Modeling Virtual Machines Misprediction Overhead

Divino César, Rafael Auler, Rafael Dalibera, Sandro Rigo, Edson Borin and Guido Araújo
Institute of Computing, University of Campinas
Campinas, São Paulo - Brazil
{divcesar, auler, sandro, edson, guido}@ic.unicamp.br, rafaeldalibera@gmail.com

***Abstract*—Virtual machines are versatile systems that can support innovative solutions to many problems. These systems usually rely on emulation techniques, such as interpretation and dynamic binary translation, to execute guest application code. Usually, in order to select the best emulation technique for each code segment, the system must predict whether the code is worth compiling (frequently executed) or not, known as *hotness prediction*.**

In this paper we show that the *threshold-based hot code predictor*, frequently mispredicts the code hotness and as a result the VM emulation performance become dominated by miscompilations. To do so, we developed a mathematical model to simulate the behavior of such predictor and using it we quantify and characterize the impact of mispredictions in several benchmarks. We also show how the threshold choice can affect the predictor, what are the major overhead components and how using SPEC to analyze a VM performance can lead to misleading results.

I. Introduction

One of the main goal of a Virtual Machine (VM) [1] 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]–[14], some targeted at emulating a sole application (process VMs) and others at emulating a whole system (system VMs). Besides portability [5], [14], [15], VMs can be used for several others purposes like: to support legacy code execution [7], [12], dynamic program optimization [4], [6], [8], program shepherding [9], [10] and dynamic program instrumentation [11].

Virtual machines are very versatile, but they come with an inherent price: the emulation cost. Following the VM taxonomy proposed by Smith and Nair [1], regarding emulation technique, there are two main classes of VMs: 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. In this technique, the source representation of the guest software is dynamically 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 benefit from the execution of the optimized/translated code.

In order to maximize performance, 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 [4], [12], [14] rely on hot code prediction. In this case the VM is typically uses 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 moment that 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 a code region is flagged as hot, the VM switches to a second stage. In this stage, the VM constructs a new code region (e.g. a trace [4], [5], [8], [14]) containing the hot code and optimize this region by doing an optimizing translation or further optimizing previously translated code. After optimized the code is stored in a code cache for future re-execution.

A widely used approach to predict hot code is based on execution frequency thresholds [2], [4]–[8], [14], [16]. This predictor is very simple: it flags a region as hot if and only if it reaches a fixed execution frequency threshold, which we refer to as T_P. The rationale behind it is: infrequently executed code regions do not reach the threshold. However there are occasions that a region have an actual accumulated execution count sufficiently high to reach the prediction threshold but its final execution count is not high enough to compensate for the compilation overhead. We call such regions **warm code** and when the predictor flags a warm code region as hot we say that a **hot code misprediction** occurred. From now on we will call this predictor the threshold-based predictor (or simply TBP), and its behavior and how mispredictions affect the performance of its corresponding VM are subject of study in this work. The focus of this study is cross-ISA VMs (a cross-ISA VM is one where the guest and host interfaces are different), since on same-ISA VMs the translation cost may be negligible because in general only a copy of the guest code is done to "emulate" it.

Besides being used on several Cross-ISA VMs, such as Transmeta CMS [12], HP Dynamo [4], Mojo [6], IA32-EL [5], StarDBT [8], Aries [7] and others, we show that the TBP's hypothesis is not sufficiently strong to be used in VMs that have a workload composed of large code footprint applications. Specifically, we show that for a wide range of VM configurations the **maximum** overhead due to mispredictions is no greater than 10% when the VM is executing workloads from SPEC CPU 2006 [17], however when the VM is executing applications from Sysmark 2012 [18], our experiments indicate

a **minimum** overhead of 27%.

The contributions of this work are as follows:

- An analytical model to estimate and characterize the misprediction overhead of threshold-based hot code predictors (Section III).
- A characterization of the emulation overhead due to hot code misprediction in a VM executing three different workloads (Figures 5 and 6).
- We show that as the VM employs more cycles to translate/optimize regions the misprediction overhead sharply increases (Figures 3 and 4).
- 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 overhead caused by mispredictions (Figure 3).

This text is organized as follows. In Section II we present a motivation for this work. Section III presents an analytical model to quantify and characterize hot code misprediction overhead. Section IV presents the methodology we used in our experiments. In Section V we present our results for hot code misprediction overhead for a VM emulating SPEC CPU 2006, Sysmark 2012 and three operating system boots. In Section VI we present related works. Finally, in Section VII we present our conclusions.

II. Motivation

Performing translation or heavy optimizations on cold/warm code may add a significant overhead to the emulation process as their execution frequency may not be sufficiently high to amortize the translation/optimization cost. This is typically the case when the hot code predictor misses the prediction.

In the following example, we assume a two-staged VM that employs interpretation to emulate cold code and Dynamic Binary Translation (DBT) to emulate hot code. The translator produces faster emulation code than the interpreter, however it employs an expensive process. 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 I, the interpreter of this virtual machine (second column) would take 250 cycles to emulate an instruction that executes 5 times (first row) and 10,000 cycles to emulate an instruction that executes 200 times (second row), 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.	**In. Cost**	**Tr. Cost**	**Cb. Cost**	**Pr. Cost,** $T_P = 6$
5	5×50	1000 + 5	5×50	5×50
200	200×50	1000 + 200	$1000 + 200$	$300 + 1194$
Total	10250	2205	1450	1744

Table I: Example of emulation and misprediction overhead.

An even 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 to use only interpretation and 1.5 times faster than to translate both instructions. According to this example the best emulation technique for each code segment depends on the number of times the region executes, and combining different techniques may allow us to reduce the emulation cost. 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 (5×50) cycles interpreting the instruction that executes 5 times and 300 (6×50) cycles interpreting plus, 1194 cycles translating and executing the translated code for the instruction that executes 200 times. The total emulation time, 1744 cycles, is 20% higher than the combined 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. This way from the VM designer perspective it is important to use a prediction threshold (T_P) as smaller as possible (to predict hot code sooner), however the TBP is an heuristic and this way decreasing T_P will increase the possibility of mispredictions happen. Nevertheless, independent of which threshold value is used there is always the possibility that the execution count for some code region be only slightly larger than T_P and in such cases the predictor will mispredicts the code as hot, causing extra overhead due to translation or optimizations of cold code. As an example consider that the VM designer sets $T_P = 5$. 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 combined case. Notice that the predictor mispredicted the first instruction as hot, however the time spent interpreting the hot instruction (second one) has decreased. One contribution of this work is that independent of the T_P used the misprediction overhead can considerably high (see Figure 5).

III. Threshold-based hot code misprediction overhead

In this section we formalize several aspects related to hot code prediction. Initially, we present functions to characterize the 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.

A. The Emulation Cost

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

$$C_I(n) = \alpha_I + \beta_I n \tag{1}$$

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

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

$$C_T(n) = \alpha_T + \beta_T n \tag{2}$$

where constant α_T represents the non-recurrent cost to translate (compile), optimize and cache instruction I, and constant β_T is the cost paid each time the translated code is executed to emulate the instruction I.

As we will see in Section IV, the parameters α_I, β_I, α_T and β_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 defines the point where translation have a lower cost than interpretation:

$$\begin{aligned} C_T(n) &< C_I(n) \\ \alpha_T + \beta_T n &< \alpha_I + \beta_I n \\ \beta_T n - \beta_I n &< \alpha_I - \alpha_T \qquad (-1) \\ \beta_I n - \beta_T n &> \alpha_T - \alpha_I \\ n(\beta_I - \beta_T) &> \alpha_T - \alpha_I \\ n &> \frac{\alpha_T - \alpha_I}{\beta_I - \beta_T} \end{aligned} \tag{3}$$

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 DBT, 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 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.

B. 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:

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

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:

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

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

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

where T_P is the prediction threshold and I_n is 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:

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

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.

C. Hot Code Misprediction Overhead

Figure 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 α_T, β_T, and α_I, β_I, the value computed for T_H (from Equation 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

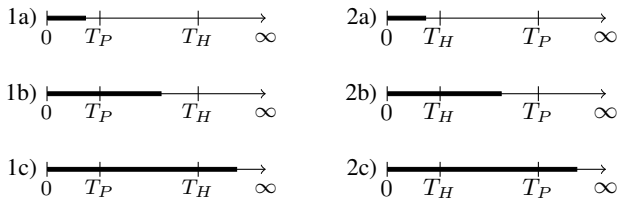


Figure 1: Prediction scenarios for the TBP. Black bar shows final instruction frequency.

cases 2(a-c) cover the scenarios for which T_H is smaller than T_P.

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 $n \leq T_P$ 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:

$$Overhead_{total} = \frac{\sum_{I \in B} Cost_{Thr}(I)}{\sum_{I \in B} Cost_{Ora}(I)} \quad (4)$$

We have used this model to characterize the overhead of several SPEC2006 [17], Sysmark 2012 [18] benchmarks and operating systems boot processes. In the next section we describe the methodology we used to estimate the parameters (α_I, β_I, α_T and β_T) based on this model.

D. 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-geared/threaded virtual machines are more common for high level languages (e.g: Java [19]) and are an exception for ISA level VMs, 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.

IV. 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, α_I, β_I, α_T, and β_T (Equations 1 and 2), we used the Bochs emulator (version 2.5.1) [15] and the Low Level Virtual Machine (LLVM) (version 3.0) [20]. The Bochs x86 system emulator was used to collect the execution frequencies of instructions of the benchmarks and to estimate the parameters α_I and β_I. LLVM was used to perform the compilation of several code fragments from which we estimated the translation cost of an instruction, α_T. Finally, the translated instruction emulation performance, β_T, was estimated by running compiled code generated by LLVM.

Notice that we are not uniques in using a combination of tools to start up our VM environment. HQEMU [21] and Harmonia [22] are examples or modern VMs that use QEMU [23] together with LLVM [20] 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.

A. The Model Input: Instructions Execution Frequency

We used the Bochs infrastructure to profile the execution frequency for instructions of the SPEC CPU2006 [17], Sysmark 2012 [18], and the boot process of Windows 7, Windows XP, and Debian 5 Linux. Our instrumentation routines uses a hash table indexed with a combination of an instruction linear address plus first byte of its opcode to keep track of instruction frequency.

Notice that Sysmark frequently restarts the computer in order to ensure 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 configuration, 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 VMs were configured with 2GB of RAM and a virtual hard disk of 15GB. Sysmark benchmarks used Windows 7 and SPEC CPU2006 used Debian Linux.

B. Model's Parameters: Interpretation Cost

Estimating interpretation start-up cost α_I and steady-state cost β_I: Bochs is known for its high portability and mature code base, an ideal candidate for characterization of a VM that uses interpretation as its emulation technique.

To measure the cost of instruction pre-decoding and interpretation, α_I and β_I, we changed Bochs to report the number of pre-decoded instructions, 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 [24]. The ratio of the number of *x86 cycles* spent in pre-decoder routines over the number of *pre-decoded instructions* gives α_I. The ratio of the number of *x86 cycles* spent in the interpreter routines over the number of *interpreted instructions* gives β_I. Each benchmark yields a different α_I and β_I, providing a range of reasonable α_I and β_I values for our model.

C. Parameters of the Model: Translation Cost

LLVM was chosen to estimate the translation cost parameters because it can be seem as a powerful VM that translates LLVM bitcodes into host binaries. LLVM bitcode is a low-level program representation that is close to machine instructions and has its own ISA, the LLVM virtual ISA [20]. 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 α_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 VM 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.

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

Estimating translation steady-state cost β_T:

To estimate β_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 β_T parameter is then estimated by the ratio of *x86 cycles* over the number of *LLVM instructions*. Each benchmark yields a different β_T, providing a range of reasonable β_T values for our model.

Variability: β_T can change depending on the optimizations used to generate the LLVM bitcode guest executable and the x86 native executable.

The more optimized is the guest program, the higher is β_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.

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

We measured β_T using an optimized LLVM bitcode ("-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.

Integer Benchmarks				Floating Point Benchmarks			
Bench.	β_I	α_T	β_T	**Bench.**	β_I	α_T	β_T
perlben.	48	91952	1.02	lbm	85	41315	2.11
bzip2	40	94429	1.40	sphinx3	117	60468	1.31
gcc	48	72546	1.58	milc	141	102610	3.50
mcf	42	26652	3.18	namd	144	96285	1.51
gobmk	48	66671	1.83	dealII	64	177565	–
hmmer	95	53780	1.47	soplex	65	148678	2.80
sjeng	40	89039	1.56	povray	92	73341	2.25
libquan.	31	115724	1.64	zeusmp	122	–	–
h264ref	40	56832	1.29	gromacs	173	–	–
omnetpp	59	175220	5.28	cactus	209	–	–
astar	44	110029	2.48	leslie3d	213	–	–
xalan	38	173994	4.38	calculix	128	–	–
-	-	-	-	gems	133	–	–
-	-	-	-	tonto	94	–	–
-	-	-	-	wrf	108	–	–
-	-	-	-	bwaves	103	–	–
-	-	-	-	gamess	106	–	–

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

V. Results

In this section 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.

During the experiments three Sysmark 2012 scenarios did not complete their execution and we chose to omit their partial results. Also, for all experiments in this section 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, we only use these values to determine a range of values to be considered in our experiments, see Table III, and not to model a VM built on bases of Bochs and LLVM. Therefore the results we show in this section 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 (α_T and β_T) are out of this range, it is reasonable to expect that the trends shown in the graphs will not change.

A. Estimation of the Model Parameters β_I, α_T and β_T

Steady-state interpretation cost β_I:

Column β_I of Table II 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 β_I varies from 30 to 220 host cycles.

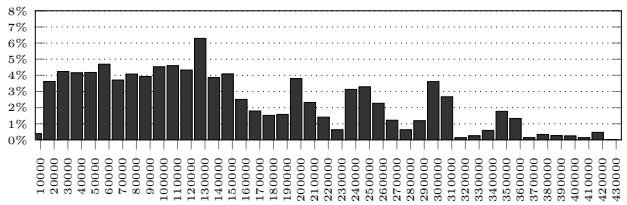


Figure 2: Histogram showing the percentage of total SPEC CPU2006 functions sharing a given cost range (α_T), in cycles. The Abscissa represent the cost to translate the function.

Start-up translation cost α_T:

Column α_T of Table II shows the average cost, in cycles, to translate each LLVM bitcode 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.

Our goal in measuring α_T is to determine the minimum number of cycles per instruction, using an LLVM-based VM, to translate the program fragment. Since with greater α_T, as we will see, the overhead is even greater, thus we are estimating a lower bound. Nevertheless, we also provide information on a VM geared at optimizing code.

For each one of the 71261 C and C++ SPEC CPU2006 functions compiled, we calculated α_T using the ratio of the number of cycles to compile the function over the number of LLVM instructions in the function. Figure 2 presents a histogram where each bin shows the percentage of functions in a given range for α_T, considering a first scenario where no optimizations are enabled. That is, the Abscissa represents different costs to translate a function while the Ordinate represents the percentage of functions which had the translation cost between two adjacent (to the left) ticks of the x-axis.

The histogram confirms that α_T varies significantly, depending on several parameters 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 α_T. On the other hand, functions with a single instruction often have the highest α_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 α_T greater or equal to 36,000 cycles. The average α_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 II 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 α_T was 1,073,000 cycles. In the next

Parameter	Start	End
Prediction Threshold (T_P)	25	3000
Interpretation (β_I)	30	220
Compilation (α_T)	30,000	850,000
Execution (β_T)	0.5	3.0

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

section, we present a study using α_T in the range of 30,000 cycles up to 850,000 cycles because we focus on the fastest cases with respect to both scenarios. For greater values, the misprediction overhead is even bigger.

Steady-state translation cost β_T: Column β_T of Table II shows the average cost, in cycles, for emulating each LLVM bitcode 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 Section IV, there are two scenarios for measuring β_T, but Table II shows only the first and more important one, namely the one for a VM that does not apply optimizations. In this case, the generated code quality is poorer and the average β_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 β_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 β_I, α_T and β_T parameters, which are summarized on Table III.

B. Misprediction Overhead

Figures 3a-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 (α_T), and the translation steady-state cost (β_T). For all points in these graphs, the parameters β_I and T_P, are unconstrained inside their respective ranges (Table III). 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 (α_T) is 300,000 host cycles (per translated instruction) and the steady-state cost (β_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.

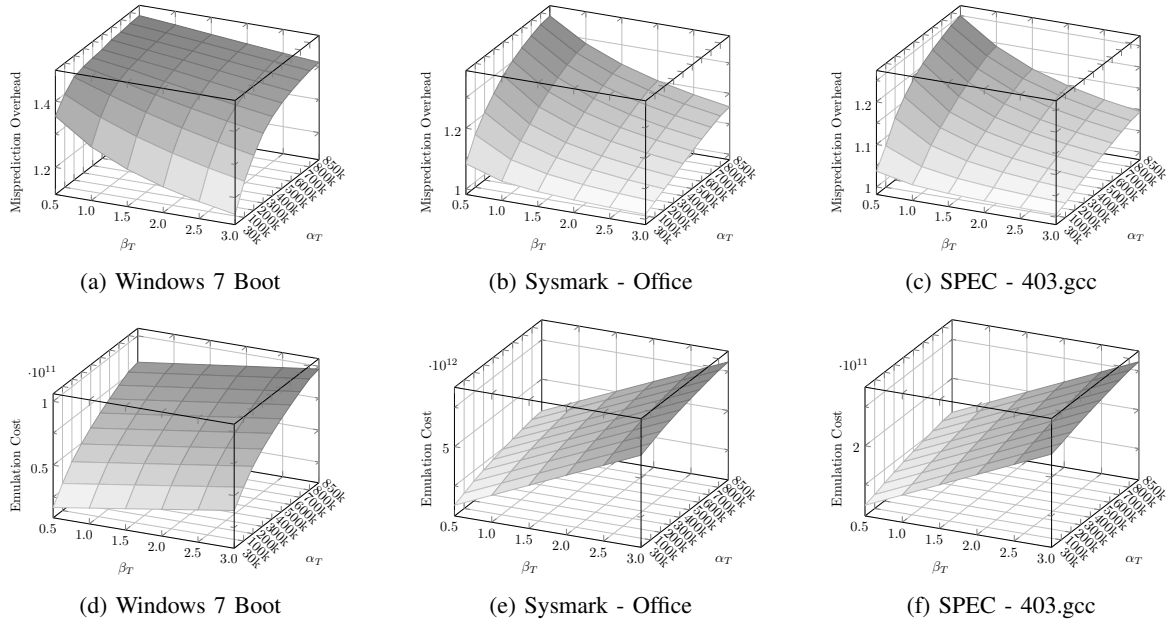
Notice that the three figures have the same pattern, and, in fact, all benchmarks we experimented present this same behavior. As it can be seen, if the compilation cost α_T increases, the misprediction overhead also increases. This is an intuitive trend, since the main source of overhead of the TBP in 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 (β_T) increases. This happens because the greater is the steady-state execution cost, the longer is the execution time, causing the misprediction overhead to become a smaller fraction of the total emulation time.

Figures 3d-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 smallest values of α_T and β_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 4 shows an example of how the development of a VM can be severely affected by the misprediction overhead. We use our measured values for β_T and α_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 β_T and α_T of our experiments. The second case considers the average measured values, as described in Section IV. The figure shows the trajectory on the overhead surface when the VM progressively supports more sophisticated optimizations and better code quality is generated. We presented a range of α_T values in this work, but here, two trend lines are presented. The first uses our measured average values and the second uses α_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 VM 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.

C. Misprediction Overhead Characterization

Figure 5 shows the misprediction overhead of all three sets of benchmarks if we consider a system that can produce a reasonable 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 $T_P = 1,000$. $T_P = 1000$ was the value of the threshold that resulted in the smallest misprediction overhead given this configuration. 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.



(a) Windows 7 Boot (b) Sysmark - Office (c) SPEC - 403.gcc

(d) Windows 7 Boot (e) Sysmark - Office (f) SPEC - 403.gcc

Figure 3: Misprediction and Emulation Overhead for Sysmark 2012 (Office), SPEC CPU2006 (403.gcc) and Windows 7 Boot.

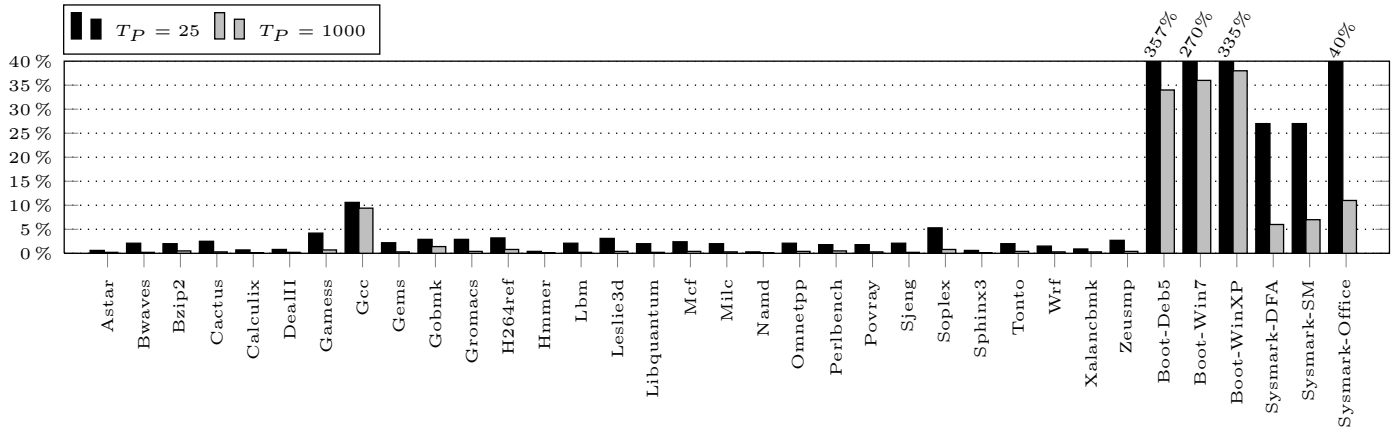


Figure 5: Maximum misprediction overhead for SPEC CPU2006 and minimum misprediction overhead for several OS boots and Sysmark 2012 benchmark.

The boot processes and interactive applications (such as those of Sysmark 2012) 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 also 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 6 the misprediction overheads quantified in Figure 5, in terms of two components: the **Warm Code** and **Hot Code** overhead, which refers to cases 1b and 1c of Figure 1, respectively. We notice that due to the parameters we used, the cases 2(a)-(c) of Figure 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

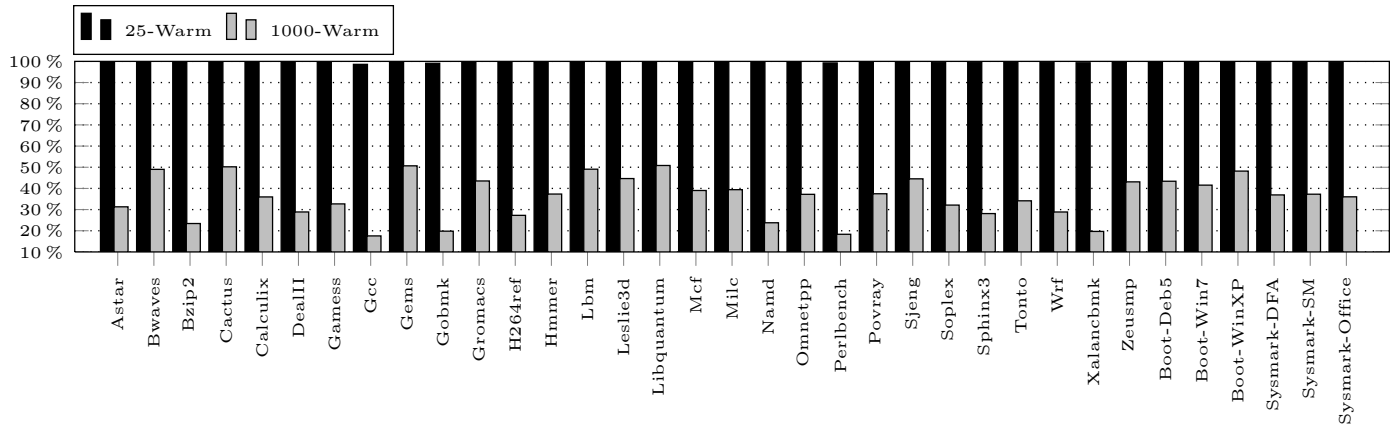


Figure 6: Hotness misprediction overhead split in two components: warm and cold code translation overhead. The bars show the warm code translation overhead when $T_P = 25$ and $T_P = 1000$, they complement represent cold code translation overhead.

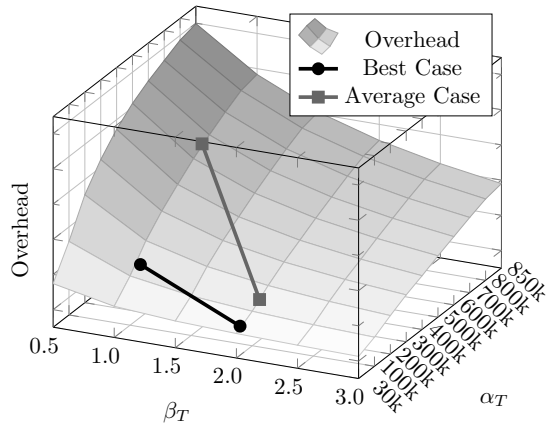


Figure 4: Trend lines for the evolution of VM in terms of generated code quality and their consequent misprediction overhead.

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.

VI. Related Work

Bala *et. al* [4] describe Dynamo, a software dynamic optimization system that is capable of improving the performance of a HP PA-800 instruction stream as it executes. To attain this, Dynamo interprets the guest code until a start-of-trace criteria is met, at which point it employs a technique called MRET [16] to form and optimize a trace. The MRET is a classic example of an implementation of the concepts of the TBP. They evaluate the system using Spec-Int-95.

The IA-32 EL [5] is a dynamic binary translator that enables the execution of IA-32 applications on Intel Itanium processors. Initially, the application code is translated on a basic-block basis using minimal set of optimization and instrumentation code is used to detect hotspots. When the instrumentation counter of a basic block reaches a prefixed threshold it is marked as a candidate hotspot. After many basic blocks are marked candidates the system form a superblock that will be further optimized and cached. They used SPEC CPU 2000 to measure the system performance.

StarDBT [8] is a multi-platform research binary translator capable of translating x86 32/64 bits applications to IA 32 bits binaries. StarDBT uses a simple/fast translator for cold code translation and once a workload hotspot is detected, it forms a trace optimize and cache the trace. The system is evaluated using SPEC CPU 2000 and Sysmark 2004 suites. Results show that the system runs comparatively well with other state-of-the-art binary translator, however for large interactive Windows applications the overhead can be considerably high. They argue that optimizing infrequently executed code regions causes the overhead.

The FX!32 [14] is a VM that enables transparent execution of x86 32 bits Windows NT applications on an Alpha host 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. Since the translation is performed offline the system does not suffer from hotness misprediction overhead.

Despite the wide spectrum of dynamic binary translators, just a few papers [25]–[28] are focused on characterizing the overhead of such systems in a production environment.

Borin and Wu [26] study the overhead of the Intel research dynamic binary translator StarDBT when emulating the SPEC CPU 2000 benchmarks [29]. They found that branch handling and code duplication represent more than 64% of the StarDBT overhead and cold code translation and hot trace building together account for 34% of the DBT overhead.

Hu and Smith [27] use a Co-Designed VM to study the overhead of an adaptive DBT system. They uses a two-phase DBT that performs simple basic block translation to initial emulation, and a superblock optimizer for emulating hotspot code - detected when the execution frequency reaches a prefixed threshold. Their results show that emulation of cold

code contributes to a major part of the VM overhead and they propose two hardware mechanisms to minimize the first phase interpretation cost.

Following the same direction as Hu and Smith research is the work of Chen et al. [25]. 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. To mitigate the problem they propose the utilization of a Decoded Instruction Cache (DICache).

Wu et al. [28] use IA-32 EL to investigate the accuracy of the initial profiling in two-phase dynamic binary translators. Their results for the SPEC 2000 benchmark indicate that a retranslation threshold on the range of 500 to 2000 can have prediction accuracy comparable to the traditional profile-guided optimizations using training input. However several benchmarks show phased behavior and a single profiling phase does not capture the average program behavior accurately for these benchmarks continuous profiling or multiple profiling phases are required.

VII. Conclusions

In this work, we used several workloads to measure the effectiveness and quantify the overhead of the threshold based predictor (TBP), a widely used hot code predictor on virtual machines. We first developed an analytical model to estimate the TBP misprediction overhead. Then, we performed a comprehensive evaluation of the TBP using SPEC CPU 2006, Sysmark 2012 and the boot process of three different operating systems. Our results indicate that, even though the TBP can accurately predict hot code on SPEC CPU 2006 benchmarks, it does not perform well on applications with large code footprint and cause at least 27% of overhead on Sysmark 2012 applications. This result suggests that using "only" SPEC SPU 2006 to analyze virtual machines performance may lead to incorrect conclusions. We also explored several VM design points and showed that, as the VM spends more cycles optimizing the translated code, the overhead due to hot code mispredictions becomes more relevant. Finally, we decomposed the misprediction overhead and showed that its major component is due to translation of warm code and that this overhead can be exacerbated in applications with large code footprint, such as those found in interactive environments.

References

[1] J. Smith and R. Nair, *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann, 2005.

[2] 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*, vol. 21665, 2000.

[3] K. Ebcioğlu and E. R. Altman, "DAISY: dynamic compilation for 100architectural compatibility," ser. ISCA'97, 1997.

[4] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," ser. PLDI'00, 2000.

[5] 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," ser. MICRO'36, 2003.

[6] W. Chen, S. Lerner, and R. Gillies, "Mojo: A dynamic optimization system," *FDDO-3*, 2000.

[7] C. Zheng and C. Thompson, "PA-RISC to IA-64: transparent execution, no recompilation," *Computer*, vol. 33, no. 3, pp. 47 –52, 2000.

[8] E. Borin, C. Wang, Y. Wu, and G. Araujo, "Software-based transparent and comprehensive control-flow error detection," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '06, 2006.

[9] J. Moreira, D. César, G. Araújo, E. Borin, and S. Rigo, "Asynchronous program flow verification through binary instrumentation on QEMU," ser. AMAS-BT'12, 2012.

[10] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "LIFT: A low-overhead practical information flow tracking system for detecting security attacks," ser. MICRO'39, 2006.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," ser. PLDI '05, 2005.

[12] 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," ser. CGO'03, 2003.

[13] C. Cifuentes and M. Van Emmerik, "UQBT: adaptable binary translation at low cost," *Computer*, vol. 33, no. 3, 2000.

[14] R. J. Hookway and M. A. Herdeg, "Digital FX!32: Combining emulation and binary translation," *Digital Technical Journal*, vol. 9, no. 1, 1997.

[15] O. Source, "Bochs - the cross platform IA-32 (x86) emulator," http://bochs.sourceforge.net/, 2011, [Accessed November 11th, 2012].

[16] E. Duesterwald and V. Bala, "Software profiling for hot path prediction: less is more," *SIGPLAN Not.*, vol. 35, no. 11, 2000.

[17] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Computer Architecture News*, 2006.

[18] B. Co., "Sysmark 2012 suite."

[19] 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*, vol. 39, no. 1, 2000.

[20] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," ser. CGO'04, 2004.

[21] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung, "Hqemu: a multi-threaded and retargetable dynamic binary translator on multicores," ser. CGO '12, 2012.

[22] 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," ser. CF '11, 2011.

[23] F. Bellard, "QEMU - Quick EMUlator," http://www.qemu.org/, 2012, [Accessed November 11th, 2012].

[24] *IA-32 Intel Architecture Software Developer's Manual*, Volume 2: Instruction set reference ed., Intel Corporation.

[25] W. Chen, D. Chen, and Z. Wang, "An approach to minimizing the interpretation overhead in dynamic binary translation," *The Journal of Supercomputing*, vol. 61, no. 3, 2012.

[26] E. Borin and Y. Wu, "Characterization of DBT overhead," ser. IISWC'09, 2009.

[27] S. Hu and J. E. Smith, "Reducing startup time in co-designed virtual machines," ser. ISCA'06, 2006.

[28] Y. Wu, M. Breternitz, J. Quek, O. Etzion, and J. Fang, "The accuracy of initial prediction in two-phase dynamic binary translators," ser. CGO'04, 2004.

[29] J. L. Henning, "SPEC CPU2000: measuring CPU performance in the new millennium," *Computer*, 2000.