The POP Detector: A Lightweight Online Program Phase Detection Framework

Karl Taht
Department of Computer Science
University of Utah
Salt Lake City, USA
taht@cs.utah.edu

James Greensky
Intel Labs
Intel
Hillsboro, USA
james.j.greensky@intel.com

Rajeev Balasubramonian
Department of Computer Science
University of Utah
Salt Lake City, USA
rajeev@cs.utah.edu

***Abstract*—Real-time phase detection enables dynamic adaptation of systems based on different program behavior. Many phase detection techniques have been proposed, with the most successful relating the phases back to application code. In the scope of online phase detection, techniques employ sampling to mitigate the overheads of the phase detection framework. When phase intervals are long enough, sampling approaches perform well.**

We reopen the question of phase interval length by performing in-depth analysis on the trade-offs between overhead and phase detector performance. We present a new metric which captures the statistical trade-off between phase interval length, phase stability, and the number of phases. We find that while shorter phases perform best in the context of online optimization, existing implementations suffer from performance degradation and overhead at shorter interval sizes.

To address this gap, we present the Precise Online Phase (POP) detector. The POP detector utilizes performance counters to build signatures, which are virtually lossless at finer granularity. As a second order, the simplicity of the detector reduces the runtime overhead to just 1.35% and 0.09% at 10M and 100M instruction intervals, respectively.

I. Introduction

Programs exhibit time-varying behavior and move between *phases*. This means that at any time, a system may not be optimally configured for the requirements of that phase. In response, architects have and will continue to develop adaptive systems that can better respond to specific application phases and conditions. At the lowest level, hardware features such as branch predictors and data prefetchers constantly monitor changing instruction and data streams to update their heuristic predictors. These online learning agents employ lightweight algorithms that can not only detect changing behavior within a matter of cycles, but also update their response within hundreds of cycles.

In particular, the update rules for these agents must be chosen through careful analysis across many workloads. One analysis technique is to generate an S-Curve, a graph which encapsulates the effects of changing a feature across many workloads. The high level goal is simple: maximize the speedup across all workloads. The S-Curve helps to identify outliers that may be skewing the average. In Figure 1, we utilize the S-Curve to show the effects of disabling the L2 spatial prefetcher on an Intel® system. On average, we find that the speedup is near one, but the S-Curve reveals both positive and negative outliers. On the left-hand side a number of workloads suffer from disabling the prefetcher, whereas on the right side a number of workloads prefer the change. A simple approach would be to tune the feature on a per-workload basis.

However, such an approach would ignore the dynamic behavior of a program during execution. Thus, we plot an additional S-Curve for the individual program phases of *lbm_s* from Spec2017 [1] in Figure 2. From a global perspective, *lbm_s* demonstrates a marginal performance improvement with the spatial prefetcher enabled. However, on a per-phase basis we see that this same program has both positive and negative outliers, motivating the need to adapt during particular intervals, or phases, of a program.

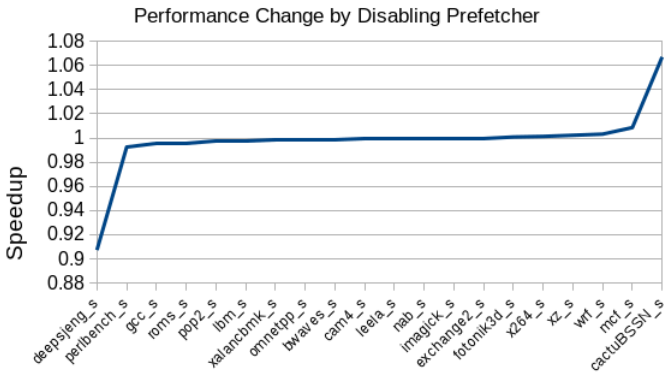

Fig. 1. Spec2017 performance change by disabling spatial L2 prefetcher, using training dataset.

Much work has been done to address this need for dynamic optimization, ranging from careful offline phase analysis techniques, to lightweight online phase detectors, and more recently control theory [2]–[6]. Each of these approaches has merit. Basic block vectors (BBVs), for example, have proved to be an excellent baseline that has stood the test of time for phase detection. Yet, this algorithm requires a large amount of data making it infeasible for online algorithms. As such, multiple prior works have attempted approximation of BBVs, originally through theoretical hardware support and

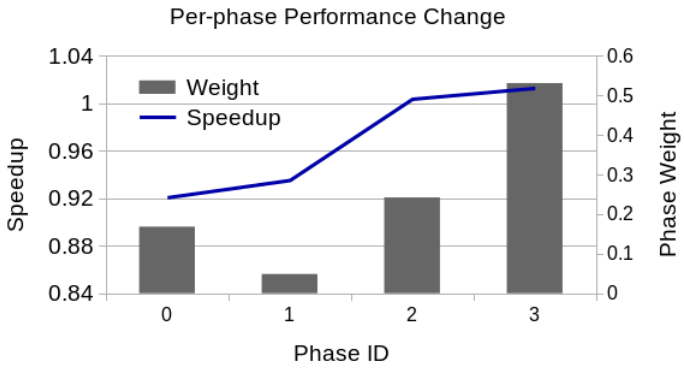

Fig. 2. The lbm_s performance change (denoted by the S-Curve) per phase via disabling spatial L2 prefetcher, with respective phase weights denoted by bars. The phase weight refers to the amount of time the program spends in a particular phase, relative to the program as whole. As a result, the sum of phase weights is always 1.

more recently on a real-system using Precise Event-Based Sampling (PEBS) [3], [4]. Ultimately, the goal remains to understand individual phases and optimize features on a per phase basis, which can be defined as a set of requirements.

Functionally, any phase detector must produce phases of similar, repeating behavior. However, using program phases for online performance optimization necessitates a number of requirements for a successful phase detection algorithm. Firstly, the phase detector must operate independently of performance changes. For a phase detector to enable analysis of different hardware configurations, program phases should remain unchanged with different configurations. In other words, the phase detector should remain *invariant* to performance knobs, i.e., phase A is still labeled phase A even after a change in a performance knob. Since both SimPoint and ScarPhase are based on code execution, they both satisfy this requirement.

From a feasibility perspective, a phase detection technique should be feasible in commodity hardware with minimal program and system perturbation. Collecting BBVs, the input for SimPoint, is infeasible in real-time. As such, detectors like ScarPhase employ sampling to mitigate this overhead. However, we find that even at large 100M instruction intervals, the overhead of such a technique is more than 3%, and if finer granularity is required, the overhead increases exponentially. This overhead is undesirable in the scope of online performance optimization.

In this work, we present the POP detector which drastically reduces phase detection overheads to as little as 0.09%, while performing competitively with existing state-of-the-art phase detection performance. Specifically, we outline our contributions as follows:

- We propose a new phase detector metric, Statistical Time Analyzing Baseline (STAB) which captures the trade-offs of phase interval length, phase stability, and the number of phases.
- We introduce the Precise Online Phase (POP) detector, a real-time phase detection algorithm which leverages performance counters for accurate, fine-grain phase detection with significantly lower overhead than existing approaches.
- We perform a detailed competitive analysis between the POP detector and the state-of-the-art ScarPhase detector, using existing metrics as well as our proposed metric.

We organize our paper as follows. In Section II, we discuss the different phase detection algorithms discussed throughout the paper, including the POP detector. The key distinguishing factor of the POP detector is the specific performance counters selected, which we detail in Section III. In Section IV, we describe the metrics for evaluating phase detection algorithms including our new metric, STAB. We then perform a case study regarding phase interval size in Section V, before performing extensive analysis between ScarPhase and the POP detector in Section VI. Finally, we provide a brief survey of related work in Section VII and conclude in Section VIII.

II. Phase Detection Algorithms

We briefly describe the phase detection algorithms used throughout the paper, including our proposed POP detection algorithm.

A. SimPoint: Offline Baseline

SimPoint traces code execution in order to classify a program into phases [7]. The original work stems from the notion of a basic block: a region of code with exactly one entry and one exit. A Basic Block Vector (BBV) is generated by profiling the number of instructions executed within each basic block for a period of time. SimPoint performs approximate K-Means clustering on the BBVs, and utilizes the Bayesian Information Criterion (BIC) to select the optimal number of phases for a program. It then outputs a number of BBVs which serve as simulation points. We treat each simulation point as a cluster center, which allows us to label each interval of execution according to the closest simulation point.

B. ScarPhase: Online Baseline

ScarPhase samples conditional branch instruction pointers (IPs) to detect program phases [3]. The framework is built utilizing the Linux *perf* subsystem and operates as a user-space program. Intel®'s Precise Event-Based Sampling (PEBS) enables sampling IPs directly at a granularity of up to once per 25K instructions. The branch IP's are hashed into a vector to construct a signature for each interval. The signature is fed to an online leader-follower clustering algorithm to identify program phases. In order to mitigate the overhead, ScarPhase makes use of a buffer in which the kernel writes directly that allows processing only at interval granularity. Additionally, ScarPhase employs a dynamic sampling frequency methodology which reduces the sample rate during stable phases of execution. As the dynamic sample rate is itself predictive, it slightly degrades phase detection accuracy in favor of lower overhead. All data reported relative to ScarPhase uses the authors' original code, available on GitHub [8].

C. The POP Detector

The POP detector was motivated by the non-trivial overhead associated with ScarPhase, as well as its performance degradation when using a phase length of less than 100M instructions. To address this, the POP detector utilizes only performance counters to detect program phases. The rationale for using performance counters is two fold. Firstly, interrupts are required only at the end of the interval to collect performance counts. This drastically reduces overhead, as we will show. Secondly, the counts are true measurements rather than samples. This means that when performing increasingly fine-grained phase detection, the underlying data does not incur additional loss.

The quality of phase detection using performance counters relies on selecting specific counters which capture code execution paths rather than performance related metrics. If the counters are related to performance, they create a feedback loop. Consider the prefetcher example, where the goal is to choose whether to enable or disable the feature. If one of the counters used to generate the signature was the number of cache misses, disabling the prefetcher would likely affect the cache miss count. As a result, the signature would change, and potentially trigger a false phase change. Therefore, the performance counters must relate specifically back to code execution paths in order to be agnostic to system changes. We dedicate Section III to our novel approach which targets optimal counter selection via machine learning.

Using this key set of performance counters, the POP detector periodically measures counter values. Similar to ScarPhase, this is done by leveraging the Linux *perf* subsystem in a user-space program. The counter values are treated as the interval's signature, and fed as input into an online leader-follower clustering algorithm. This lightweight online algorithm takes just 10,000 cycles, or about 5μs at 2GHz, to cluster a new point. The leader-follower algorithm automatically generates new clusters when the similarity between the newest data point and existing cluster centers is too large. We measure distance using average percentage difference, and set the similarity threshold to 20% (the same as ScarPhase). We allow the POP detector to track up to 32 unique phases simultaneously. If a new cluster is detected after 32 unique phases have been detected, we employ an LRU replacement policy to remove a cluster.

III. Performance Counter Selection

In this section we describe our process for selecting key performance counters which are able to capture program phases which relate back to repeating segments of code, rather than repeating behavior. Our two-step approach examines over 200 candidate counters, and ultimately provides a subset of just 8 through the use of machine learning techniques and a ground truth of SimPoint phases.

A. Performance Counter Collection

Other than limiting our search to per-core events, we offer no domain expertise to try to reduce the set of performance counters. This results in a list of over two hundred performance counters to profile for our Haswell-based Intel® system. Because the hardware offers only 8 events to be simultaneously collected, the most common approach is to group events and use time-multiplexing to approximate counts. However, because the goal of our study is to *avoid* sampling error, we instead collect over 50 separate traces, each with real counter values to eliminate the possibility of sampling error.

Counter	Normalized Score
UOPS_RETIRED.RETIRE_SLOTS	100
UOPS_RETIRED.ALL	93.28
BR_INST_RETIRED.ALL_BRANCHES	79.29
BR_INST_RETIRED.NOT_TAKEN	77.16
MEM_UOPS_RETIRED.ALL_STORES	74.58
BR_INST_RETIRED.NEAR_TAKEN	69.17
MEM_UOPS_RETIRED.ALL_LOADS	64.75
UOPS_EXECUTED.CYCLES_GE_4_UOPS_EXEC	64.60

TABLE I
THE HIGHEST RANKED COUNTERS USING THE GINI IMPURITY METRIC.

B. Ensuring Counter Invariance

Our first pass is to remove performance counters which correlate to system behavior rather than program execution. At first glance this may not seem intuitive, as similar system behavior implies similar code execution. While this is true, it is best to think of the program's instructions as the input, and the system's behavior as a result. A good phase detector correlates intervals which have similar *inputs*. This enables analysis of the output, including performance, in different system configurations. In other words, we seek to eliminate any performance counters which relate to performance.

To find a set of performance counters which relates back to code execution rather than system behavior, we perform a test to measure change as a result of system reconfiguration. Our system has four hardware prefetchers which can be enabled or disabled through a Model Specific Register (MSR) [9]. The default configuration, prefetch MSR $= 0$, enables all prefetchers, while the disabled configuration, prefetch MSR $= 15$, disables all prefetchers. We measure the average percentage difference of each performance counter on a per interval basis. For a run of N intervals, we measure the difference by comparing the counts on interval i in prefetcher configurations 0 ($c_{i,0}$) and 15 ($c_{i,15}$):

$$\text{Avg \% Diff} = \frac{1}{N}\sum_{i=1}^{N}\frac{|C_{i,0} - C_{i,15}|}{(1/2)(C_{i,0} + C_{i,15})}$$

Figure 3 shows a ranking of the top 50 performance counters which responded least to changes in prefetcher configuration, making them potential candidates. Many of the low variance counters relate back to instruction mix, as expected. In order for a program counter to advance to our next stage, we require less than 1% average percentage difference due to prefetcher reconfiguration.

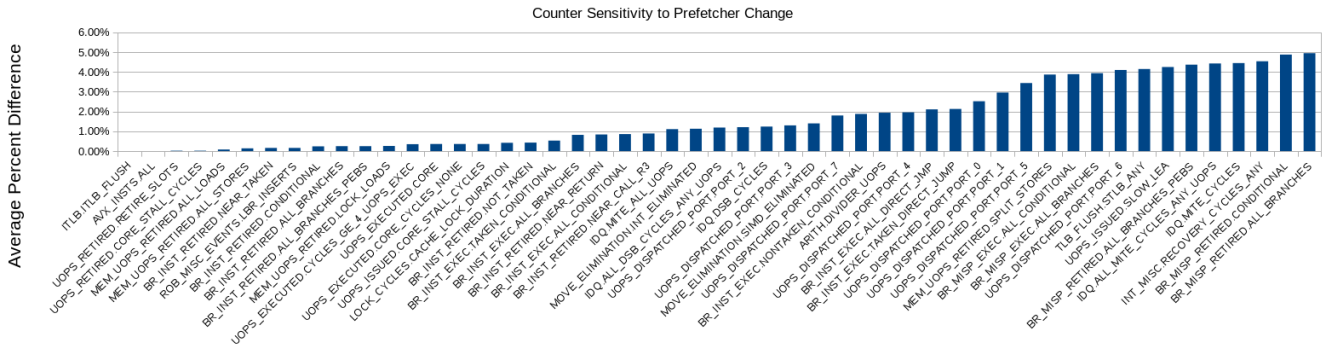

Fig. 3. Interval-to-Interval average percentage difference from varying prefetchers. Data collected using Spec2006 benchmarks.

C. Counter Pruning

Our final step ranks program counters by their ability to accurately predict phases based on code execution. We collect Basic Block Vectors (BBVs) and use SimPoint to determine ground truth program phases. A naive approach would be to train a series of predictors using every possible combination of eight counters, and select the counters with the best prediction accuracy. Unfortunately, this is infeasible from a compute perspective as choosing eight counters from just the 20 best candidates involves testing over 125,000 combinations. Moreover, such a method provides no insight as to a ranking of the counters.

Thus, we employ a decision forest classifier to predict the SimPoint phases. After training the decision forest, we can view the importance associated with each variable, measured using the Gini impurity metric [10]. Using this approach across Spec2006 benchmarks [11], we rank the variables as shown in Table I.

Lastly, we validate that the counters do in fact capture data similar to that of basic block vectors by performing a distance visualization experiment. Motivated by the original work by Sherwood et al. [12], we measure similarity between signatures at different intervals to find repeating patterns. A darker color indicates greater similarity, with black being an exact match. Similarly, white is a complete mismatch. The results for *mcf* workload are shown in Figure 4. Note that this figure captures only signature similarity over the course of program execution, and include no notion of program phase.

By examining the *mcf* visualization, it is obvious that performance counters capture the same pattern as BBVs. However, while BBVs can indicate complete mismatch, performance counters always retain some element of similarity (indicated by the gray in place of the white). As a result, the phases given by the performance counter signatures are more generalized than BBV phases. While the difference may seem significant, we will later show in the results section that the difference only results in slightly worse stability compared to IP-based approaches.

IV. Phase Detection Evaluation

In this section we present a number of metrics used for evaluating phase detection algorithms. We utilize CPI as the basis to compute all metrics. For example, μ_i refers to the standard deviation of CPI among all intervals labeled as i. Note that all computations presented are done post-execution as a means to compare program phase detectors. Many of such metrics could be adapted to be computed online, but to remain consistent with prior art and for a fair competitive analysis, we utilize these metrics a posteriori.

A. Phase Stability

A method for measuring the quality of a phase detection algorithm is to look at the stability (a percentage). Stability is a statistically grounded metric, which measures the *Coefficient of Variation* (CoV) within each phase. CoV is simply a normalized method to compute variance, more formally,

$$\text{CoV} = \frac{\sigma}{\mu}$$

The stability metric also factors duration of the program spent in each phase, or the phase's *weight*. For N phases, the stability can then be computed as:

$$\text{Stability} = 1 - \sum_{i=1}^{N} \text{CoV}_i * \text{Weight}_i$$

However, a naive phase detection algorithm could classify each interval as a unique phase, pushing the stability to 100%. To correct this, Sembrant et al. [3] proposed the *Corrected Coefficient of Variation* (CCoV). CCoV penalizes algorithms for detecting more unique phases by adjusting the basic CoV computation. CCoV places all phases which are dissimilar to their neighbors as part of a virtual phase. All samples in the virtual phase are given the CoV of the program as a whole. This gives the desirable property that in both trivial cases, a single phase or all unique phases, the CCoV will converge to the same value.

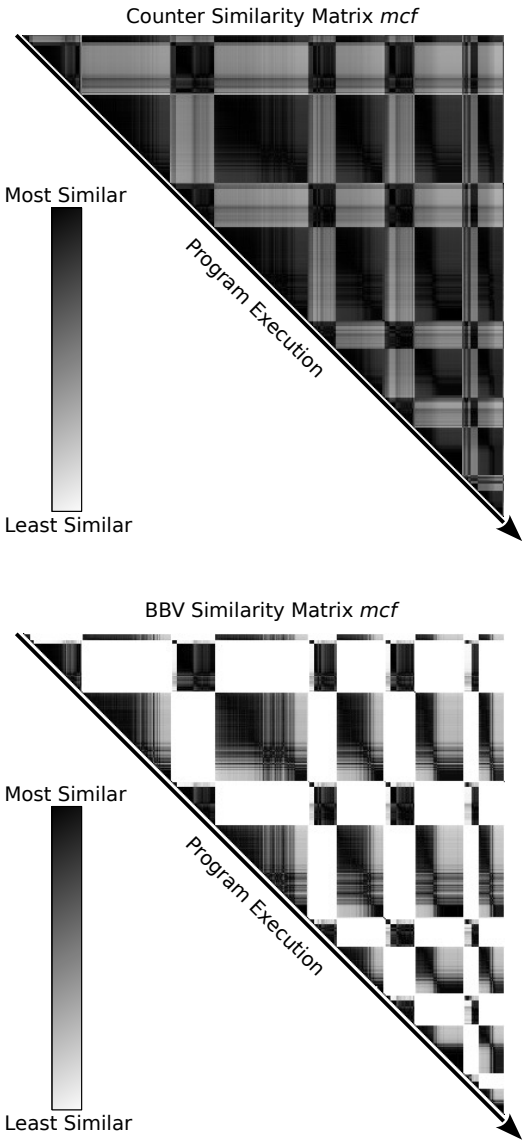

Fig. 4. Similarity matrices show the similarity between interval signatures, where the diagonal of the matrix represents the program's execution. A darker color indicates greater similarity, as measured by percentage distance. We show the similarity matrix for counters (top) and BBVs (bottom) for the Spec2006 *mcf* benchmark.

B. Phase Interval Length

The previous metric may be affected by phase interval length, but do not use it as a weighting metric to favor smaller or larger intervals. As such, we propose a new metric: *Statistical Time Analyzing Baseline* (STAB). This metric examines the balance between number of phases, stability, *and* phase interval length. STAB is a metric with the application of phase detection algorithms for real-time optimization in mind.

In order to dynamically tune a system, performance must be assessed in a given configuration. Consider an algorithm which must choose between the default configuration A and a new configuration B. In order for the algorithm to move to configuration B, the performance difference must exceed measurement error for the change to be reasonable. Thus, we may need to sample both configuration A and B multiple times to establish a small enough margin of error with enough confidence.

For clarity, we describe a specific example. Consider that a program has $3\times$ more stability at 100M instruction intervals versus 10M instruction intervals. There exists a quadratic relationship between stability and the number of samples required to establish a given confidence interval and a particular error tolerance [13]. Given that, we can compute that at 100M instruction intervals, x^2 samples are required, whereas at 10M instruction intervals, $(3x)^2$ samples are required. This means for the 100M instruction intervals, we would spend $100x^2$ instructions establishing a baseline, but for 10M instruction intervals, we would only spend $90x^2$ instructions. As we show in the results sections, this leads to some non-linear trade-offs with shorter phases.

We begin the formulation of STAB by computing the number of samples needed for a given confidence interval and error tolerance by the formula:

$$\text{Num Samples} = \left(\frac{Z * CoV}{Error}\right)^2$$

where Z is a factor related to the confidence interval (e.g. 95% confidence interval Z-value is 1.96) [13]. STAB calculates the number of samples required for a given confidence requirement and error tolerance on per-phase basis. These samples are then summed for each phase seen to give the total number of samples required to establish a performance baseline. Thus for a program with N phases:

$$\text{Baseline Samples} = \sum_{i=1}^{N} \left(\frac{Z * CoV_i}{Error}\right)^2$$

Next, we account for phase interval size, to determine the total number of instructions the program would need to spend establishing this baseline. This is simply baseline samples multiplied by the phase interval size. Finally, the Statistical Time spent Analyzing a Baseline (STAB) can be computed as a ratio:

$$\text{STAB} = \frac{\text{Baseline Samples} * \text{Interval Size}}{\text{Total Instructions}}$$

The result of this computation is an intuitive ratio: how many instructions (as a percentage) would be spent establishing a confidence interval with a given error margin. However, this ratio is affected by the total duration of the program, an undesirable artifact. For instance, Spec2017 *omnetpp* has just one phase, and requires 50 samples to establish baseline performance using either the training or reference dataset[1]. However, because the training set is smaller and has fewer instructions than the reference set, the unnormalized version

[1]This measurement was performed using ScarPhase with 100M instruction intervals and dynamic sampling enabled.

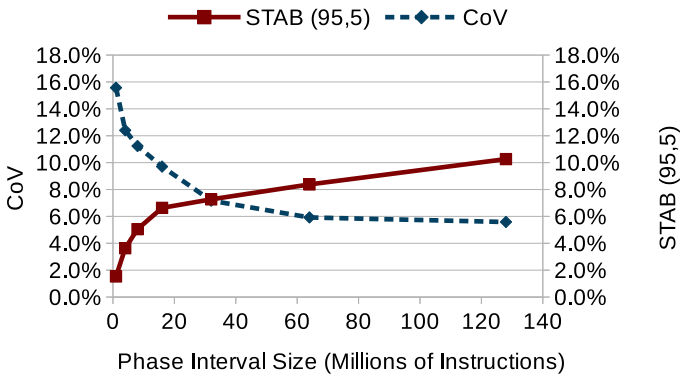

Fig. 5. Trade off between CoV and STAB using various interval sizes. While CoV suffers at smaller intervals, many more intervals are available for analysis. As a result, the STAB overhead reduces at smaller intervals.

of STAB is 7.3% for the training dataset, but just 0.91% for the reference dataset. Thus, we choose to normalize by a fixed number of instructions so that the total run length of the program does not affect STAB.

$$\text{STAB}_{norm} = \frac{\text{Interval Size} * \text{Baseline Samples}}{\text{100B Instructions}}$$

Using this formulation, *omnetpp* has a STAB_{norm} of 5% with either the training or reference datasets. For the remainder of the paper, we utilize the normalized version of STAB with 100B instructions as the normalization factor. This choice is arbitrary and has no bearing on the final results as it only linearly scales the values. We report normalized STAB for a given confidence interval c and error tolerance e as $\text{STAB}_{c,e}$. For example a confidence interval of 95% and error tolerance of 5% would be $\text{STAB}_{95,5}$.

V. Case Study: Interval Size

Utilizing our STAB metric, we perform an analysis to understand the trade-offs of phase detection at various interval sizes. To create a fair baseline, which does not lose information or incur overhead, we perform trace-based analysis using SimPoint, as described in section II-A. We utilize QEMU [14], a full-system emulator, to collect accurate, complete traces of long-running programs. Once an instruction trace is collected, we can generate BBVs and corresponding phase labels for any desired interval size.

To perform analysis with CoV and STAB we require performance data, specifically the IPC for each interval. We collect 50 performance traces per workload and average the per interval data to mitigate run-to-run error. Using 19 Spec2006 workloads with the training dataset, we perform the SimPoint analysis at instruction intervals ranging from 1M to 128M. The results are shown in Figure 5.

Firstly, note that both CoV and STAB are lower-is-better metrics. As expected, the smaller intervals have an increasingly high CoV. However, this is non-linear in respect to the number of intervals available for analysis. While COV

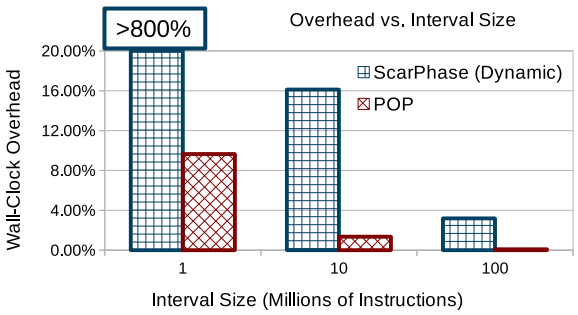

Fig. 6. Wall-clock Overhead comparison. Results averaged across Spec2017 benchmarks using the training dataset.

increases from 4.5% to over 15%, the number of intervals available increases by 128×. As a result, the $\text{STAB}_{95,5}$ metric decreases from 10% at 128M instruction intervals, to just 1.9% at 1M instruction intervals.

While the STAB metric conveys the benefits of utilizing smaller granularity phases, it neglects to consider overhead. We define overhead as the additional time required for the program to execute with a phase detection framework versus without any monitoring framework. We analyze the overhead of both ScarPhase and the POP phase detector at 1M, 10M, and 100M instruction interval sizes utilizing wall-clock time and plot the results in Figure 6. At 100M instruction intervals, we measured ScarPhase to incur 3.19% overhead on average across Spec2017 speed benchmarks, whereas the POP detector incurs just 0.09% overhead. At 10M instruction intervals this overhead rises to an undesirable 16.12% for ScarPhase, but just 1.35% for the POP detector. While the STAB metric continued to improve at smaller intervals, even the more efficient POP detector causes a 9.65% increase in execution time at 1M instruction intervals. Because the goal of the POP detector is low overhead, we do not present 1M instruction interval results in the following section.

VI. Results

A. Methodology

To assess the proposed POP detector, we compare it to the prior art, ScarPhase, utilizing the Spec2017 benchmark suite on an Intel® system[2]. Both phase detectors are configured with a similarity threshold of 20% for their clustering algorithm, the default specification for ScarPhase. To obtain consistent, repeatable results we evaluate single-threaded applications via *speed* benchmarks. It is also known that single threaded applications typically have more phase variability, making them more applicable to our analysis [4]. Furthermore, we apply a few command line boot parameters to ensure that results are not skewed. Table II details each of our Intel® Haswell evaluation node's specifications.

Note that both the POP detector and ScarPhase are implemented as user-space programs which tap into the Linux

[2]Performance results are based on testing as of February 13, 2019 and may not reflect all publicly available security updates.

Base Board	Intel® Server Board S2600WT2
CPU	2x Intel® Xeon® CPU E5-2699 v3 @ 2.30GHz
BIOS	SE5C610.86B.01.01.0027.071020182329
DRAM	24x 16G DDR4 ECC DIMM @ 1600 MHz
OS	CentOS 7.5 w/ Linux 4.14.58
Kernel Cmdline	nmi_watchdog=0 transparent_hugepages=never cpuidle.off=1 intel_idle.max_state=0 cpufreq.off=1 intel_pstate=disable procesor.max_state=0 isolcpus=0,1 rcu_nocbs=0,1 rcu_nocb_poll nohz_full=0,1 skew_tick=1
Hyperthreading	Off

TABLE II
EVALUATION SYSTEM

perf subsystem, allowing them to collect counter and branch IP information. This means that no super-user privileges are required to run either detector. While kernel-module implementations can offer superior performance, the user space implementation allows us to perform a fair comparison to the prior art. As Sembrant et al. [3] mention, context switches to this user space implementation can incur significant overhead at smaller time scales but is often more desirable for system administrators.

B. Phase Detector Performance

We perform an in-depth comparison of ScarPhase and the POP detector. As discussed in Section II-B, ScarPhase has an optional dynamic sampling mode which reduces overhead while slightly degrading phase detection performance. To remain succinct and capture ScarPhase's optimal phase detection ability, we focus our discussion of results on ScarPhase without dynamic sampling. We report results for both 100M and 10M instruction intervals, the smallest granularity possible without significant overhead. We include all *speed* benchmarks in the Spec2017 benchmark suite, which includes some outliers. As such, we utilize the geometric mean to draw final conclusions for various metrics. Note that when we use the term "average", we are referring to the geometric mean unless otherwise stated.

a) Stability: We begin by examining phase stability in Figure 7. The POP detector finds 0.51% more stable phases at 100M instruction intervals. In particular, POP performs significantly better on lbm and fotonik3d, where its stability is more than 10% higher than ScarPhase at 100M instruction intervals. These programs are among the most variable with respect to IPC changes in the Spec2017 benchmark suite, but ScarPhase inadequately separates the phases. In the case of fotonik3d at 100M instruction intervals, ScarPhase generates 112 unique phases, but more than 85% of intervals reside in just 3 phases. As a result, the phases are poorly separated and phase stability is just 80.5%.

At 10M instruction intervals, ScarPhase is able to identify phases which are 0.52% more stable compared to the POP detector. Nevertheless, for half of the benchmarks tested the POP detector is within a 2% margin compared to ScarPhase, and never performs more than 7% worse than ScarPhase. Also worth emphasizing is that this result is with ScarPhase's dynamic sampling mode disabled. Enabling dynamic sampling mode nets a 2% decrease in average stability for ScarPhase, placing it behind the POP detector in terms of phase stability.

b) Number of Phases: While the two detectors perform similarly from a phase stability standpoint, they begin to diverge when analyzing the number of phases detected, as shown in Figure 8. At 100M instruction intervals, ScarPhase detects 13.7 phases compared to 18.5 with the POP detector. Yet the reverse is true at 10M instruction intervals, where ScarPhase detects 214 unique phases compared to POP's 86, a reduction of nearly 60%. Specifically, for 12 of the 20 benchmarks tested, ScarPhase finds more than 100 unique phases. While more unique phases at finer granularity is intuitive, the divergence of ScarPhase and the POP detector can be traced back to the fundamental difference of sampling versus measurement. ScarPhase's signature vector is more susceptible to noise in the case of finer granularity due to its lack of sufficient samples. This result is particularly prominent in x264 and wrf, where the number of phases detected increases by more than an order of magnitude.

c) CCoV: Sembrant et al. proposed the "Corrected" Coefficient of Variation as a means to account for the intrinsic trade-off between number of phases and stability. We compute the CCoV and report the results in Figure 9. The CCoV metric most emphasizes the effectiveness of the POP detector at smaller phase intervals. ScarPhase's CCoV degrades by 83% at finer granularity due to the large number of phases detected. The lower-is-better metric increases from 8.12% to 14.93% when moving from 100M to 10M instruction intervals using ScarPhase. By comparison, the POP detector experiences negligible deviation in CCoV, improving from 8.65% to 8.49% when moving from 100M to 10M instruction intervals.

d) STAB: While CCoV balances number of phases and stability, the STAB metric applies statistics to also balance phase interval size. We report the normalized STAB metric using a 95% confidence interval and 5% error tolerance in Figure 10. The overall winner here is the POP phase detector with 10M instruction intervals. ScarPhase's optimal configuration using 100M instruction windows boasts a $\text{STAB}_{95,5}$ of 26.5%, 2.1× worse than the POP detector's 12.5% $\text{STAB}_{95,5}$ with 10M instruction intervals. Based on our statistically grounded metric, this means use of the POP detection framework would net significantly quicker analysis to enable better online optimization.

C. Summary of Results

We provide a summary of results in Table III to highlight the key takeaways of the POP detector compared to the existing state-of-the-art, ScarPhase. All values reflect the geomean across all benchmarks tested; the same data is presented in Figures 6, 7, 8, 9, and 10. While the two phase detection frameworks are closely matched in the traditional phase detection metrics, the two parameters that apply most directly to the success of deploying an online phase detection algorithm are execution overhead and STAB. For any potential performance gain, the optimization will have to overcome the overhead of the phase detection framework. Our experiments concluded

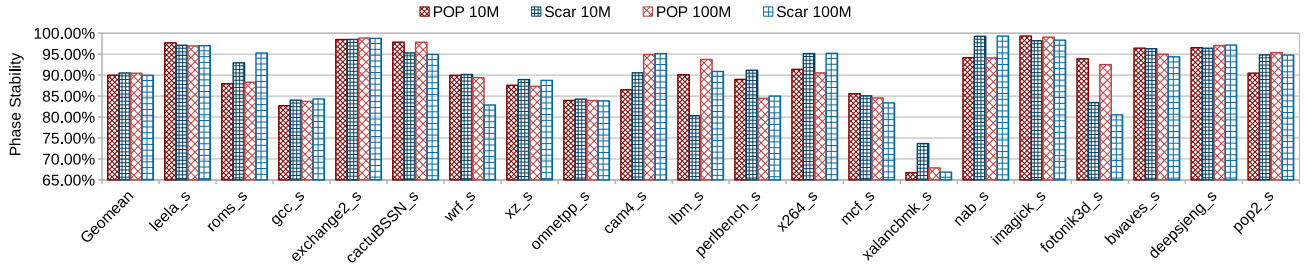

Fig. 7. Phase Stability metric for Spec2017 Speed benchmarks, Reference Dataset.

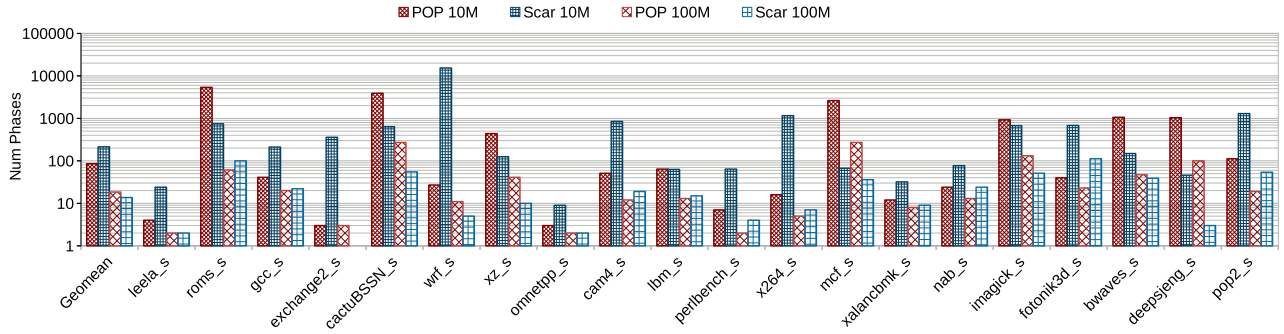

Fig. 8. Number of Phases detected for Spec2017 Speed benchmarks, Reference Dataset.

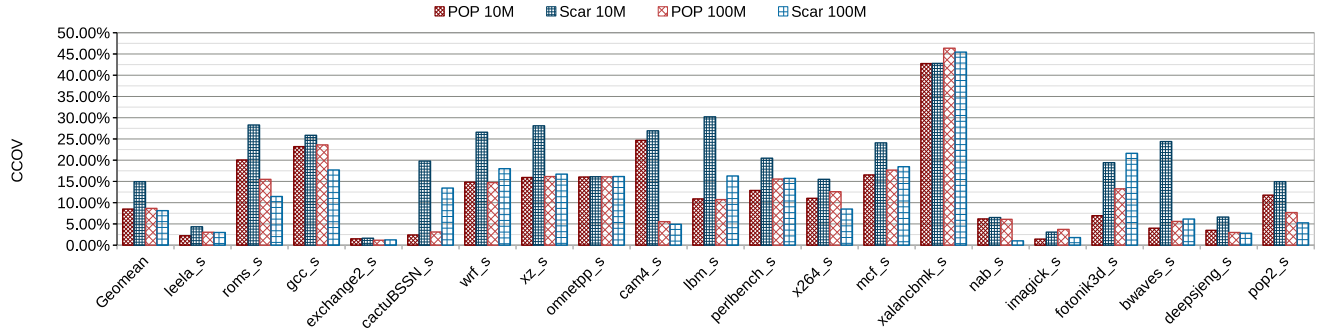

Fig. 9. Corrected Coefficient of Variation (CCoV) metric for Spec2017 Speed benchmarks, Reference Dataset.

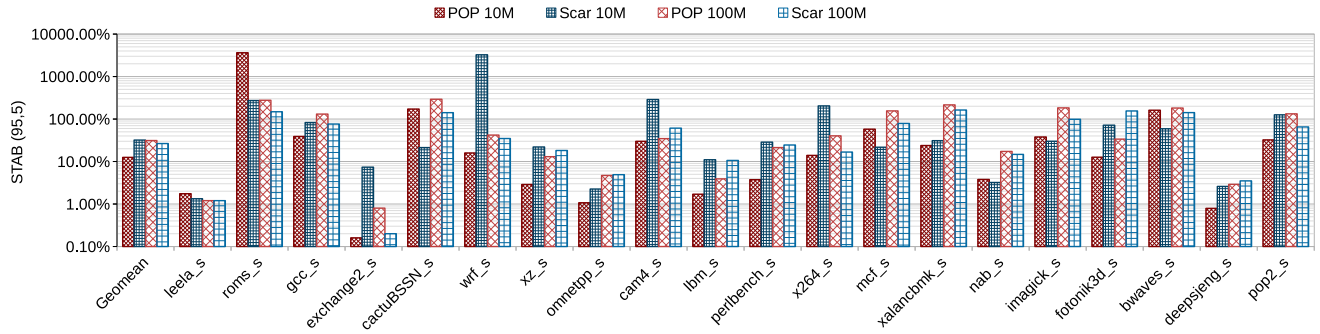

Fig. 10. Normalized $\text{STAB}_{95,5}$ metric for Spec2017 Speed benchmarks, Reference Dataset.

that even in the most optimistic scenario with dynamic sampling enabled and larger 100M instruction intervals, ScarPhase still incurs a 3.19% overhead. POP detection incurs just 1.35% execution overhead at shorter 10M instruction intervals, and just 0.09% with 100M instruction intervals.

In addition to overhead, an adaptive system must adequately analyze performance before applying a final configuration. Motivated by this, we propose the STAB metric to statically quantify the amount of time an adaptive system must spend analyzing behavior to gain a particular level of confidence with a given error tolerance. This metric captures trade-offs of phase stability, number of phases, *and* phase interval size. STAB quantifies that POP detection enables 2.1× faster performance analysis compared to ScarPhase.

Geomean	Overhead	Stability	Phases	CCoV	STAB
POP 10M	1.35%	89.99%	85.8	8.49%	12.53%
Scar-D 10M	16.12%	88.39%	195.2	16.72%	53.30%
Scar 10M	17.85%	90.51%	213.9	14.93%	31.98%
POP 100M	0.09%	90.45%	18.5	8.49%	31.18%
Scar-D 100M	3.19%	88.62%	12.7	9.57%	35.04%
Scar 100M	9.01%	89.94%	13.7	8.12%	26.53%

TABLE III
SUMMARY OF RESULTS FOR PHASE DETECTORS AT BOTH 10M AND 100M INSTRUCTION WINDOW SIZES. SCARPHASE WITH DYNAMIC SAMPLING IS LISTED AS *Scar-D*, WHILE WITH FIXED SAMPLING AS JUST *Scar*. OVERHEAD IS REPORTED IN WALL-CLOCK EXECUTION TIME, AND THE *STAB* METRIC REFERS TO 100B INSTRUCTION NORMALIZED *STAB* WITH A CONFIDENCE INTERVAL OF 95% AND ERROR TOLERANCE OF 5%, AS DESCRIBED IN IV-B.

VII. RELATED WORK

Sherwood et al. began to define the standards for program phase detection in their early work which utilized basic blocks. Their initial work did an in-depth dive to understanding the cyclical behavior of a program via frequency analysis [12]. They later expanded this work to SimPoint, which attempts to cluster basic vectors to define phases [15]. SimPoint's main objective is to speed up simulation work by drastically reducing the amount of a program that must be simulated to gain an accurate representation. SimPoint identifies phases and their respective weights such that a sample from each phase can can be used for simulation. More than a decade later, this technique is still proven to be effective as shown in [16].

While random projections and approximate K-means clustering help to speed up SimPoint, the fundamental idea still hinged on collecting basic blocks which are expensive to profile. Dhodapkar and Smith attempted to mitigate this overhead by proposing a hardware mechanism which constructs signatures hashing branches into an n-bit vector [17]. Sherwood et al. later refine this proposal by adding in the amount of time spent in each branch as well as a phase predictor [2]. Dhodapkar and Smith then provided an evaluation framework utilizing sensitivity, false-positives, and stability which lay the foundation and motivation for CCoV and STAB [18]. In their comparison work, they find that while BBVs provide the best phase detection, utilizing branch counts nets a phase detection with 80% of the performance. In many ways, this is the motivation for our work.

Shen et al. explore the notion of phases as repeating, but non-uniform behavior [19]. They utilize wavelet filtering and allow for variable phase sizes. While their technique performs comparably to manual-code injection, it requires a separate training step to determine program phases. Nagpurkar et al. also approach the problem utilizing dynamic phase sizes by suggesting the notion of stable and transition periods [20]. They use an adaptive trailing window policy to detect phases and instrument a new binary. However, their technique is only applicable to Java applications. Nevertheless, both of these approaches are particularly applicable when the goal is optimizing software as the program can be run and profiled many times. Since these publications, Linux *perf* has improved dramatically, and supports such analysis. Gregg has an excellent blog which includes many tutorials for such software optimization using *perf* and other tools [21].

However, another application of phase detection is online hardware optimization, which requires lightweight online phase detection. In this scope, ScarPhase [3] represents the state-of-the-art. ScarPhase bridges the gap of previous phase detection techniques by designing a framework which works on present hardware, operates completely online without prior training, and has significantly lower overhead than previous approaches [3]. However, ScarPhase chooses 100M instruction window phases as a baseline by citing prior work. We differ from ScarPhase in that we explore trade-offs associated with different phase sizes. We show that while variance increases with finer resolution, the trade-off is nonlinear and has auxiliary benefits.

VIII. CONCLUSION

In this paper we explored program phase detection for the use case of online performance optimization. To perform real-time optimization, an algorithm must be able to evaluate per-phase performance in a particular configuration. The faster per-phase performance can be evaluated with a phase detection algorithm, the quicker a reinforcement learning agent can adapt the system. With this in mind, we create a new metric, STAB, which weights phase stability, number of phases, and interval size. It statistically grounds time required for a given algorithm to establish per-phase performance.

Utilizing STAB, we perform an oracle study using SimPoint and show that smaller phases are desirable to minimize learning overhead. However, we find that the existing state-of-the-art phase detector, ScarPhase, suffers from excessive run-time overheads when attempting fine-grain phase classification. In addition to overhead, ScarPhase's sampling techniques break down due to limited information, resulting in hundreds of unique phases being identified. To attempt to fill this void, we explore the use of performance counters for online phase detection.

We employ statistical and machine learning techniques in a two-step process to select a core subset of performance counters. Using these counters, we build the POP phase detector, which is able to accurately detect phases at fine granularity while incurring just 1.35% overhead. In the context

of online optimization, the POP detector requires 2.1× fewer instructions to establish baseline performance. If even lower overhead is required, POP imposes just 0.09% overhead when using 100M instruction interval phases. In the future we would like to test the POP detector with more benchmarks and system configurations to further validate its efficacy.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful suggestions. This work was funded in part by Intel.

REFERENCES

[1] Standard Performance Evaluation Corporation, "SPEC CPU 2017," https://www.spec.org/cpu2017/.

[2] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2. ACM, 2003, pp. 336–349.

[3] A. Sembrant, D. Eklov, and E. Hagersten, "Efficient software-based online phase classification," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 104–115.

[4] A. Sembrant, D. Black-Schaffer, and E. Hagersten, "Phase behavior in serial and parallel applications," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2012, pp. 47–58.

[5] R. P. Pothukuchi, A. Ansari, P. Voulgaris, and J. Torrellas, "Using multiple input, multiple output formal control to maximize resource efficiency in architectures," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 658–670.

[6] R. P. Pothukuchi, S. Y. Pothukuchi, P. Voulgaris, and J. Torrellas, "Yukta: multilayer resource controllers to maximize efficiency," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 505–518.

[7] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '03. New York, NY, USA: ACM, 2003, pp. 318–319. [Online]. Available: http://doi.acm.org/10.1145/781027.781076

[8] A. Sembrant, D. Eklov, and E. Hagersten, "Scarphase," https://github.com/uart/scarphase, 2012.

[9] "Disclosure of H/W prefetcher control on some Intel processors," https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors, accessed: 2018-10-12.

[10] P. Dollár, Piotr and L. C. Zitnick, "Structured forests for fast edge detection," in *Proceedings of the IEEE International Conference on Computer Vision*, 2013, pp. 1841–1848.

[11] Standard Performance Evaluation Corporation, "SPEC CPU 2006," https://www.spec.org/cpu2006/.

[12] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*. IEEE, 2001, pp. 3–14.

[13] G. D. Israel, "Determining sample size," 1992.

[14] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247401

[15] T. Sherwood, E. Perelman, G. Hamerly, and B . Calder, "Automatically characterizing large scale program behavior," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5, pp. 45–57, 2002.

[16] Wu, Qinzhe and Flolid, Steven and Song, Shuang and Deng, Junyong and John, Lizy K, "Hot Regions in SPEC CPU2017," in *Invited Paper, Special Session on Hot Workloads, IEEE International Conference on Workload Characterization (IISWC)*, 2018.

[17] A. S. Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. IEEE, 2002, pp. 233–244.

[18] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003, p. 217.

[19] X. Shen, Y. Zhong, and C. Ding, "Locality phase prediction," *ACM SIGPLAN Notices*, vol. 39, no. 11, pp. 165–176, 2004.

[20] P. Nagpurkar, P. Hind, C. Krintz, P. Sweeney, and V. Rajan, "Online phase detection algorithms," in *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*. IEEE, 2006, pp. 13–pp.

[21] B. Gregg, *Systems performance: enterprise and the cloud*. Pearson Education, 2013.