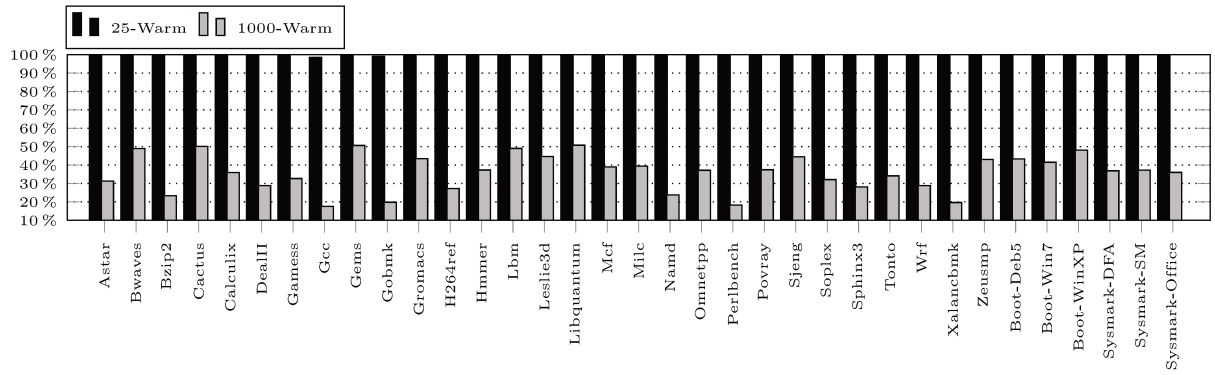


Figure 5.6: Hotness misprediction overhead split in two components: warm and cold code translation overhead. The bars show the warm code translation overhead (when  $T_P = 25$  and  $T_P = 1000$ , they complement represent cold code translation overhead).

# Chapter 6

## Conclusions

In the last two decades virtual execution environments have gained increasing attention from both the Industry [16,19,37] and Academia [2,14,46]. This interest has been fostered by the versatility of such environments, which through the decoupling of two distinct interfaces enables innovative design [16], portability [6,16,24,48] and efficient resource usage [5,13], among others. However, in order for these environments to become a reliable alternative for end-user scenarios, a careful study of the behavior of such VMs is required.

This work characterized a widely used threshold based technique to predict hot code and showed that it can lead to misleading results on understanding VM performance. We developed a mathematical model to estimate the misprediction overhead incurred from a widely used approach to predict hot code on VMs. Our results showed that using **only** SPEC CPU 2006 benchmarks to report VM performance can lead to incorrect results – while SPEC benchmarks may suffer from up to 10% of hotness misprediction overhead, Sysmark benchmarks may suffer from at least 27% of overhead. We also show that, as VM spend more cycles optimizing the translated code, the overhead due to hot code mispredictions becomes more relevant. We decompose the misprediction overhead and show that its major component is translating warm code and that this overhead can be exacerbated in large code footprint applications, such as those found in interactive environments.

# Bibliography

- [1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, & Tools*. Pearson, 2007.
- [2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno virtual machine. *IBM Systems Journal*, 39(1), 2000.
- [3] 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*, 21665, 2000.
- [4] T. Baba, T. Masuho, T. Yokota, and K. Ootsu. Design of a two-level hot path detector for path-based loop optimizations. In *Proceedings of the third conference on IASTED International Conference: Advances in Computer Science and Technology*, 2007.
- [5] 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*, 2000.
- [6] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on itanium-based systems. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003.
- [7] F. Bellard. QEMU - Quick EMUlator. <http://www.qemu.org/>, 2012. [Accessed November 11th, 2012].
- [8] E. Borin and Y. Wu. Characterization of DBT overhead. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, 2009.

- [9] R. P. L. Buse and W. Weimer. The road not taken: Estimating path execution frequency statically. In *Proceedings of the 31st International Conference on Software Engineering*, 2009.
- [10] H. Chen, W. Hsu, J. Lu, P. Yew, and D. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, 2003.
- [11] T. Y. Chen, F. Kuo, and R. Merkel. On the statistical properties of the f-measure. In *Proceedings of the Fourth International Conference on Quality Software*.
- [12] W. Chen, D. Chen, and Z. Wang. An approach to minimizing the interpretation overhead in dynamic binary translation. *The Journal of Supercomputing*, 61(3), 2012.
- [13] W. K. Chen, S. Lerner, and R. C. D. M. Gillies. Mojo: A dynamic optimization system. *Proceedings of the 3rd Workshop on Feedback-Directed and Dynamic Optimization*, 2000.
- [14] C. Cifuentes and M. Van Emmerik. UQBT: adaptable binary translation at low cost. *Computer*, 33(3), 2000.
- [15] BAP Co. Sysmark 2012 suite.
- [16] J. C. Dehnert, B. K. Grant, J. P. 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. 2003.
- [17] E. Duesterwald and V. Bala. Software profiling for hot path prediction: less is more. *SIGPLAN Notices*, 35(11), 2000.
- [18] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003.
- [19] K. Ebcioglu and E. R. Altman. DAISY: dynamic compilation for 100In *Proceedings of the International Symposium on Computer Architecture*, 1997.
- [20] E. J. Emond and D. W. Mason. A new rank correlation coefficient with application to the consensus ranking problem. *Journal of Multi-Criteria Decision Analysis*, 2002.

- [21] J. L. Henning. SPEC CPU2000: measuring CPU performance in the new millennium. *Computer*, 2000.
- [22] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 2006.
- [23] D. Hong, C. Hsu, P. Yew, J. Wu, W. Hsu, P. Liu, C. Wang, and Y. Chung. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012.
- [24] R. J. Hookway and M. A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1), 1997.
- [25] S. Hu and J. E. Smith. Reducing startup time in co-designed virtual machines. In *Proceedings of the International Symposium on Computer Architecture*, 2006.
- [26] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, volume 2: instruction set reference edition.
- [27] S. Johnson and S. Valli. An approach to predict hot methods using support vector machines. In *Proceedings of the 16th International Conference on Advanced Computing and Communications.*, 2008.
- [28] J. C. King. Symbolic execution and program testing. *Communications of ACM*, 19(7), 1976.
- [29] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.
- [30] C. Lee. Utdsp benchmark suite. <http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html>, 2013. [Accessed January 17th, 2013].
- [31] S. Liu and J. Zhang. Program analysis: from qualitative analysis to quantitative analysis. In *Proceedings of the 33rd International Conference on Software Engineering*, 2011.
- [32] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Programming language design and implementation*, 2005.

- [33] F. Ma, S. Liu, and J. Zhang. Volume computation for boolean combination of linear arithmetic constraints. In *Proceedings of the 22nd International Conference on Automated Deduction*, 2009.
- [34] M.C. Merten, A.R. Trick, C.N. George, J.C. Gyllenhaal, and W.W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support run-time optimization. In *Proceedings of the 26th International Symposium on Computer Architecture*, 1999.
- [35] J. Moreira, D. César, G. Araújo, E. Borin, and S. Rigo. Asynchronous program flow verification through binary instrumentation on QEMU. In *Proceedings of the Workshop on Architectural and MicroArchitectural Support for Binary Translation*, 2012.
- [36] G. Ottoni, T. Hartin, C. Weaver, J. Brandt, B. Kuttanna, and H. Wang. Harmonia: a transparent, efficient, and harmonious dynamic binary translator targeting the intel architecture. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, 2011.
- [37] M. Paleczny, C. Vick, and C. Click. The java hotspot server compiler. In *Proceedings of the Symposium on Java™ Virtual Machine Research and Technology*, 2001.
- [38] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the International Symposium on Microarchitecture*, 2006.
- [39] R. Rosner, M. Moffie, Y. Sazeides, and R. Ronen. Selecting long atomic traces for high coverage. In *Proceedings of the 17th annual international conference on Supercomputing*, 2003.
- [40] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2001.
- [41] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the Annual international symposium on Computer architecture*, 2003.
- [42] J.E. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann, 2005.
- [43] Open Source. Bochs - the cross platform IA-32 (x86) emulator. <http://bochs.sourceforge.net/>, 2013. [Accessed January 17th, 2013].

- [44] PC Magazine VeriTest. Business WinStone Benchmark. <http://www.veritest.com/benchmarks/bwinstone/>.
- [45] B. C. Wang, H. Zheng, M. Jr. Breternitz, and Y. Wu. Two-pass mret trace selection for dynamic optimization, 2010.
- [46] C. Wang, S. Hu, H. Kim, S. Nair, M. Breternitz, Z. Ying, and Y. Wu. StarDBT: An efficient multi-platform dynamic binary translation system. volume 4697 of *Lecture Notes in Computer Science*. 2007.
- [47] Y. Wu, M. Breternitz, J. Quek, O. Etzion, and J. Fang. The accuracy of initial prediction in two-phase dynamic binary translators. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.
- [48] C. Zheng and C. Thompson. PA-RISC to IA-64: transparent execution, no recompilation. *Computer*, 33(3):47–52, 2000.