Efficient Software-based Online Phase Classification

Andreas Sembrant, David Eklov and Erik Hagersten
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
{andreas.sembrant, david.eklov, eh}@it.uu.se

Abstract

Many programs exhibit execution phases with time-varying behavior. Phase detection has been used extensively to find short and representative simulation points, used to quickly get representative simulation results for long-running applications. Several proposals for hardware-assisted phase detection have also been proposed to guide various forms of optimizations and hardware configurations.

This paper explores the feasibility of low overhead phase detection at runtime based entirely on existing features found in modern processors. If successful, such a technology would be useful for cache management, frequency adjustments, runtime scheduling and profiling techniques.

The paper evaluates several existing and new alternatives for efficient runtime data collection and online phase detection. ScarPhase (Sample-based Classification and Analysis for Runtime Phases), a new online phase detection library, is presented. It makes extensive usage of the new hardware counter features, introduces a new phase classification heuristic and suggests a way to dynamically adjust the sample rate. ScarPhase exhibits runtime overhead below 2%.

1. Introduction

It is well known that many programs exhibit time-varying behavior [29]. As an example, some parts of an application's execution may be memory bound while others are compute bound. Categorizing this application based on its average behavior can be misleading as it may appear to be neither compute nor memory bound.

To capture such time varying behaviors, researchers have proposed detecting *program phases*. A program phase is defined to be a period of execution with a stable behavior. There are several important optimizations, both offline [5, 13, 27, 30, 32] and online [8, 15, 17, 24], which benefit from knowledge of program phases. For example, a CPU scheduler that knows the present phase of each runnable process can make better scheduling decisions and improve resource utilization [24, 32]. Other example include dynamic recompilation where knowledge of program phases can help determining when it is worth recompiling [5, 13, 26], efficient profiling where profiling is only performed for a phase with yet unknown behavior[25], or collection of representative simulation points to speed up simulation [30].

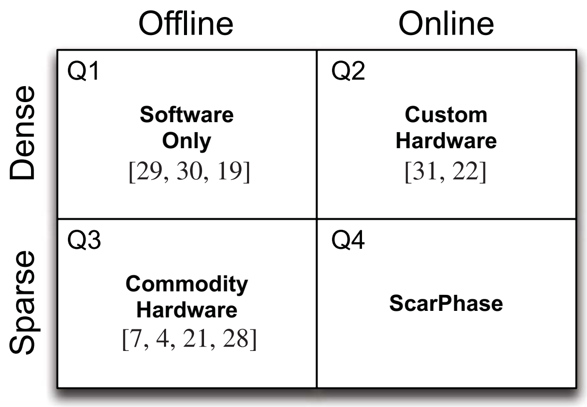


Figure 1. Taxonomy of program phase classification methods.

The goal of this paper is to develop and explore online techniques for phase classification. For such a technique to be generally applicable, it must have the following properties: 1) It should not require custom hardware support; 2) It should have minimal runtime overhead without loss of accuracy and fidelity; 3) It should be transparent and non-intrusive (e.g., require no recompilation of the analyzed program and work with dynamically generated code), 4) It should be independent of the system load, and finally; 5) It should capture general purpose phase behavior (i.e., not study a single property such as L3 miss rate for the phase classification).

A large variety of program phase classification and prediction techniques have been studied in previous work [8, 9, 31, 19]. Most of these methods rely on the observation that the behavior of an application is highly correlated with the code it currently executes. This is typically captured using *basic block execution frequencies* that are collected during the applications execution. The application's execution is divided into non-overlapping, adjacent, intervals for which execution frequencies are collected. If two intervals have similar enough execution frequencies they are classified as belonging to the same program phase.

Program phase classification methods can be categorized along two dimensions: whether they count the execution frequencies of every basic block (dense) or randomly sample the execution frequencies (sparse), and whether they do the classification on- or off-line. Figure 1 shows the four quadrants for such a classification. As we target online phase classification, which require low runtime overhead, the most interesting aspect of this classification is the runtime overhead of the different methods. The methods in quadrant Q1 use instrumentation to collect dense execution frequencies, which is accurate but often has a high runtime overhead. Two different approaches have been investigated to reduce the runtime overhead; using custom hardware (Q2) and using sparse execution frequencies (Q3). The methods in Q2, that use custom hardware, have mainly been used for online classification. While being both accurate and low overhead; the drawback of these methods is that they cannot be used on commodity hardware. The methods shown in Q3 use sparse execution frequencies. This allows them to use hardware performance counters to collect runtime information, which results in very low runtime overhead. However, collecting sparse frequency vectors using hardware performance counters in combination with online classification (Q4) has not yet been investigated. Given the advancement in hardware counter technology, we envisioned that this quadrant could be conquered without the need for dedicated hardware support. If at all feasible, such a solution would have a fast uptake and be an enabler for implementing the many phase guided optimizations proposed to date.

In this work we try to leverage many of the results from previous work. We utilize sparse execution frequency vectors collected using hardware performance counters and we compare the quality of previously proposed frequency vectors in the context of online phase classification. The main contributions of this work are:

- We propose a new and efficient way to collect sparse frequency vector at a basic block granularity using Intel's Precise Event Based Sampling (PEBS) to sample branch instruction.
- We propose and evaluate a new sparse frequency vector based on conditional branches enabled by PEBS. This vector requires fewer samples and therefore reduces the overhead of data collection compared with previous work.
- We propose a method that allows us to dynamically adjust the sampling frequency, which reduces the runtime overhead by an additional factor of two.
- We have developed a general and easy to use library for online phase classification and prediction, called ScarPhase (Sample-based Classification and Analysis for Runtime Phases).

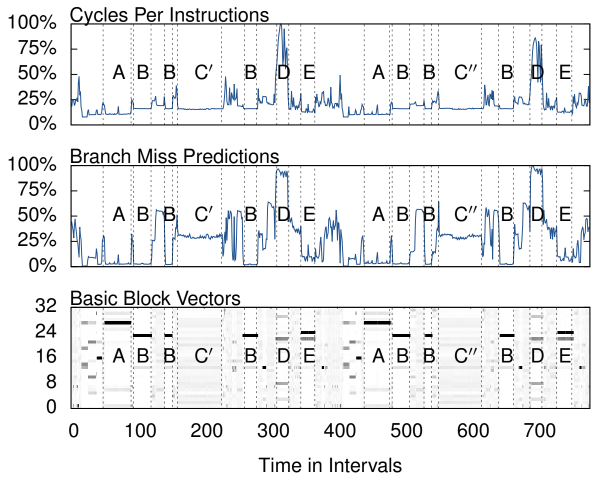


Figure 2. The top and middle figures show how gcc/166's CPI and BMP changes over time, respectively, expressed as percentage of the maximum observed CPI/BMP. The bottom figure shows BBVs with 32 entries for each execution interval. The counters are shown on the y-axis, where darker shades of gray indicated more frequently executed basic blocks.

2. Phase Detection Algorithms

A *program phase* is defined to be a period of execution with stable behavior with respect to a given performance metric, e.g., cycles per instruction (CPI) or branch miss predictions (BMP). For example, Figure 2 shows how gcc/166's CPI and BMP changes over time. While both the CPI and BMP vary greatly, there are periods where they are relatively constant. These periods of constant behavior are the program phases of gcc. Two of the most prominent phases are labeled C' and C''. As in previous works, we do not distinguish between C' and C'', and instead say that they are two occurrences of the same phase.

Most program phase classification methods divide the application's execution into non-overlapping, fixed size, *ex-*

ecution intervals. Their size is typically measured in executed instructions. To find the program phases, the application is profiled in order to measure the behaviors of individual execution intervals. The intervals with relatively similar behaviors are then classified as belonging to the same phase. For example, the first occurrence of phase C in Figure 2 (labeled C'), consists of about 70 consecutive execution intervals of 100M instructions with relatively similar behaviors.

2.1. Execution Frequency Vectors

It has been observed that most performance metrics are strongly correlated with the code being executed [8, 9, 29, 30]. This observation has been the basis for many program phase classification techniques [19]. These methods typically use some form of *execution frequency vectors* to capture what code is executed during an execution interval, and define a measure of distance between the vectors. This distance is then used to compute the similarity of the intervals, and classify them into program phases.

Dhodapkar and Smith [8, 9] use a bit vector to keep track of what instructions have been executed during the execution intervals. When an instruction is executed its address is hashed into the bit vector, and the corresponding bit is set. Sherwood et al. [29, 30] keep track of the execution frequencies of basic blocks. They found that it is not necessary to have one counter for each basic block. Instead they use a small vector of counters, called a *Basic Block Vector* (BBV), and hash the basic blocks (using their addresses) onto the counters in the BBV. This has the benefit of reducing storage overhead and lowering cost of computing the distances between BBVs.

Figure 2 shows the BBVs for all the execution intervals of gcc/166. Dark colors indicate high execution frequencies. The figure shows a strong correlation between the BBVs and the program phases. For example, consider phase C for which the corresponding BBVs are nearly identical.

Lau et al. [20] evaluated the quality of phase classification based on the execution frequencies of a number of different program structures such as: function calls, loop branches, particular op-codes, register usage, memory access strides, and memory working sets. Their results suggest that none of these alternatives results in significant improvement over basic block execution frequencies.

In this work we therefore focus mainly on basic block execution frequencies.

2.2. Sparse Execution Frequency Vectors

To reduce the runtime overheads of collecting execution frequencies, Davies et al. [7] use hardware performance counters to collect sparse samples of instruction execution frequencies, which they use to estimate *Extended Instruction Pointer Vectors* (EIPV). EIPVs are similar to BBVs, but instead of capturing execution frequencies of basic blocks they capture execution frequencies of individual instructions.

Davies et al.'s original implementation uses VTune [2]. Our implementation, uses comparable features of Linux-perf_events [1], which has been available in the mainline Linux kernel since version 2.6.32. Using perf_events, the performance counters can be programmed to operate in two main modes. Either they count the occurrences of certain events, e.g., executed instructions, or they periodically trigger interrupts after a given number of occurrences of the event, called the sample period. These interrupts are caught by perf_events that in turn reacts to the interrupt in one of two different ways. Either it forwards the interrupt to user space in the form of a signal[1], or it records the state of the CPU, including its instruction pointer, at the time the interrupt is handled. These recorded CPU states can later be requested from user space.

To capture EIPVs Davies et al. use two performance counters. The first counter is used to select what instructions to "sample". This is done by programming perf_events to record the state of the CPU every N executed instructions, where N is the sample period. The second counter is used to notify user space at the end of each execution interval. This is achieved by programming the counter to send a signal after I executed instructions, where I is the execution interval size. When the signals are received we read the recorded CPU states and use the instruction pointers to build an EIPV for the current execution interval.

While EIPVs can be collected fairly easily using standard hardware performance counter features, they are less accurate than BBVs. Since all the instructions in a basic block are executed the same number of times, it is more effective to count basic block execution frequencies and weight the count with the number of instructions in the basic blocks. Lau et al. [21, 28] showed that the results can be significantly improved by mapping the sampled instructions to their corresponding basic block. This however requires analysis of the program binary and is therefore not suitable for dynamically generated code. To leverage the low overhead data collection of Davies et al. but at the same time achieve the accuracy of BBVs, Lau et al. [21, 28] map instruction addresses to their corresponding basic blocks using the program binary. This however, has two problems: it increases the runtime data collection overhead, and it cannot be used to analyze dynamically generated code (e.g., JIT'ed code) as the mapping information is not readily available. We refer to these vectors as *Mapped Basic Block Vector* (MBBV).

[1]To receive a signal, the user space process has to setup asynchronous notification on the perf_events file descriptor

2.3. Sparse Branch Vectors

It would be a clear advantage if we could find a way to directly capture basic block frequencies instead of first capturing instruction frequencies and later translate them to basic block frequencies.

Since each basic block ends with one branch instruction, we could attempt to record the addresses of sparsly selected branch instruction instead of the address of sparsely selected instructions. We could, for example, program perf_events to record the address of every Nth branch instruction, which at the end of the execution intervals would give us a sample of the executed branch instructions that directly corresponds to the BBVs. This, however, does not work due to performance counter skid [3].

If the performance counter is used to generate interrupts after N branch instructions, there will be a short delay between the time of the Nth branch and the time when the interrupt handler is invoked, during which the CPU keeps executing instructions. This delay is referred to as *performance counter skid*. It is first when the interrupt handler is invoked that perf_events records the CPU state, which due to the skid can not be guaranteed to point to a branch instruction.

To reduce the impact of performance counter skid, Intel introduced *Precise Event Based Sampling* (PEBS) [16]. When PEBS[2] is enabled, the CPU saves its state at the time when the Nth event occurs. This saved state can then be read by perf_event's interrupt hander

One of the key contributions of this paper is the proposal to use PEBS to directly measure sparse BBV vectors. This is done by setting up perf_event to sparsely collect the address of every Nth branch instruction and to build BBV frequency vectors based on those addresses. We refer to this method as *BRanch Vectors* (BRV).

2.4. Sparse Conditional Branch Vectors

To further reduce the overhead of BBV collection we also propose an alternative method and collect an even more distilled form of BBVs. This idea is inspired by the observation that most functions contain loops and if statements. Recording the entry and exit point will therefore not add any additional information. The execution count of the first basic block in a function can be inferred from the execution count of the basic block following it. Counting conditional branches only, therefore, results in a minimal loss of information compared to counting all branches.

[2]However, there is a small delay before the processor state is recorded, called shadowing [23]. For example, if we program a performance counter to trigger after the execution of 1000 branch instructions, the processor state might be recorded first when executing the 1001th instruction. This, however, does not present a problem for the work presented in this paper.

Table 1. List of abbreviations

EIPV	Extended Instruction Pointer Vector [7]
MBBV	Mapped Basic Block Vector [21, 28]
BRV	BRanch Vector (New)
CBRV	Conditional BRanch Vector (New)

To capture *Conditional BRanch Vectors* (CBRVs) we use a performance counter that counts conditional branches only, which is available on Intel's Nehalem chips.

Table 1 shows a list of abbreviation. We will refer to the different execution frequency vectors by their abbreviations in the rest of this paper.

3. Experimental Setup

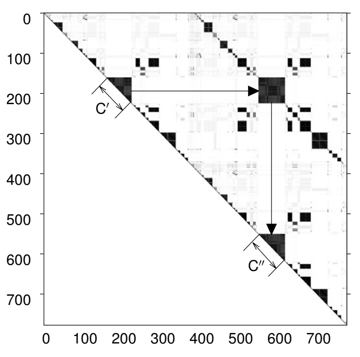
We performed our evaluation on *astar/lakes*, *bzip2/chicken*, *bwaves*, *dealii*, *gcc/166*, *mcf*, *perl/splitmail*, *wrf* and *xalan* from the SPEC 2006 [14] benchmark suite. We chose the programs above because they display the most interesting phase behavior. All benchmarks were run to completion with their reference input on an Intel Xeon E5620 (Nehalem) system.

4. Case Study: GCC

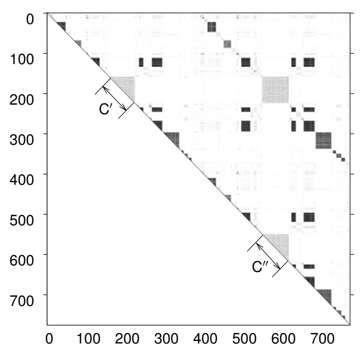
In this section we evaluate the quality of the execution frequency vectors obtained using the methods discussed in Section 2.3 and 2.4 by comparing them to execution frequencies obtained using a Pin [6] based reference implementation. The reference implementation dynamically instruments the benchmark applications and counts each execution of all basic blocks, and provides a ground-truth reference. We use an execution interval size of 100M instructions for both the reference and the evaluated methods.

For the evaluation in this section we focus on gcc which is one of the SPEC CPU2006 benchmarks with the most complicated phase pattern. Figure 2 shows how the CPI and branch miss predictions changes over time for gcc/166. The figure shows that the behavior of gcc changes greatly over time.

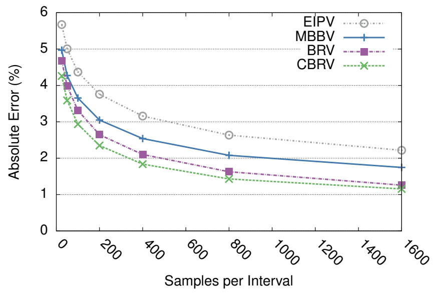
To quantify the difference between the captured execution frequency vectors and reference execution frequencies vectors, we compare their basic block similarity matrices [30]. A basic block similarity matrix is $N \times N$ upper triangular matrix M, where N is the number of intervals. Each element $M_{n,m}$ is the Manhattan distance between the execution frequency vectors of the nth and mth interval. Figure 3(a) shows the similarity matrix for gcc/166 generated from the reference BBVs. The gray scale indicates the similarity between BBVs, where darker shades represents shorter Manhattan distances. The similarity matrix allows



(a) Matrix are generated based on BBVs captured using the reference implementation. Darker shades of gray indicate shorter Manhattan distances.



(b) Matrix are generated based on BBVs captured using ScarPhase. Darker shades of gray indicate shorter Manhattan distances.



(c) The average absolute element-wise differences between the reference similarity matrix and the similarity matrices generated from data captured using EIPVs, MBBVs, BRVs and CBRVs for gcc/166. Lower is better.

Figure 3. Basic block similarity matrices for gcc/166.

us to evaluate the quality of execution frequency vectors independent of the phase classification method.

We can interpret the basic block similarity matrix as follows. The application's execution progresses along the diagonal. The triangles above the diagonal indicate that neighboring intervals have similar execution frequency vectors and therefore belong to the same phase. For example, see Figure 3, where we have labeled the two occurrences of phase C (C' and C''). To find possible reoccurrences of a phase, we start at the triangle marking the phase and move horizontally to the right across the matrix until we reach a dark rectangle. Then, from the rectangle, we continue straight down until we reach the diagonal. The triangle that we land on indicates the similarity of a set of intervals which belong to the phase where we started.

Figure 3(b) shows the similarity matrix for gcc/166 generated from the BBVs captured using ScarPhase. Comparing the two matrices (Figure 3(a) and Figure 3(b)), we see that the one generated using ScarPhase has an overall similar structure, but is somewhat brighter, indicating longer distances between BBVs. This difference is mainly due to statistical sampling errors. For example, the triangles corresponding to phase C is much brighter in Figure 3(b). This is because gcc accesses a large instruction working set in phase C, and accurately capturing the BBV for such intervals requires higher sample rates.

To quantify the difference between two similarity matrices we compute the average element-wise difference between the two matrices. Figure 3(c) shows the average elemen-wise difference between the reference similarity matrix and the ones based on EIPVs, MBBVs, BRVs and CBRVs for a range of sample periods[3], i.e., the number of "sampled" instructions/branches during execution intervals. As expected the similarity improves when the number of samples per interval is increased. However, the curves start to level of when more than 800 samples per interval is used, suggesting that there is no real benefit to use more than 800 samples for any of the methods.

EIPV and MBBV both rely on the same runtime data; sparse instruction execution frequencies, and do not use PEBS. The difference between the error for these two methods shows that basic block vectors more accurately capture the phase behavior than the extended instruction vectors. BRV and CBRV both use PEBS, BRV, however, count all types of branches, while CBRV only counts conditional branches. The difference between MBBV (not using PEBS) and BRV (using PEBS), shows that PEBS significantly improves accuracy.

CBRV has the lowest overall error. If we count all the dynamic executions, i.e., not using sampling, then CBRV would yield a larger error than BRV, as it ignores all non-conditional branches. However, this is not the case when sampling. As the figure shows, for a given number of samples, CBRV performs better than BRV. This is because some of the non-conditional branches sampled when capturing BBVs can be inferred from the conditional branches sampled when capturing CBRVs (see Section 2.4). In this sense, each conditional branch sample contains more information, and the CBRV therefore outperforms BBV for low sample rates.

[3]For all the experiments presented in this paper we use periodic sampling. We also tried random sampling, but the results showed only minor improvements over periodic sampling. However, random sampling had to be implemented in the kernel to work with in-kernel buffering, and since changing the kernel is often undesirable, we decided to use periodic sampling.

5. Phase Analysis

In the previous section we evaluated how well the different execution frequency vectors (EPIV, MBBV, BRV and CBRV) capture changes in the applications behavior independent of the program phase classification algorithm by comparing their basic block similarity matrices. In this section, we evaluate their impact on the quality of the resulting phase classifications. Our goal is to identify which execution frequency vector gives us the best phase classification while incurring the lowest runtime data collection overhead.

For phase classification we use the leader-follower clustering algorithm [10]. It identifies cluster of similar execution frequency vectors, and we interpret each clusters as program phases. The algorithm works as follows. At the end of every execution interval, we apply leader-follower clustering to the interval's execution frequency vector. If the vector is close enough to the center of an existing cluster, it is added to the cluster, and the cluster's center is recomputed. Here, close enough means that the Manhattan distance between the execution frequency vector and the cluster center is below a threshold. Otherwise a new cluster is created to which the vector is added. The interval is classified as belonging to the phase defined by the cluster to which it was added.

5.1. Online Phase Classification

There are several important aspects to consider when evaluating the quality of program phase classification. The importance of these aspects depends on the context in which the program phases are used. In this section we therefore focus on what we believe to be a fairly general application of program phases; phase guided profiling [25].

To reduce profiling overhead, phase guided profiling only profiles a few execution intervals from each program phase. In this context, there are two important aspects to evaluate, the phase classification; the homogeneity of the phases, and the number of different phases detected. Since the behavior of a whole phase is estimated based on the behavior of only a few intervals, homogeneity has a large impact on the profiling accuracy. The number of detected phases, on the other hand, impacts the profiling overhead, if too many phases are detected more execution intervals have to be profiled, resulting in larger profiling overheads.

To measure phase homogeneity we use the *Coefficient of Variation* (CoV) [31]. To compute the CoV we first measure the CPIs of all execution intervals. Then, for each phase, we compute both the average and the standard deviation of the CPIs of the intervals belonging to the same phase. The per-phase CoV is then the standard deviation divided by the average. Finally we compute the whole program CoV as the weighted average of the per-phase CoVs. Note that CoV is a lower-is-better metric.

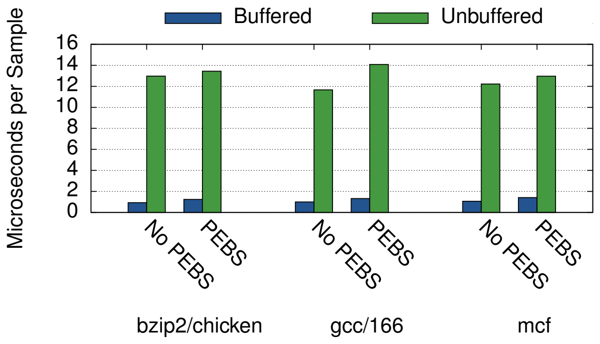


Figure 4. The average time in microseconds to collect one sample. With and without using in kernel buffering, and with and without using PEBS.

One issue when using CoV to assess the quality of phase classifications is that it does not consider the number of detected phases. For example, a naive phase classification method that classifies all intervals as belonging to different phases achieves a CoV of zero. With such a classification method we would end up profiling all intervals, which defeats the purpose of phase guided profiling. Therefore, we want to adjust the CoV metric to penalize phase classification methods that detect too many different phases. For this we use the following observation. As we can identify a new phase first after having seen its first interval, phases must span at least two intervals in order to be profiled. Therefore, whenever we identify an execution interval that is not classified as belonging the same phase as its neighbors, we classify it as belonging to a "virtual" phase (this is only done for CoV calculations). When computing the CoV of the whole program, we set the CoV of the "virtual" phase to the CoV computed from all the application's intervals (i.e., the CoV of a phase classification method that classifies all intervals as belonging to the same phase). This gives us a goodness metric that penalizes phase classification methods that identify too many phases. We call this new metric Corrected CoV (CCoV). Now, in the extreme case when all execution intervals are classified into different phases, they will all end up in the virtual phase and the CCoV will be equal to the CoV of the whole program, which is the desired result.

Figure 5 shows the CCoV of the phase classification for different number of samples per interval, lower is better. EIPV has on average the lowest accuracy. For example, 25 samples per interval with MBBV/BRV/CBRV are on average better than 50 samples with EIPV. We find that all four methods have comparable accuracies when a high number of samples are used. However, at a lower sample rate, CBRV has the best quality followed closely by BRV then MBBV.

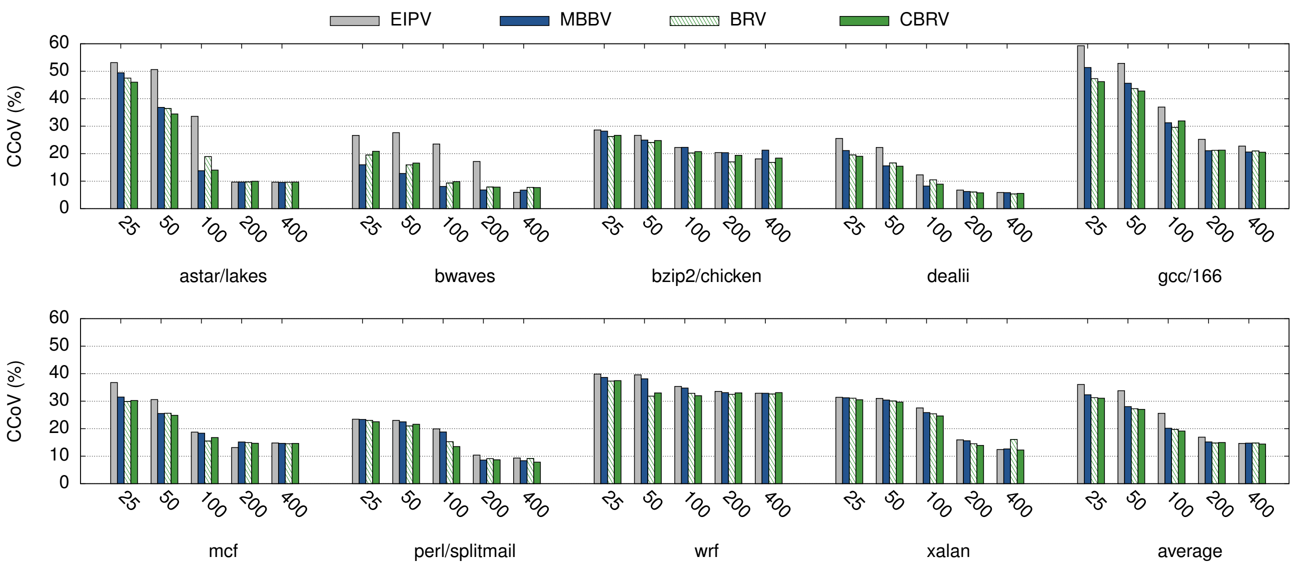


Figure 5. CCoV for different number of samples per interval. Lower is better.

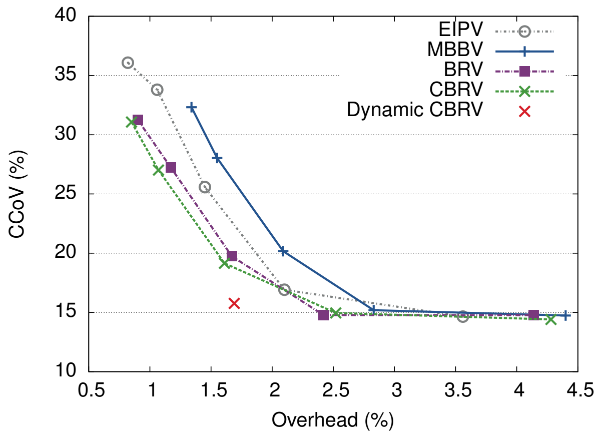


Figure 6. Average CCoV vs runtime overhead.

5.2. Runtime Overhead

In this section we examine the overhead of the different methods. EIPV has the lowest per sample cost as it require no PEBS support or mapping, but on the other hand it requires more samples to achieve the same accuracy (almost twice the number of samples). BRV and CBRV has an additional PEBS cost, each sample require the processor to write the state of the registers to main memory. MBBV first parses the program binary to create a instruction to basic block mapping, then during each sample map the instruction pointer to a basic block.

To measure what it cost to use PEBS we measured the overhead of sampling every one thousand instructions with and without PEBS and divided the overhead by the number of samples. Figure 4 (Buffered) shows the time in microseconds to collect one sample. For gcc/166, the cost to collect one sample using PEBS is 33% more expensive than without PEBS support.

There is an expensive context switch between user and kernel mode when a signal is sent to notify the monitoring process of a new sample. Linux perf_events can be configured to store the samples in a memory mapped file. By disabling event notification, i.e., no signal is sent, the samples can be buffered. Only at the end of the execution interval is the memory mapped file parsed (another counter is used to divides the execution into intervals).

Figure 4 (Buffered vs. Unbuffered) shows the time to collect one sample with and without buffering. On average, the sample cost is reduced with a factor 11. This means that for the same overhead, 11 times more samples can be collected. It is therefore vital to buffer the samples in kernel space for online phase classification.

In the previous section we evaluated the accuracy at different number of samples, however the cost of a sample varies between the methods. Figure 6 shows the accuracy against the overhead. This shows that CBRV results in the most accurate phase classification for a given overhead and MBBV the worst. Comparing EIPV with BRV/CBRV shows that the benefit of using PEBS clearly outweighs the additional runtime cost.

One design parameter of great importance for both accuracy and overhead, but not evaluated here, is the execution interval size. In practice, the most appropriate interval size may depend on how the phase information will be used. Using larger execution intervals would have an almost linear effect on the runtime overhead for all methods studied in this section (assuming that the number of samples per interval is held constant). Throughout this paper we use a execution interval size of 100M instructions as in most prior works [30, 7, 4, 28].

To summarize; MBBV, BRV and CBRV have comparable accuracies, and are all better than EIPV. CBRV has the best accuracy for a given overhead. Overall, the accuracy starts to level out at 200 samples, suggesting that 200 samples per interval is a good trade-off between accuracy and overhead. Throughout the rest of the paper, we therefore focus mainly on CBRV and use a sample rate of 200 samples per interval.

6. Dynamic Sample Rate

In the previous section we found that 200 samples per execution interval results in the best trade-off between accuracy and performance, resulting in an average runtime overhead of 2.5%. In this section we will develop a method to dynamically adjust the sample rate. This method reduces the average runtime overhead down to 1.7% without sacrificing accuracy.

By profiling our ScarPhase implementation we found that it spends about 85% of the time handling samples in the kernel. The most effective way to reduce the runtime overheads is therefore to reduce the cost of handling samples. However, as this is handled in the kernel, it is out of our reach to reduce the cost per sample. To reduce the runtime overhead we therefore need to reduce the number of samples per interval. However, as we saw in Section 5.1 (see Figure 5), reducing the number of samples per interval below 200 can have a negative impact on the phase classification accuracy.

Most programs have relatively long runs of consecutive intervals that belong to the same phase. We can take advantage of this in order to lower the number of samples per interval. When entering a new phase, we start off sampling 200 branches per interval, and can confidently classify the first interval. Once we have classified the first interval, we assume that the following intervals belong to the same phase, and we therefore only need to detect when a phase change occurs. This turns out to be much easier, and can be accurately done based on fewer samples (i.e., less than 200)[4]. This allows us to dynamically reduce the sample rate. This is done exponentially until we reach a lower limit of 25 samples per interval. When a phase change is detected, we go back to sample 200 branches per interval.

There are two things to note about the above method. First, when detecting a phase change, we have to classify the current interval (i.e. the first interval in the new phase), however, as we have lowered the sample rate, we have less than 200 samples, and the classification might therefore be less accurate. To reduce the impact of this we use a next phase predictor that predicts the phase of the next interval. If it predicts that the next interval belongs to a different phase, we pessimistically increase the sample rate back to 200 samples per interval, and can now more accurately classify the interval. For this, we use a history-based predictor similar to the one used by Sherwood et al. [31]. Second, if the predictor incorrectly does not predict a phase change, it will not help us, and we need to classify the current interval based on less than 200 samples. This is particularly troublesome if we incorrectly classify the interval as a new phase (i.e. one that we have not previously encountered) as this increases the total number of detected phases. Therefore, whenever we detect a new phase while sampling with a lowered sample rate, we do not attempt to classify the interval, and consider it as unclassified.

[4]When we look for phase changes and are sampling at a lower rate, we are more sensitive to sampling noise, and we therefore use a somewhat higher threshold than when doing full phase classification.

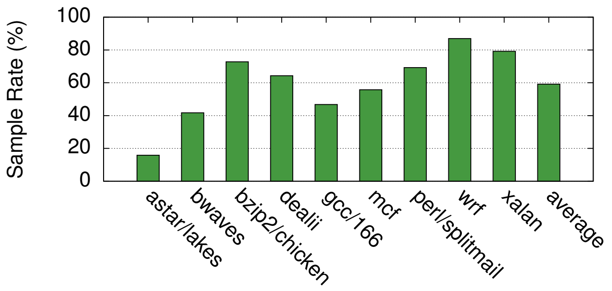


Figure 7. Average number of samples per execution interval when using dynamic sample rate. The number of samples are shown as percentages of the maximum number of samples per interval (200).

Figure 7 shows the average sample rate in percentage of the maximum sample rate (corresponding to 200 samples per interval) for all benchmarks. astar/lakes has few phases with very long run lengths, the sample rate is therefore reduced to 15% (30 samples per interval). wrf on the other hand has many phase changes, and the sample rate is therefore only reduced to 87% (174 samples per interval). Using the dynamic sample rate adjustment, the sample rate is reduced on average to 59% (118 samples per interval), which results in an average runtime overhead of 1.7%.

Figure 8 shows the CCoV for all benchmarks when using both static sample rate (i.e. 200 samples for all interval) and dynamic sample rates adjustment. When computing the CCoV for dynamic sample rate, we group all unclassified intervals into a “junk” phase. This phase will have a large CoV, as its constituent intervals belong to many different phases, and undesired unclassified intervals therefore contribute to an increased CCoV. As the figure shows, using dynamic sample rate does not significantly increase the CCoV. The benchmark where the CCoV increase the most

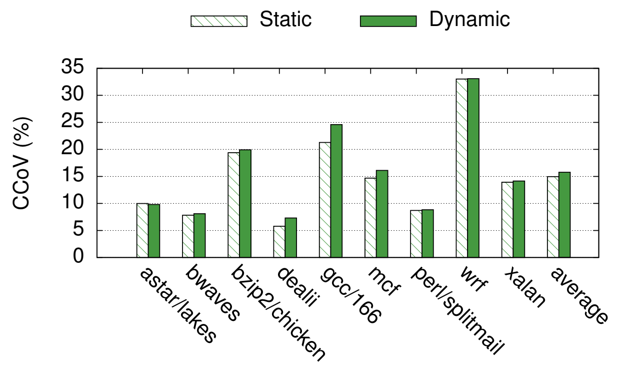


Figure 8. CCoV when using static sample rate (left bar) and dynamically adjusted sample rate (right bar).

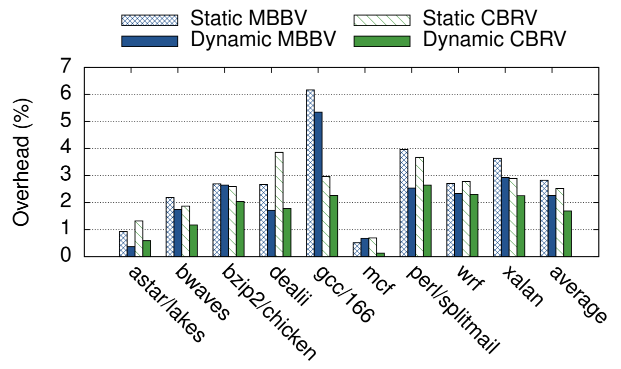


Figure 9. Runtime overhead in percentage, when using both static and dynamically adjusted sample rate to collect MBBVs and CBRVs.

is gcc/166, this is mainly due to gcc having the largest number of unclassified intervals, 1.7% of its intervals are unclassified. Across all benchmarks only 0.4% of the intervals are unclassified.

Figure 9 shows the runtime overhead for all benchmarks, for both static and dynamic sample rate, and for collecting both MBBVs and CBRVs. The figure clearly shows that using dynamic sample rate adjustment significantly reduced the runtime overhead for collecting both MBBVs and CBRVs. On average the runtime overhead is reduced by 20% and 33% for MBBV and CBRV respectively.

In summary, the results of this section shows that collecting CBRVs using dynamic sample rate outperforms all previous approaches, both in terms of accuracy and runtime overhead. This is clearly shown in Figure 6 where we have marked the average CCoV and runtime overhead of CBRV based phase classification using dynamic sample rate with an $\times$.

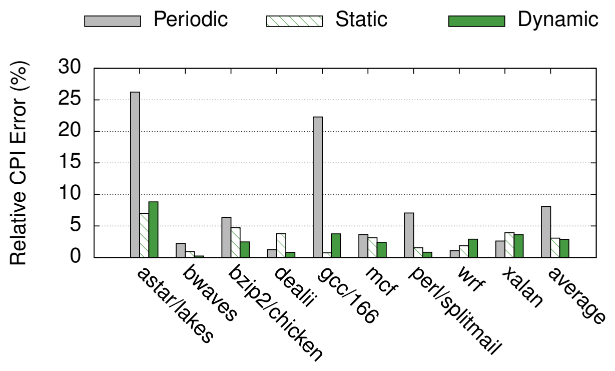


Figure 10. The relative error between the measured CPI and the estimated average CPI based on periodic sampling and phase guided profiling using both static and dynamically adjusted sample periods. Lower is better.

7. ScarPhase

The methods that have been described in this paper have been consolidated into an easy to use library. It is written in C/C++ and exposes a simple C interface. Two callback functions are used to notify the application of the program phase behavior. They are called when the program changes phase and after an interval has been executed. The callbacks pass along information on what phase the interval belonged to and a prediction of what phase the next interval will belong to. This makes it easy to plug the library into existing tools to take advantage of program phase behavior.

In the rest of this section we describe a practical usage study describing how ScarPhase is utilized to cheaply and accurately profile an application.

7.1. Use Case: Phase Guided Profiling

Profiling is commonly used for application performance tuning. A typical work flow might look as follows. First, the application is profiled to find performance bottlenecks, once found, the program is rewritten to remove the bottlenecks. Then the application is profiled again to verify that the changes to the application successfully removed the bottlenecks. This process is repeated until the program meets the desired performance goal. In this scenario, it is important that the profiling overhead is as low as it adds to the development cycle.

In this section, we used the ScarPhase library to implement a phase guided profiling tool which only profiles a single execution interval from each program phase. We compare its accuracy against a sampling approach which randomly selects a small subset of the execution intervals to profile.

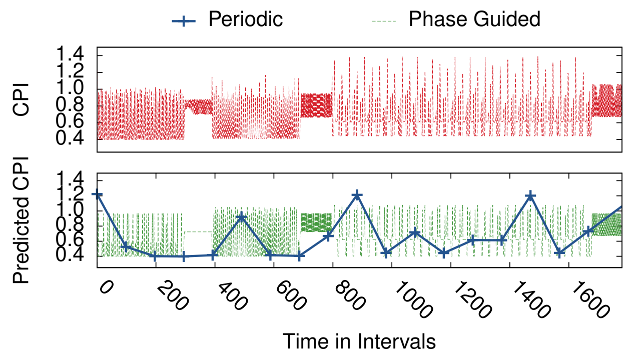


Figure 11. CPI changes over time for bzip2/chicken (top), and the reconstructed program execution (bottom) using periodic sampling (front, bottom) and phases guided sampling (back, bottom). It should be noted that the period from 800 to 1600 is not a single phase, but a series of smaller phases.

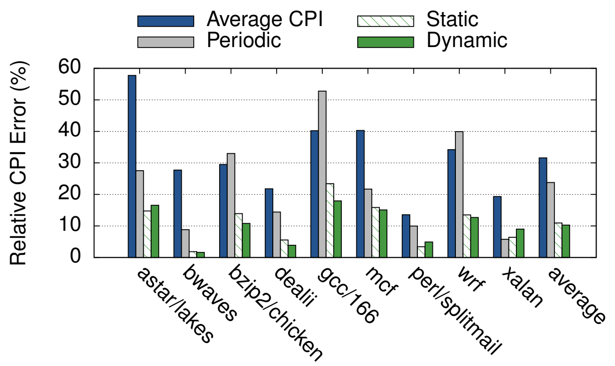


Figure 12. The average relative difference between the measured CPI and the predicted CPI for each interval. The executions were reconstructed using both periodic sampling and phase guided profiling with both static and dynamically adjusted sample periods. Lower is better.

To implement the phase guided profiling, we use the library to predict the next interval; the interval is then profiled if it belongs to a phase that has not yet been profiled. (In this example, we simply measure the CPIs of the intervals.) To avoid small phases that do not make a significant contribution to the average CPI, a phase is only profiled after it has been seen in a certain number of intervals. To estimate the overall CPI of the profiled application, we compute average CPI of the phases weighed with the number of intervals detected for each phase. (If the same phase was profiled more than once due to miss-prediction, we use the median CPI).

Figure 10 shows the relative error between the measured CPI and the estimated average CPI, for both periodic sampling[5] (Periodic) and phase guided profiling. The phase guided profiling was done using both static (Static) and dynamic (Dynamic) sample period adjustment. Phase guided profiling can accurately estimate the CPI with an average error of 3%, while periodic sampling has an average error of 8%. Furthermore, the accuracy of periodic sampling varies a lot between the applications. For example, gcc/166 has an error of 22%, while the error for wrf is only 1%. The standard deviation for phase guided profiling and periodic sampling is 2 and 9.5 respectively.

With phase guided profiling we can cheaply reconstruct the profiled application's execution behavior over time. This cannot be easily done using periodic sampling[6]. To do this, the CPI of each interval is predicted to be the same as the CPI of the phase it belongs to. (The CPI of a phase is the CPI of the profiled interval.) Figure 11 shows how the CPI changes over time for bzip2/chicken. The top figure shows the measured CPI, and the bottom figure shows the reconstructed CPI. The line labeled periodic in the bottom figure, shows an attempt to reconstruct the CPI using periodic sampling by interpolating between the profiled intervals. For phase guided profiling, only 0.7% and 0.8% of the execution intervals are profiled using static and dynamic sample period adjustment respectively, however the profiled intervals represent 90% and 91% of the application's execution.

Figure 12 shows average relative difference between the measured CPI and the predicted CPI for each interval. Using phase guided profiling with and without dynamic sample rate results in an error of 10% and 11% respectively, while periodic sampling has an error of 24%. Using the average CPI to predict the behavior for the whole execution results in an error of 32%. This shows that predicting the program's behavior with an average can be misleading.

In this section we profiled the CPI, however it should be noted that the technique is general purpose and applicable to any type of profiling. This shows that ScarPhase can be used for phase guided profiling and accurately reconstruct the program execution with only a small increase in overhead.

8. Related Work

In this paper we have focused on *temporal phases* [29, 8], however, there are alternative definitions of phases such as *code phases* [15, 12, 18, 13, 32]. Instead of observing the

[5] Both methods has roughly the same overhead, i.e., sample the same number of intervals, however phase guided profiling distribute the samples in a more efficient way.

[6] If every interval is profiled, the execution behavior can be reconstructed. However, only a few number of intervals are profiled to lower the overhead.

execution behavior over time, the program binary is analyzed, where parts of program's control flow is grouped into phases. For example, if the number of instructions in a function is above a threshold, the function is considered to be a phase. This has been used to instrument code at the phase boundaries for various optimizations. Both software [12, 13, 32] and hardware [15, 18] approaches have been suggested.

Gu and Verbrugge [13] used code phases to find the best optimization level for each function with respect to compilation/execution time for dynamic recompilation in Java virtual machines. Sondag and Rajan [32] instrumented phase boundaries for scheduling, and moved threads between heterogeneous processors when they changed phase. They found that phase guided scheduling can significantly improve the throughput compared to the standard Linux scheduler.

Temporal phases (see Section 2) divide the execution into intervals and group similar intervals into phases. This has been used to reduce the overhead of profiling [25] and finding representative part of the execution for architectural simulation [29, 30, 7, 28].

A sub set of temporal phase classification is phase change detection and predicting the applications performance behavior [9, 11, 17, 26]. Peleg and Mendelson [26] showed that changes in the CPI can not be used as a metric for loaded systems. Instead, BBVs, instruction working sets, and other architecture independent metrics have been used to detect performance behavioral changes. This is different from phase classification in that only the last intervals are of interest, i.e., the program changes phase if the difference between the last two intervals is above a threshold. However, remembering reoccurring phases can improve performance by reusing configuration settings [11, 17].

Dhodapkar and Smith [9] used instruction working set to reconfigure the size of caches at runtime when the application changes phase. Isci et al. [17] predicted the ratio between memory transactions and micro operations to reduce the power consumption using dynamic voltage frequency scaling.

9. Conclusions

This paper shows that it is possible to implement a low-overhead general purpose library for online phase classification and prediction without the need to add dedicated hardware support. Several new and existing options are evaluated and combined to reach this goal. We show that Intel's Precise Event Based Sampling can be used to sample basic block frequencies without any software mapping. We combine this finding with a new execution frequency metric based on conditional branch counters to improve accuracy and efficiency. A new dynamic sampling rate technique further brings down the runtime overhead to below two percent.

We anticipate a quick uptake of this software-only technique, since many existing phase guided optimizations typically gain more than 1.7% runtime overhead of ScarPhase. We are in the process of making our phase detection library for Sample-based Classification and Analysis for Runtime Phases, ScarPhase, available for free download.

References

[1] Linux perf_events. URL `Linux/include/linux/perf_event.h`.

[2] Intel vtune performance analyzer homepage. URL `http://www.intel.com/software/products/vtune/`.

[3] *Intel VTune Amplifier XE 2011 Getting Started Tutorials for Linux* OS*, 2010. Section Key Concept: Event Skid.

[4] M. Annavaram, R. Rakvic, M. Polito, J.-Y. Bouguet, R. A. Hankins, and B. Davies. The fuzzy correlation between code and performance predictability. In *Int. Symposium on Microarchitecture*, 2004.

[5] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W.-m. W. Hwu. Vacuum packing: extracting hardware-detected program phases for post-link optimization. In *Int. Symposium on Microarchitecture*, 2002.

[6] 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. In *Conf. on Programming Language Design and Implementation*, 2005.

[7] B. Davies, J. Bouguet, M. Polito, and M. Annavaram. ipart : An automated phase analysis and recognition tool. Technical Report IR-TR-2004-1-iPART, Intel Corporation, 2004.

[8] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Int. Symposium on Computer Architecture*, 2002.

[9] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Int. Symposium on Microarchitecture*, pages 217–, 2003.

[10] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*, chapter 10.11. On-line Clustering, pages 559–565. Wiley-Interscience, 2 edition, 2001. ISBN 0-471-05669-3.

[11] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Int. Conf. on Parallel Architecture and Compilation Techniques*, 2003.

[12] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level phase behavior in java workloads. In *Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[13] D. Gu and C. Verbrugge. Phase-based adaptive recompilation in a jvm. In *Int. Symposium on Code generation and Optimization*, 2008.

[14] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 2006.

[15] M. C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. In *Int. Symposium on Computer Architecture*, 2003.

[16] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, volume 3b: system programming guide edition, September 2010. 30.4.4 Precise Event Based Sampling (PEBS).

[17] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Int. Symposium on Microarchitecture*, pages 359–370, 2006.

[18] J. Kim, S. V. K. W. chung Hsu, D. J. Lilja, and P. chung Yew. Dynamic code region (dcr)-based program phase tracking and prediction for dynamic optimizations. In *Int. Conf. on High Performance Embedded Architectures and Compilers*. Springer Verlag, 2005.

[19] J. Lau, S. Schoemackers, and B. Calder. Structures for phase classification. In *Int. Symposium on Performance Analysis of Systems and Software*, 2004.

[20] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *Int. Symposium on Performance Analysis of Systems and Software*, 2005.

[21] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *Int. Symposium on Performance Analysis of Systems and Software*, 2005.

[22] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *Int. Symposium on High-Performance Computer Architecture*, 2005.

[23] D. Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. Technical Report Version 1.0, Intel Corporation, 2009.

[24] Y. Luo, V. Packirisamy, W.-C. Hsu, A. Zhai, N. Mungre, and A. Tarkas. Dynamic performance tuning for speculative threads. In *Int. Symposium on Computer Architecture*, 2009.

[25] P. Nagpurkar, C. Krintz, and T. Sherwood. Phase-aware remote profiling. In *Int. Symposium on Code generation and Optimization*, 2005.

[26] N. Peleg and B. Mendelson. Detecting change in program behavior for adaptive optimization. In *Int. Conf. on Parallel Architecture and Compilation Techniques*, 2007.

[27] C. Pereira, J. Lau, B. Calder, and R. Gupta. Dynamic phase analysis for cycle-close trace generation. In *Int. Conf. on Hardware/software Codesign and System Synthesis*, 2005.

[28] E. Perelman, M. Polito, J. yves Bouguet, J. Sampson, B. Calder, and C. Dulong. Detecting phases in parallel applications on shared memory architectures. In *Int. Parallel and Distributed Processing Symposium*, 2006.

[29] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Int. Conf. on Parallel Architecture and Compilation Techniques*, 2001.

[30] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[31] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. *SIGARCH Comput. Archit. News*, 2003.

[32] T. Sondag and H. Rajan. Phase-guided thread-to-core assignment for improved utilization of performance-asymmetric multi-core processors. In *ICSE Workshop on Multicore Software Engineering*, 2009.