Accurate Off-Line Phase Classification for HW/SW Co-Designed Processors

Aleksandar Branković[†], Kyriakos Stavrou[§], Enric Gibert[§], Antonio González[†§]
[†]Universitat Politècnica de Catalunya, Spain
[§]Intel Barcelona Research Center, Intel Labs Barcelona, Spain
abrankov@ac.upc.edu
{kyriakos.stavrou, enric.gibert.codina, antonio.gonzalez}@intel.com

ABSTRACT

Evaluation techniques in microprocessor design are mostly based on simulating selected application's samples using a cycle-accurate simulator. These samples usually correspond to different phases of the application stream. To identify these phases, relevant high-level application statistics are collected and clustered using a process named "Off-Line Phase Classification". The purpose of phase classification is to reduce the number of samples that need to be simulated with the minimum loss in accuracy (compared to simulating the complete set of samples).

Unfortunately, when directly applied to HW/SW co-designed processors[1] the traditional phase classifications do not provide a good trade-off between accuracy and the number of samples. As an example, according to our experimental results, to achieve a 4% error (compared to simulating all the samples) one needs to simulate 2.5X more samples for the case of HW/SW co-designed processors compared to what is necessary for HW-only processors.

In this paper, we propose a novel off-line phase classification scheme called TOL Description Vector (TDV), which is suitable for HW/SW co-designed processors. TDV targets at estimating the TOL particularities and on average gives significantly better accuracy than traditional phase classification for any number of selected samples. For instance, TDV reaches the average error of 3% with 3X less samples than traditional classification. These benefits apply for different TOL and microarchitecture configurations.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Design studies

General Terms

Performance

Keywords

Dynamic Binary Translation, HW/SW Co-Designed Processors, Simulation, Warm-Up Methodology

1. INTRODUCTION

HW/SW co-designed processors are equipped with a software layer that dynamically analyzes, profiles, translates and optimizes instructions from a guest ISA to an internal microarchitecture with its own customized host ISA. Such a software layer sits below the OS and it is totally transparent to the entire software stack. This layer is often organized as a staged compiler, in which lower optimization stages consist of an interpreter or a fast translator for individual instructions. Code regions are formed and promoted to higher and more aggressive optimization stages as they become hotter. Optimized code regions are stored in a code cache and most code is fetched and executed from it in the steady state. While this software layer has received many different names (e.g. Code Morphing Software in Transmeta designs [8, 19]), we refer to it as the Transparent Optimization Layer or TOL throughout the rest of this paper. Splitting the processor design into the two components aims to pursue better performance, better power consumption, lower design complexity, better design customization, backward / forward ISA compatibility or a combination of them [8, 10, 9, 15, 19, 20, 24, 25, 27, 28, 31, 33].

Due to this HW/SW collaborative combination in such systems, designers need to decide what part of a particular performance / power / compatibility feature is implemented in hardware and what part is implemented in software. Solutions range from an entire hardware implementation to an entire software implementation, with many of them requiring support from both components. Examples of techniques that require support from hardware and software include, but are not limited to, the dual-address return address stack [18] or flook stack [21] to handle subroutine calls and returns, memory disambiguation support for data speculation optimizations [8], the link pipe to handle indirect branches [21] and hardware support to accelerate the execution of particular TOL components as the instruction decoder [15] or the optimizer [27]. These solutions may also require the addition of new host ISA instructions in order to define the interface between the internal hardware and the Transparent Optimization Layer (TOL). In addition, the decision on what

[1]The scope of this work regards designs similar to Transmeta's Efficeon [20], not on ASIPs or others. We use the term *HW/SW co-designed processors* following the taxonomy provided by Smith and Nair [30]. They specifically call them "*HW/SW co-designed virtual machines*", but we found that the word "processor" is less confusing.

part is implemented in hardware and what part in software may also depend on the details of the microarchitectural implementation, such as the configuration of the memory hierarchy or the design of the branch predictor. In order to evaluate these trade-offs in a reasonable amount of time an accurate but fast simulation methodology is required.

In order to speed up the simulations, computer architects use sampling [29], a technique where only few application samples are selected as representatives of the workload whereas the rest of the samples are skipped. Each of these selected samples should represent a different phase of the application. The process that classifies the similar phases is based on statistics which do not require cycle-accurate simulation and thus it is called *Off-Line Phase Classification*. In the case of HW-only processors the statistic typically used is the basic block (BB) execution frequency [29], and the phase classification is called Basic Block Vector (BBV) phase classification. BBV is based on the assumption that each basic block of a given application behaves similarly each time executed.

In this paper we analyze BBV for the case of HW/SW co-designed processors. Due to the nature of these processors (the code regions are dynamically translated and they can be executed in different optimization stages), the execution time of each basic block has a significantly wider variance compared to HW-only processors. In order to back up this statement, Figure 1 compares the range of execution time in terms of number of cycles for a basic block in 400.perlbench application for HW-only and HW/SW co-designed processor. The results clearly indicate that the execution time varies significantly more for HW/SW co-designed processors. More specifically, for HW/SW co-designed processors the execution time for this basic block ranges from 20 to 1000 cycles, whereas for HW-only processors it ranges from 15 to 100 cycles. Consequently, BBV phase classification may classify dissimilar samples as similar in the case of HW/SW co-designed processors, because the assumption that each basic block behaves similarly each time executed is not valid. On the other hand, in the case of HW-only processors each basic block behaves similarly almost always.

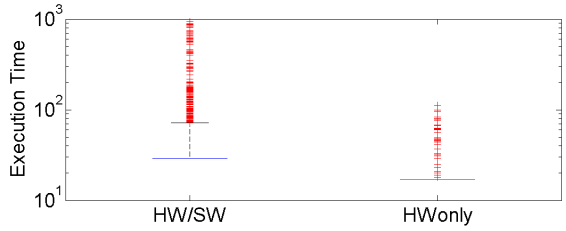

Figure 1: Execution time boxplot chart of the one BB (0x80c6d87) in 400.perlbench in two different architectures: HW-only and HW/SW co-designed.

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

In particular, the main contributions of this paper are:

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

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

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

The rest of the paper is organized as follows: Section 2 discusses the related work, while Section 3 gives the necessary background. Section 4 introduces the TDV phase classification. Experimental methodology is described in Section 5, while evaluation results are presented in Sections 6. Finally, Section 7 summarizes the paper.

2. RELATED WORK

Off-line phase classification schemes have been widely studied for HW-only architectures. The authors in [16] and [22] list, describe and compare these techniques. In all of these classifications different high-level statistics are used to describe the similarity of samples: instruction mix, loop detection, memory access addresses etc. However, the traditional phase classification is based on the clustering using the Basic Block Vector (BBV). Recent studies show that BBV does not behave well in the presence of frequent L2 misses [3, 7]. What differentiates our work from the previous contributions is that we perform phase classification analysis for HW/SW co-designed architectures, where the effects of the dynamically execution and the staged compilation cause different application behavior.

The phase behavior of JIT compilers was also well targeted in the literature. The main issues are however different. In particular, these papers try to identify on-line phase changes [23, 11, 13]. JIT compilers use phase change information to improve the performance of the system. On the other hand, in this paper we are focused on the off-line phase classification. This problem is not relevant to the current JIT compilers, since research in this area is typically preformed using real hardware and not simulators, like in the case of HW/SW co-designed processors.

Sampling methodologies have also been widely researched in the past. SimPoint [29] is most probably the most dominant technique according to which the most representative samples are determined by off-line phase classification. Besides SimPoint, there are other sampling methodologies such as Smarts [32, 12] and Cotson [4]. Smarts sampling methodology considers the architecture as a black box and does not analyze samples' similarity. It simply assumes random sampling and only determines the parameters for this random sampling. On the other hand, similar to SimPoint, Cotson simulates only phases, but it predicts them based on the on-line classification, using the internal simulation events.

However, it is not clear how these internal simulation events apply in the case of HW/SW co-designed architectures. In this paper we will use SimPoint as our baseline as it is the most used technique.

3. BACKGROUND

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

3.1 SimPoint

To speedup the simulation process researchers typically use only few samples of a particular application, instead of simulating the whole application. The samples are usually chosen such that they present dissimilar phases [29]. The approach followed by SimPoint and is illustrated in Figure 2. Across all the samples, only n samples, out of M, are selected for the simulation.

Figure 2: Sampling simulation approach. Only n samples (out of M) are selected for cycle-accurate simulation.

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

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

$$\Sigma_{i=1}^{n} \alpha_i = 1, \quad (1)$$

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

$$CPI_{real} = \Sigma_{i=1}^{M} (1/M) \cdot CPI_i^{sample}. \quad (2)$$

Selected samples $(1, 2, ..., n)$ are usually chosen as the ones which show the most phases' dissimilarity between each other and they are weighted with different coefficients (α_i) depending on their contribution. The similarity / dissimilarity between the samples is estimated by employing the *off-line phase classification*. The entire process is illustrated in Figure 3.

Off-line phase classification is the process of selecting similar samples and does not require cycle-accurate simulation results. It predicts which samples will have similar cycle-accurate statistics between each other analyzing only the microarchitectural independent statistics (*e.g.* instruction mix, BB execution frequency etc.), as depicted in Figure 3-a. These statistics are usually called *high-level binary* statistics and the vector which contains them is attached to every sample. For instance, in Figure 3-a CPI, D$ miss rate, I$ miss rate (which are microarchitectural dependent statistics) are mapped to number of the integer instructions, number of the loads and number of the stores (which are microarchitectural independent statistics).

After collecting the high-level binary statistics across all samples, the phase classification algorithm is applied in order to identify the samples that are similar (Figure 3-b). Computer architects usually use the algorithm which contains the following phases: normalization, dimensionality reduction, clustering and representative members choosing (Figure 3-b).

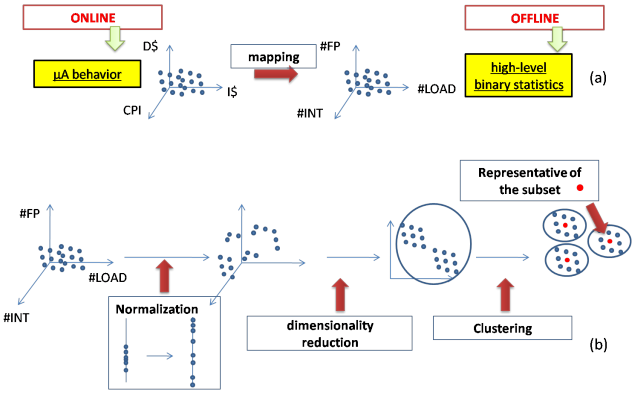

Figure 3: Phase Classification process in computer architecture.

Normalization is the process of shrinking or expanding each dimension of the off-line vector to the range [0, 1]. This is done in order to avoid favoring one dimension over the others. Figure 3-b depicts this process as compacting the values to the range [0, 1].

The next step is *Dimensionality Reduction*. This is needed to limit the execution time of the clustering algorithm itself; the initial dimensionality of the off-line vector can be in the order of thousands. In this paper we use Random Linear Projection (RLP), the same scheme as the one used by SimPoint. In Figure 3-b this is illustrated as reducing the dimensionality from 3 dimensions to 2 dimensions.

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

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

3.2 BBV Phase Classification

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

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

$$CPI^{sample} = \Sigma_{bb=1}^{N} \omega_{bb} \cdot CPI^{bb}, \quad (3)$$

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

$$BBV = [\omega_1, \omega_2, ..., \omega_N]. \quad (4)$$

However, for the cases with a significant number of non-deterministic long latency events (like for example L2 misses) CPI^{bb} varies widely across executions [3, 7]. Consequently, BBV phase classification in this case will give inaccurate results.

Similar to that, in the case of HW/SW co-designed architectures, the execution of a particular BB will not be similar every time. The main reason is the overhead cycles which correspond to TOL execution. Such an overhead is 2-3 orders of the magnitude higher than the highest penalties in HW-only architectures (*e.g.* L2 cache miss). Moreover, this penalty is not constant and totally unpredictable [6], so it requires additional off-line statistics to express it.

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

TOL Look Up task is the main reason for non-similar execution time of a specific basic block. This is illustrated in Figure 4. Imagine three basic blocks: BB1, BB2 and BB3 such that the branch BB1→BB3 is a direct branch, while the branch BB2→BB3 is an indirect branch. Imagine also that the translations of these basic blocks are already stored into the code cache. In this example we are focused on the execution time of BB3. In the cases when control flow goes from BB1 to BB3, the execution time of BB3 contains only the code cache execution time. On the other side, when control flow goes from BB2 to BB3, the execution time of BB3 contains the code cache execution time plus the *TOL Look Up* execution time, since BB2→BB3 is an indirect branch. For this reason BB3 will not have the same execution time. Although this example is related to indirect branches, similar behavior can be observed in the cases of indirect calls and returns.

In this paper we have also studied *Path Profiling*, the most accurate profiling. This approach is similar to BBV, with the only difference that instead of the contribution of the basic blocks we profile the contribution of the each path. In theory path profiling should give better sampling accuracy since it can distinguish scenarios which BBV cannot. For example, the execution stream *a-b-c-a-b-c-a-b-c* is different than the execution *a-a-a-b-b-b-c-c-c*, where *a*, *b* and *c* are different BBs. However, from BBV's perspective these two executions are the same. Although the path profiling is a promising solution, we found out that it gives only slightly better accuracy than BBV but not better than our solution. The main reason is that path profiling cannot express the TOL behavior accurately. Similar to BBV, paths $p1$ and $p2$ can be similar, but the TOL execution can be different among these two paths [6]. Moreover it requires bigger profiling effort than BBV.

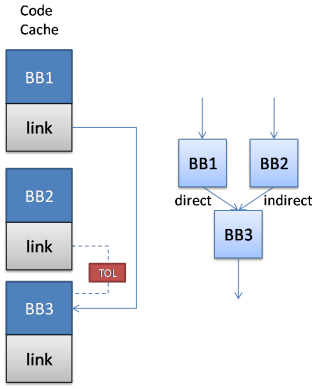

Figure 4: Example which illustrates the main reason for high CPI variance per BB in HW/SW co-designed architectures.

4. TOL BASED PHASE CLASSIFICATION

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

4.1 The contents of TDV

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

$$TDV = [sd_1, sd_2, sp_0, sp_1, ..., sp_9, ind, im_1, im_2, ..., im_5]. \quad (5)$$

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

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

$$cycles^{sample} = cycles_{LUP} + cycles_{Transl.} + cycles_{CodeCache}, \quad (6)$$

where $cycles_{LUP}$, $cycles_{Transl.}$ and $cycles_{CodeCache}$ stand for the cycles spent during the *Look Up*, *Translation* and

Code Cache execution respectively. Equation 6 can be written similar to Equation 3, in order to express CPI^{sample}:

$$CPI^{sample} = CPI_{LUP} + CPI_{Transl.} + CPI_{CodeCache}. \quad (7)$$

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

4.1.1 Look Up estimation

Based on the previous studies [26], the most important contributor to the TOL overhead is the ***Look Up (LUP)***. LUP overhead is needed to guarantee forward progress of the translated application. In particular it does look up in a table of translated code regions in order to find the starting address of the code region in the code cache. As we discussed in Section 3.2, the absence of the linking between the code regions in the code cache is the reason for LUP. Among many sources of the possible cases in which the linking is not possible, the branch indirection is predominated. Thus LUP is estimated by the number of the indirect branches, indirect calls and returns (*ind* field in Equation 5).

4.1.2 Translation estimation

The second TOL task is ***Code region translation and optimization***. The TOL invokes this process whenever a block is encountered for the first time; the result of this process is to create and store an optimized version of the specific code region. The metric which is used in the literature to estimate this task is *static / dynamic instruction ratio* (*sd*)[26]. Higher ratio implies higher translation overhead. The rationale behind this is very straight forward. Lower ratio means that code regions are repeated more often and so the relative contribution of the translation overhead is smaller.

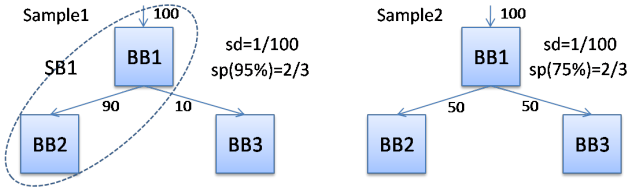

Figure 5: Example which illustrates same static / dynamic ratio, but different code region formation.

However, the previous metric estimates only roughly the translation overhead and it might be very inaccurate when the code regions are constructed using superblocks (single-entry, multiple-exit code regions) - SBs. To target this we introduce a metric called *static percentage* (*sp*). Static percentage expresses the percentage of static code needed to cover $X\%$ of the dynamically executed code. Such a metric is important to differentiate the behavior between two samples with same static / dynamic ratio, but different code region formation. In order to explain this situation, consider a scenario in Figure 5 where two samples have only three basic blocks ($BB1$, $BB2$, $BB3$) and each basic block has the same amount of instructions (*e.g.* 5). In the case of the first sample BB1 is executed 100 times, BB2 90 times and BB3 10 times; while in the case of the second sample BB1 is executed 100 times, BB2 50 times and BB3 50 times. Focusing only on *sd* metric these two samples are the same, which is wrong because in the case of the first sample branch $BB1 \rightarrow BB2$ is biased and the superblock $SB1$ is constructed, whereas in the second case that superblock is not constructed. On the other hand *sp* metric differentiates these two samples. The formal definition of *sp* is as follows:

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

$$\frac{|dynamic(S_X)|}{|dynamic(S_{total})|} \leq X, \quad (8)$$

where S_{total} is the set of all instructions in a given sample, while functions *static*() and *dynamic*() respectively corresponds to number of static and dynamic instructions executed by the sample. According to our experiments, the set of values for X that optimizes the estimation of $CPItransl$ is: $X = \{90\%, 80\%, 70\%, 60\%, 50\%\}$.

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

4.1.3 Code Cache estimation

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

4.2 Correlation of TDV and CPI

In order to back up the assumptions taken in the estimation of TOL particularities in previous section, we show Figure 6. Y-axis presents the correlation between the summands in Equation 7 (CPI_{LUP}, $CPI_{Transl.}$, $CPI_{CodeCache}$) and each of the statistics used for TDV (X-axis). The higher the correlation is, the strongest the dependency between the particular summand and the particular field in TDV. For example, the dependency between $CPI_{Transl.}$ and static / dynamic instruction ratio is quite strong with a correlation of 0.48.

As can be observed the CPI_{LUP} correlates the most with the number of indirect branches, indirect calls and returns (correlation of 40%) which confirms the assumption from Section 4.1.1. On the other hand, $CPI_{Transl.}$ highly correlates with static / dynamic ratio (*sd0* and *sd1*). Other metrics introduced in Section 4.1.2 ($sp0 - sp9$) also show high correlation with $CPI_{Transl.}$. Finally, $CPI_{CodeCache}$ correlates the most with Instruction Mix ($im1 - im5$), as proposed in Section 4.1.3.

4.3 TDV vs. BBV example

Table 1 shows an example where TDV clearly outperforms BBV. It contains three samples (**A**, **B**, **C**), which correspond

Table 1: Three samples (A, B, C) of 410.bwaves application represented in different statistics space: (a) TDV, (b) BBV and (c) CPI. The shaded lines refer to the similar samples for a given statistics space.

(a) TDV

	sd1	sd2	sp0	sp1	sp2	sp3	sp4	sp5	sp6	sp7	sp8	sp9	ind	im1	im2	im3	im4	im5
A	691	131	0.57	0.72	0.42	0.60	0.30	0.47	0.22	0.39	0.16	0.29	0.22	0.01	0.23	0.42	0.13	0.00
B	739	131	0.54	0.72	0.37	0.58	0.25	0.43	0.17	0.27	0.11	0.20	0.24	0.01	0.59	0.08	0.08	0.00
C	721	133	0.56	0.73	0.39	0.58	0.28	0.45	0.20	0.36	0.13	0.24	0.23	0.01	0.38	0.28	0.09	0.02

(b) BBV

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

(c) CPI

	CPI
A	1.72
B	2.27
C	1.96

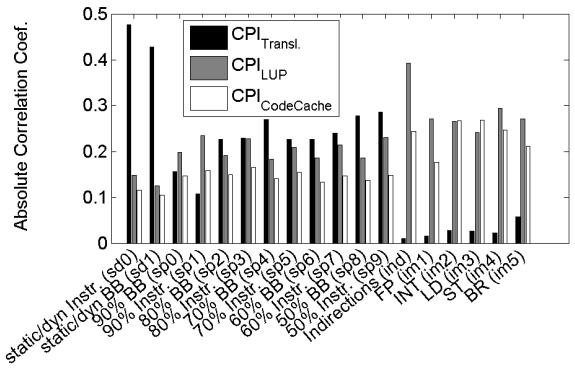

Figure 6: Correlation of the each field of TDV with three main components of CPI in HW/SW co-designed processors.

to 410.bwaves application stream, in three different cases (spaces): (a) TDV, (b) BBV and (c) CPI. This application is chosen as the one which has relatively low number of static BBs, so the BBV has a reasonably small length and it is easy to manually compare all the fields in BBV, whereas at the same time it has a relatively high TOL overhead compared to other applications.

Table 1 shows that samples **A** and **C** are similar in the TDV and CPI space. Similarity between the samples is calculated as Euclidean distance. For instance, if samples **A** and **B** have smaller Euclidean distance between each other than samples **A** and **C**, we say that **A** and **B** are *more similar* than **A** and **C**. Having a careful look at TDV we can notice that among all fields, sample **C** is closer to sample **A** than to sample **B**. Across all TDV fields the values collected for the sample **C** are closer to the values of the sample **A** than the values of the sample **B**.

On the contrary, in BBV space samples **A** and **B** are similar, which is against the CPI space. Consequently BBV will give worse similarity analysis than TDV in this case. The reason why in BBV space samples **A** and **C** are not similar is that BBV considers all basic blocks equally, while for HW/SW co-designed processors basic block which ends with an indirect branch is more important than the others. In particular in Table 1 basic blocks with indirect branches are BB4, BB12 and BB16, with the BBV coefficients ω_4, ω_{12} and ω_{16} respectively.

5. EXPERIMENTAL METHODOLOGY

5.1 Simulation Infrastructure

All the studies have been performed using DARCO [26], an infrastructure designed for research in HW/SW co-designed processors. DARCO uses QEMU [1] support for its TOL design and it models a HW/SW co-designed processor which executes x86 applications on top of a PPC-like CPU through TOL. PowerPC (PPC) has been chosen as an internal host ISA because it is a typical 3-input RISC ISA with broad tool support (such as QEMU, gcc and others). In addition, the PPC ISA is extended with new instructions specific for HW/SW co-design processors and our intention is to add more extensions in the future.

DARCO uses a TOL with 3 levels of execution and optimization: interpretation mode (IM), basic block translation mode (BBM) and superblock optimization mode (SBM). According to sensitivity studies [26], the optimal promotion thresholds are 5 ($TH_{IM \rightarrow BBM}$) and 10000 ($TH_{BBM \rightarrow SBM}$) respectively.

At the basic block translation level, the TOL only stores code regions consisting of a single basic block and it does not apply optimizations to them. On the contrary, at the superblock optimization level, superblocks are created by including basic blocks until the probability to reach the end of the superblock is below 80% based on profiling information. At this optimization level, superblocks pass through several optimizations (copy/constant propagation, constant folding, common sub-expression elimination, dead code elimination, register allocation, instruction scheduling and control and data speculation). DARCO's TOL also supports typical run-time optimizations, such as code region linking [5] and devirtualization of indirect branches [14].

The cycle-accurate simulator of DARCO models a configurable PPC-like in-order core. This CPU is a multi-way in-order processor with 9 pipeline stages, 3 levels of the memory hierarchy, a prefetcher and scoreboard scheduling logic. The main microarchitectural parameters are given in Table 2. The closest industrial processor to the modeled processor is PPC 450 [17].

For our experiments, we used the SPEC2006 benchmark suite [2].

5.2 HW-only processor model

In order to study how the different clustering schemes apply to HW-only versus HW/SW co-designed processors, we modeled the former as a HW/SW co-designed processors

Table 2: Microarchitectural Parameters

General			
Issue width	2;	Issue queue size	16
Units			
ALU INT	1 cycle latency		
MUL INT	2 cycles latency		
ALU FP	2 cycles latency		
MUL FP	5 cycles latency		
Caches			
I$	32KB, 4way, line:64b, LRU, 1 cycle		
D$	32KB, 4way, line:64b, LRU, 1 cycle		
L2	512KB, 8way, line:128b, LRU, 16 cycles		
Main Mem.	hit: 128 cycles		
TLB			
L1	64 entries, 8way, hit 1 cycle		
L2	256 entries, 8way, hit 16 cycles		
Main Mem.	miss: 512 cycles		
Branch predictor (G-share)			
Size of history register	12		
Stride Prefetcher			
Number of entries	32		

with only BB translation and without feeding the cycle-accurate simulator with TOL instructions. In such a scenario, we closely model a HW-only processor, as the main particularities of HW/SW co-designed processors are skipped (staged compilation and TOL overhead).

5.3 Experimental setup

Different phase classifications are evaluated using 100 samples per each application spread uniformly up to 200B x86 instructions (Figure 7). Each sample is 10M x86 instructions long. This amount of samples is used for the simulation time reasons. SPEC2006 is a very large benchmark suite, at least an order of magnitude bigger than SPEC2000, so cycle-accurate simulation of whole dynamic application stream is almost not feasible.

In order to ensure that using 100 samples per each application does not restrict the conclusions we did experiments for the case of 1000 samples per application for HW/SW co-designed processor. The error curves and the conclusions are similar to the case in which we simulate 100 samples. This means that in the case of taking into the consideration 1000 samples per application we do not catch many new phases in comparison with 100 samples scenario. In order to be consistent thought the paper, all results are based on the case of 100 samples per application.

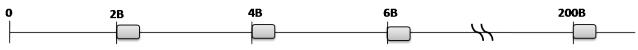

Figure 7: Experimental setup - studied samples.

5.4 Definition of the simulation metrics

The accuracy of the simulation technique is expressed by the average error of the estimation of gCPI[2] among all applications. The accuracy of gCPI is calculated as the relative error between the estimated gCPI (Equation 1) and the real gCPI (Equation 2):

$$gCPI_{re} = abs(gCPI_{real} - gCPI_{est})/gCPI_{real}. \quad (9)$$

Finally, the average error is computed across all applications as the arithmetic mean:

$$error = avg_{application}(gCPI_{re}) \quad (10)$$

[2]gCPI refers to the guest CPI, where instructions refer to guest instructions (in our case x86 instructions) error. This is the metric that allows comparing two different TOL configuration executions and is the typical metric used for studies for HW/SW co-designed processors.

6. RESULTS

6.1 BBV: HW/SW vs. HW-only

Figure 8 shows the simulation error of BBV phase classification for two cases: HW-only and HW/SW co-designed processors. X-axis presents the normalized number of samples selected for simulation, while the Y-axis presents the average relative error. As the figure shows, for the BBV phase classification scheme, the HW/SW co-designed processors suffer from significantly higher error compared to the HW-only designs. Moreover it backs up initial assumption made in this paper, that BBV phase classification does not provide a good trade-off between accuracy and number of samples.

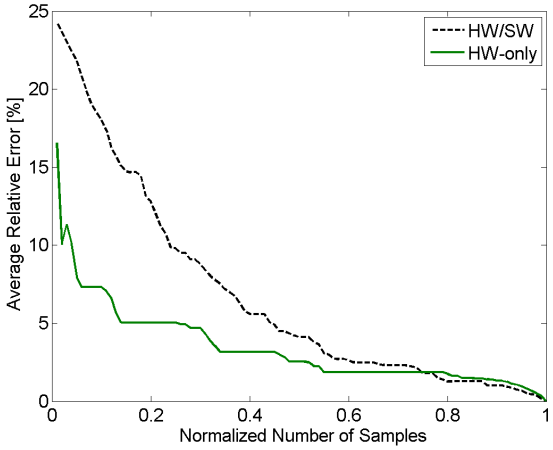

Figure 8: Average relative error vs. number of selected samples for BBV phase classification in two different scenarios: HW-only and HW/SW co-designed processor.

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

To eliminate the Random Linear Projection (Section 3.1) from being the reason behind the high inaccuracy of the BBV we performed studies with different lengths for the Random Linear Projection (RLP) vector. In particular we studied the effect of increasing the default value of the RLP length from 15 to 20, 30, 50 and 100 respectively. On the

contrary to the expected behavior, having bigger length of the RLP vector introduces higher errors and thus cannot be the reason for high error in the case of BBV phase classification. Such behavior is due to the nature of RLP, where the lower dimensionality of RLP makes statistical space more spherical and easier for clustering.

6.2 HW/SW: BBV vs. TDV

The results presented in Figure 9 compare the BBV and the TDV phase classification in the case of HW/SW co-designed processor. Similar to Figure 8, X-axis presents the normalized number of samples, while the Y-axis presents the average relative error.

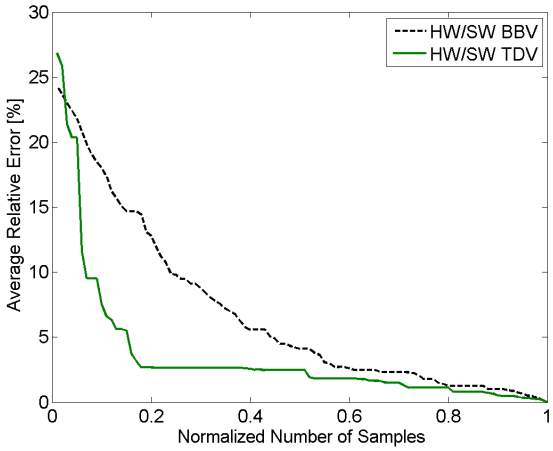

Figure 9: Average relative error vs. number of selected samples for BBV and TDV phase classification in the case of HW/SW co-designed processor.

The first observation from Figure 9 is that TDV has errors smaller than BBV. For 20 samples per application, TDV delivers a simulation error of 3%, whereas BBV delivers an error of 13%, an improvement of more than 4X. As can be observed, when the number of samples is between 10 and 40 (between 0.1 and 0.4 in Figure 9) the average simulated error in the case of TDV is around 2-4X smaller than in the case of BBV. The benefit of TDV can be also seen from a different perspective, which is the simulation time reduction. In order to reach the relative error of 3%, which is the typical threshold used in the literature, TDV requires 3X lees number of samples than BBV.

Figure 10 shows the simulation error across all applications from SPEC2006 benchmark suite in the case of 20 samples selected for each application. Across all samples TDV phase classification is almost always better than BBV phase classification. Only in two cases (453.povray and 471.omentpp) BBV gives better accuracy then TDV. However, these cases are cases with small error (up to 3%). In these cases the TOL overhead is minimal and the behavior of the application executed on the HW/SW co-designed processors is similar to the execution on the HW-only processor.

Moreover, notice that BBV has a maximum error of 50%, whereas TDV bounds the maximum error to 16%. For these benchmarks in order to have acceptable error, more samples have to be simulated. For instance for 400.perlbench, if we simulate 35 samples, the error for TDV is 4%. Finding the number of the samples needed to deliver a given simulation error is out of the scope of this paper. However the approach used in SimPoint, based on Bayesian Information Criterion [29], can be used.

There are 11 applications with really high simulation error (more than 10%) in the case of BBV phase classification. It is important to notice that in all of these cases TDV significantly outperforms BBV. Moreover, only for one application (400.perlbench) TDV gives an error more than 10%. The reason behind it is based on the fact that 400.perlbench has high number of static SBs which makes its behavior very unpredictable under a HW/SW co-designed execution scenario [6].

6.3 HW/SW: Different Configurations

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

We study the variations for both components: microarchitectural (HW) and TOL (SW). For the variations of TOL, we studied the following configurations: (*i*) default, (*ii*) without Optimizations and (*iii*) without Linking. For the variations of microarchitecture we studied: (*i*) doubled data cache, (*ii*) half data cache, (*iii*) double second level cache, (*iv*) half second level cache and (*v*) four times smaller branch predictor history.

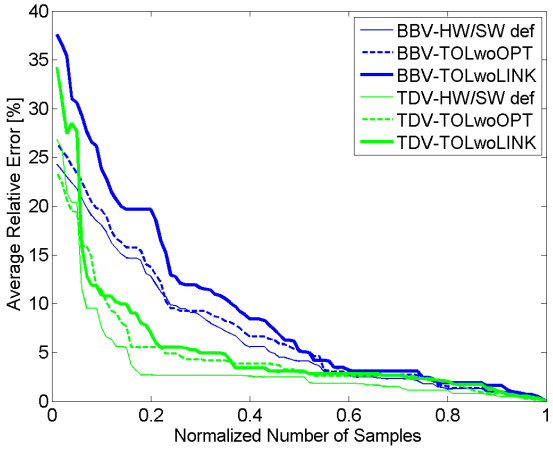

Figure 11: Average relative error vs. number of selected samples for BBV and TDV phase classification in the case of different TOL in HW/SW co-designed processor: (*i*) default, (*ii*) without Optimizations and (*iii*) without Linking.

The results for TOL variation are presented in Figure 11. Two things can be observed. First, for every TOL configuration and when less than 40 samples are selected, TDV still has 2-3X smaller error than BBV. The second observation is that the accuracy depends on the specific configuration. For instance TOL *without Linking* has around 3-4% bigger error then default TOL. This is the case for both, BBV and TDV phase classification. Therefore, these studies show that for different configurations different numbers of samples are needed to deliver particular error.

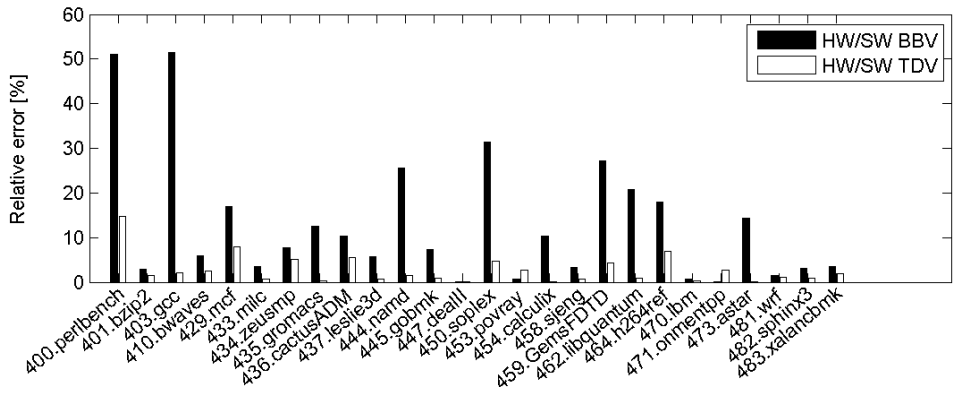

Figure 10: Relative error per application (SPEC2006) for BBV and TDV phase classification in the case of HW/SW co-designed processor for a fixed (k=20) samples per application.

The error of the variation in the case of different microarchitectural configurations is negligible. In other words, the error curve is the same like the one presented in Figure 9. Such a behavior is expected, since the host hardware is a simple in-order processor, so for different microarchitectural configurations, the amount of cycles is just scaled similarly up or down across all samples. Therefore, the error remains almost the same.

6.4 HW/SW: Other Statistics

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

Figure 12 shows the error of other statistics such as: the number of the host instruction and the SB coverage, for TDV and BBV phase classification. This shows that the errors are smaller than the error of gCPI, and that TDV has around 2X better accuracy than BBV. The results also show that even though BBV in some cases estimates accurately the code cache behavior, it does not express accurately SB coverage. This means that $sp0$ - $sp9$ fields in TDV express more accurately the SB coverage than BBV. This is due to the fact that BBV does not have any information about SB estimation.

7. CONCLUSIONS

Evaluation techniques in traditional microprocessor design are mostly based on simulating a few samples across whole application. Samples are usually specified by offline phase classification. However, traditional phase classification is not applicable to HW/SW co-design processors,

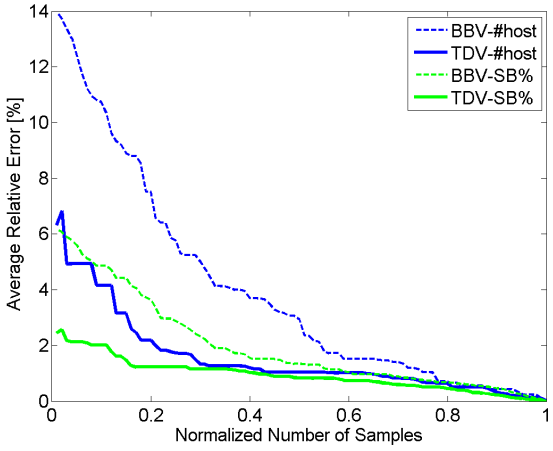

Figure 12: Average relative error vs. number of selected samples for BBV and TDV phase classification in the case of HW/SW co-designed processor for number of host instructions (#host) and SB coverage (SB%).

in which a Transparent to the entire software stack Optimization Layer (TOL) dynamically translates and optimizes guest instructions to an internal host ISA.

In short the contributions of this paper are the following: (i) we show that traditional Basic Block Vector (BBV) phase classification is not suitable for HW/SW co-designed processors, (ii) we propose a novel phase classification called TOL Description Vector (TDV). On average, TDV phase classification reaches the same error like BBV with 3X less number of samples. Such a trend does not depend on different TOL neither on microarchitectural configurations.

8. ACKNOWLEDGMENTS

This work is partially supported by the Generalitat de Catalunya under grant 2009SGR1250, the Spanish Ministry of Education and Science under contract TIN2010-18368, and Intel Corporation. Aleksandar Branković was partially funded by the Generalitat de Catalunya with a FI-AGAUR grant.

9. REFERENCES

[1] Quick EMUlation tool (http://http://www.qemu.org/).

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

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

[4] E. Argollo, A. Falcon, P. Faraboschi, M. Monchiero, and D. Ortega. Cotson: Infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, January 2009.

[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*, PLDI '00, pages 1–12, 2000.

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

[7] T.E. Carlson, W. Heirman, and L. Eeckhout. Sampled simulation of multi-threaded applications. In *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 2–12, 2013.

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

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

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

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

[12] N. Hardavellas et al. Simflex: A Fast, Accurate, Flexible Full-System Simulation Framework for Performance Evaluation of Server Architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4):31–34, March 2004.

[13] M. Hauswirth and A. Diwan. Phases in Branch Targets of Java Programs. Technical Report CU-CS-983-04, 2004.

[14] J. D. Hiser and D. Williams et al. Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems. *ACM Trans. Archit. Code Optim.*, 8(2):9:1–9:28, June 2011.

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

[16] T. Huffmire and T. Sherwood. Wavelet-based phase classification. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 95–104, 2006.

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

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

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

[20] K. Krewell. Transmeta gets more efficeon. *Microprocessor Report*, 2003.

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

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

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

[24] N. Neelakantam, D. Ditzel, and C. Zilles. A Real System Evaluation of Hardware Atomicity for Software Speculation. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS XV, pages 29–38, 2010.

[25] G. Ottoni et al. AstroLIT: enabling simulation-based microarchitecture comparison between Intel and Transmeta designs. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 21:1–21:2, 2011.

[26] D. Pavlou, A. Brankovic, R. Kumar, M. Gregori, S. Kyriakos, E. Gibert, and A. Gonzalez. DARCO: Infrastructure for Research on HW/SW co-designed Virtual Machines. In *Proceedings of AMAS workshop, in conjuction with ISCA*, 2011.

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

[28] S. Sathaye et al. BOA: Targeting multi-gigahertz with Binary Translation. In *Proceedings of the 1999 Workshop on Binary Translation, IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pages 2–11, 1999.

[29] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 45–57, 2002.

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

[31] Y. Wu, S. Hu, E. Borin, and C. Wang. A HW/SW co-designed Heterogeneous multi-core Virtual Machine for energy-efficient general purpose computing. In *Proceedings of the 2011 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 236–245, 2011.

[32] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical sampling. In *Proceedings of the 30th annual International Symposium on Computer Architecture*, ISCA '03, pages 84–97, 2003.

[33] C. Wung, Y. Wu, and M. Cintra. Acceldroid: Co-designed acceleration of Android bytecode. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '13, pages 1–10, 2013.