Detecting Recurrent Phase Behavior under Real-System Variability

Canturk Isci and Margaret Martonosi
Department of Electrical Engineering
Princeton University
{canturk,mrm}@princeton.edu

Abstract

As computer systems become ever more complex and power hungry, research on dynamic on-the-fly system management and adaptations receives increasing attention. Such research relies on recognizing and responding to patterns or phases in application execution, which has therefore become an important and widely-studied research area.

While application phase analysis has received significant attention, much of this attention thus far has focused on simulation-based studies. In these cycle-level simulations without indeterministic operating system intervention, applications display behavior that is repeatable from phase to phase and from run to run. A natural question, therefore, concerns how these phases appear in real system runs, where interrupts and time variability can influence the timing and behavior of the program.

Our work examines the phase behavior of applications running on real systems. The key goals of our work are to reliably discern and recover phase behavior in the face of application variability stemming from real system effects and time sampling. We propose a set of new, "transition-based" phase detection techniques. Our techniques can detect repeatable workload phase information from time-varying, real system measurements with less than 5% false alarm probabilities. In comparison to previous value-based detection methods, our transition-based techniques achieve on average 6X higher recurrent phase detection efficiency under real system variability.

1 Introduction

Phase behavior in application characteristics has long been observed and exploited [8]. In recent years, application phase behavior has seen growing interest with two main goals. Some seek to identify program phases in order to select representative points within a run to study or simulate [2, 14, 21, 24, 26, 27]. Others seek to recognize phase shifts on-the-fly in order to perform optimizations such as dynamic adaptations in cache organization, voltage/frequency scaling, thermal management, or even dynamic compiler optimizations of hotcode regions [3, 4, 10, 16, 19, 29].

Most of the recent phase analysis work has focused on simulation studies; here the largely repeatable and deterministic behavior means that phases can stand out quite clearly. In order to move towards using on-the-fly phase analysis broadly in real systems, it is important to understand how these effects manifest themselves in more comprehensive measurements. Recent work shows the degree of time and space variability visible in real-systems that is generally not captured in simulations [1, 23]. This variability can stem from changes in system state that can alter cache, TLB and I/O behavior, system calls or interrupts, resulting in noticeably different timing and power/performance behavior.

In this paper, we characterize application phase behavior as seen from real-system performance monitoring counter (PMC) measurements. We start from previously proposed performance-counter sampling [18] and phase analysis [17] techniques. We discuss the repeatability of phase extraction experiments from run to run on a real system, and we demonstrate the extent and type of alterations an application can experience in different experiments. We categorize these alterations as *time shifts*, *time dilations*, and *phase mutations*, as well as transitional *glitches* and *gradients*. We propose a transition-based phase characterization scheme and then develop and evaluate effective methods for recognizing phases under these alterations. Specifically, we use glitch/gradient filtering to handle sampling effects and to focus on true phase transitions. We use correlation and blurring techniques to identify time shifts and dilations in the measured data. We test these, with a step-by-step phase recognition system, on several SPEC2000 benchmarks and common desktop applications.

There are four primary contributions of this work. First, our work presents a taxonomy of real system effects on phase behavior based on our application measurements. Second, we propose a transition-based phase characterization that proves to be more effective in phase detection under variability. Third, we present a complete flow of methods to recognize phases under variability and sampling effects. Fourth, we provide a quantitative evaluation of these techniques on a variety of benchmarks and demonstrate their effectiveness in phase recognition. In particular, our transition-based approach can detect recurrent workload phase behavior with less than 5% false alarms under real-system variability.

The remainder of this paper is structured as follows. Section 2 provides a detailed discussion of real system variability. Section 3 describes our phase analysis methodology and the impact of variability on phase behavior. Section 4 introduces our transition based phase representation and compares this to original phase sequences. Section 5 demonstrates our techniques to handle different variability effects. Section 6 evaluates the success of our methods in detecting recurrent work. Section 7 describes related work. We envision various applications to our detection methods under workload directed dynamic management. We discuss these and future research directions in Section 8. Finally, Section 9 offers our conclusions.

2 Real-System Behavior Variability

In order for a phase technique to be applicable on a real system, the phase characterizations of applications should

lead to similar classifications across different runs. That is, a fundamental check of a phase analysis technique on real systems is whether repeated runs of the same application give similar metric behavior and phase distributions. Differences upon repetition can arise from time-alignment of sampling with the execution, as well as with system calls and memory/IO effects. Nonetheless, in most cases, we expect that the phase analysis of two runs of the same application should be much more similar than that of two different applications.

In this section we present data on the extent of system-induced variability in real, measured application behavior and in Section 3 we show how this variability is reflected in the corresponding phase sequences. Here, we emphasize a clear distinction between two cases: (i) where an application uses the same dataset (or sequence of datasets) and (ii) where an application uses two different datasets. In the latter case, even though the application is mostly executing the same code, different datasets can lead to drastically different metric behavior. This necessitates different phase representations that might lead to different dynamic optimization actions.

2.1 Variability Analysis Methodology

In our variability analysis experiments, we collect real power and timing behavior information with a non-intrusive runtime processor power measurement setup. In these experiments, we use a current probe clamp over the Pentium 4 processor power lines, which is connected to an Agilent 34401 digital multimeter. The multimeter sends the instantaneous current information to a logging machine over RS232. The logger machine then converts this to runtime power dissipation information for the currently running applications on the tested system.

Here, we present metric variability in terms of "actual measured power". Therefore, this reported variability is purely a characteristic property of applications running on real systems, regardless of sampling effects or any applied phase analysis technique. While we cannot present data due to space constraints, our other experiments show this metric variability similarly reflected in metrics other than power, such as IPC and miss rates.

2.2 Variability Across Different Runs

Applications exhibit two types of variability on a real-system, across multiple runs. First, they show slightly different instantaneous behaviors in their characteristic metrics, such as IPC, miss rates and power dissipation. Therefore, at any specific time instance, these values show some deviation in each run. Second, and following from this, the applications show different timing behavior. This results in deviations in both total runtime and in the duration of each phase.

To quantify these two forms of variability, we collect data related to characteristic metrics and timing behavior of several applications for five different runs on the same system. In order to minimize cold start effects, we run each application six times and discard the results of the initial run. In all the experiments, the benchmarks are run to completion with reference datasets. After data collection, we align the traces of five runs such that all have the same first transition from idle to active phase. The first form of variability is then observed with the variation in individual measured metrics at each time sample. To show the second form of variability—different timing behavior—we specify 3 execution checkpoints for each application. We measure how long each run required to reach these points, starting from the idle-to-active transition common reference.

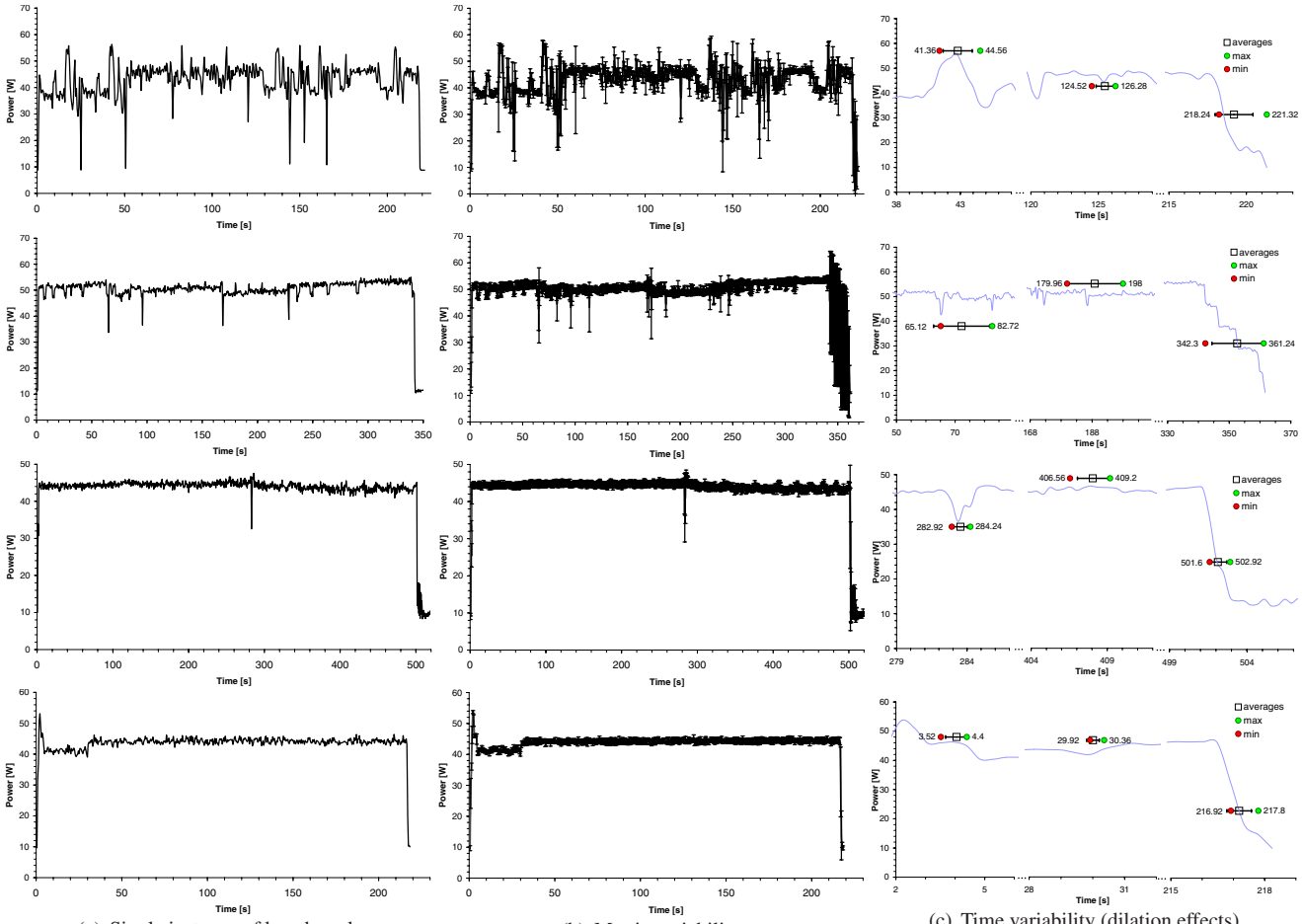
In Figure 1 we demonstrate the observed variability for Spec benchmarks `gcc` (1st row), `gzip` (2nd row), `vpr` (3rd row) and `equake` (4th row). For each benchmark, the leftmost graph shows the representative power timeline from a single run. The middle graphs show the measured metric variability. At each time sample, they plot the ∓ 1 *Standard deviation* error bars around the average measured power from all five runs. The rightmost graphs show each application's time variability at the three checkpoints. At each checkpoint, they show the average time, ∓ 1 *Standard deviation* error bars and maximum and minimum time elapsed until checkpoint over the five runs. We also show the average power behavior at these regions for reference.

All benchmarks exhibit some level of both metric and time variability. `Gcc` and `gzip`, two of the most highly-variant benchmarks of SPEC2000, have significantly higher metric variability, part due to the timing mismatches—jitter—at the power jumps. All experimented benchmarks exhibit time variability on the order of a few seconds, with the exception of `gzip`, which shows variations on the order of ten seconds. This variability is a fundamental aspect of real-system behavior, and is neither a side-effect of our phase analysis methodology, nor can be diminished with finer data sampling. In general, all applications result in visually similar power and performance behavior across different runs. However, some variability always exists in both characteristic metrics and run-time.

2.3 Variability Across Different Datasets

An orthogonal source of variability in application behavior results from the input datasets. In some cases, the behavior of an application is completely different under different datasets. In Figure 2 we show the `gcc` benchmark with very distinct behavior for different datasets. In `gcc`, both metric behavior and execution time are highly dependent on the compiled code.

In general, although the application in part executes the same functions on different datasets, this should not necessarily imply the same phase sequence. In Figure 2, the application exhibits different metric and timing behavior for different datasets. Consequently, the two separate dataset runs have distinct phase distributions. In such cases, it is a desirable feature—rather than a deficiency—for different inputs to result in phase behavior variability. These different phases can then be used to choose different dynamic optimizations based on their power and performance characteristics. For this reason, the rest of this work focuses on optimizing phase detection and repeatability to handle variability across different runs with the same datasets. Any mention of variability refers specifically to this form in the remainder of this paper.



(a) Single instance of benchmark (b) Metric variability (c) Time variability (dilation effects)

Figure 1. Measured time and metric variability in `gcc`, `gzip`, `vpr` and `equake`.

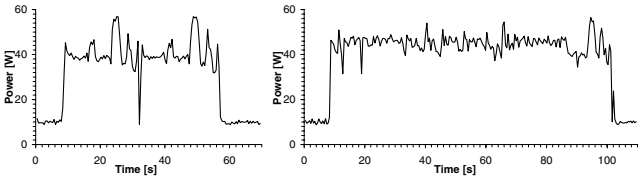


Figure 2. Measured power behavior of Spec `gcc` with different datasets, 166 (left) and 200 (right).

3 Impact of Real-System Variability on Phase Behavior

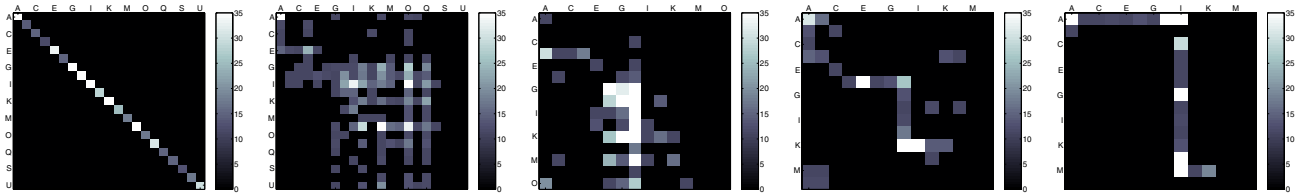
3.1 Phase Analysis Methodology

All phase analysis stems from tracking the similarities and trends in a time-varying set of metrics. In this work, we use the hardware PMCs on a 1.4 GHz Pentium 4 processor [30] to collect sets of data points representing time-varying power behavior of applications running on the same system at runtime. Our methodology is based on previous work [17], which uses PMCs to estimate the power behavior for 22 different on-chip hardware units. These 22-dimensional estimates, referred to as "power vectors", then sum up to an estimate of instantaneous total processor power consumption.

In this work, we start from these power vectors as detailed fingerprints of current microarchitectural activity. Although these 22 dimensions give a comprehensive view of processor activity, their high detail actually impedes discerning phase patterns. Therefore, we use Factor Analysis [11] to unify highly-correlated dimensions, while preserving the same predictive capacity. These reduced vectors have 12 dimensions.

The experimental implementation consists of a Linux Loadable Kernel Module (LKM) on a 2.4.7-10 kernel and a user application that samples counters every 100ms. This user application then transfers the PMC samples to a different platform over ethernet, for runtime phase analysis with minimal intrusion.

Phase analysis is inherently about gauging similarity and dissimilarity of sampled data over time. To gauge the similarity of two vector datapoints gathered by runtime PMC sampling, we use the composite similarity metric given in Equation 1.



(a) No variability (b) Gcc (c) Gzip (d) Vpr (e) Equake

Figure 3. Joint histograms of phase distributions for two separate runs of 4 benchmarks. (a) shows an example histogram in the case of no variability (i.e. repeatable simulations). (b-e) show the actual variability in phase behavior observed in real system runs. The letters at the top and left-hand side of the matrix plots are the phase labels.

$$S(i,j) = \min\left(\frac{AM(i,j)}{\max_{i,j}(AM(i,j))} + \frac{NM(i,j)}{\max_{i,j}(NM(i,j))}, 1\right) \quad (1)$$

In this equation, $S(i,j)$ varies from 0 to 1 and quantifies the similarity (or lack thereof) between two vectors i and j. A higher similarity between two vectors translates to a lower value in $S(i,j)$. $AM(i,j)$ represents the absolute L1 (manhattan) distance between the i^{th} and j^{th} vectors and $NM(i,j)$ represents the distance between the same vectors normalized, such that L1-norm of each vector is 1.

Our starting point in this work is a value-based phase clustering method. In this, we apply a set of thresholds to this similarity metric to cluster sampled data into phases. Applying a phase assignment technique similar to [17], we label encountered phases alphanumerically, starting from 'A' in each case. We call this phase representation *Value-Based Phases* (*VBP*s), where different observed phases are given different labels (phase IDs).

3.2 Value-Based Phase Behavior under Variability

Although the qualitative visual behavior of a benchmark is often preserved across multiple real-system runs, differences in phase assignments occur due to inter-run variability. Even small variations can lead to the spurious interpretation of a phase change, thus changing the phase assignments and sequence information that follow. In addition, the durations of an application's observable phases are not identical, which also impedes exact runtime-based phase tracking techniques.

Figure 3 gives examples of how variability affects phases. Here, we use *joint histograms* to illustrate these effects. We again use the same set of applications, and we use the value-based method to split them into *VBP*s. We then time-align them with respect to first phase transition. The joint histogram h of two phase sequences is a matrix, where entry $h(X,Y)$ shows how many times run1 was assigned to phase X when run2 was assigned to phase Y for the same data sample. The plots show the intensity of this matrix, where brighter regions correspond to higher number of matches and darker regions show poor matches. The x and y axes on the plots show the phase labels of the two runs. Figure 3.a shows the ideal matching in the case of perfect repeatability i.e., a simulation environment. In this case h is only a diagonal matrix, where the diagonal values differ depending on how often each phase is encountered during application runtime. In this case, if run1 is in phase 'C' at time t, then run2 is also in phase 'C' at t.

In Figure 3.b-3.e, we show the joint histograms resulting from real-system runs. In these cases, the phase assignments are far from ideal. Especially in `gcc` and `gzip`, phase assignments show a large spread, indicating significant mismatches. More well-behaved benchmarks `vpr` and `equake` show some level of regularity as they have few phases governing most of their phase behavior. Nonetheless, in both cases, several phases in one run collapse into a single phase in another, leaving little consistent phase behavior.

In summary, the observable cross-run variability seen in application power and time behavior also exists in the value-based phase characterizations of applications. This variability causes different runs of the same applications to be characterized by different phase sequences; this conceals the actual recurrent phase behavior.

3.3 Taxonomy of Phase Transformations

Figure 3 highlights the fact that direct, brute force comparisons of phase traces are ineffective in conveying repetitive behavior similarity. Before discussing our proposed methods, we first present a taxonomy of the effects of variability on phases. In Figure 4 we illustrate these effects and resulting phase transformations. The figure shows their cumulative effect on an ideal hypothetical phase distribution, shown as the phase sequence "A,B,C,B" where the length of each labeled block indicates the duration of the corresponding phase. The first effect—*time shifts* in phase sequences—will always occur, as the processor power trace can be considered as a stream of data with no specific beginning and end. The startpoint merely depends on where we start logging the sampled power information. The second effect, *time dilations*, inevitably results from indeterministic system effects. The length of a specific task depends on the state of the machine, the available locality, number of page faults and load of the system. *Glitches* occur when brief snippets of isolated behavior occur in some, but not all, runs. Finally, *mutations* are cases where a different phase name is seen in a run; this can be either due simply to labeling issues or it can be due to variable behavior in the application on different runs.

In the following sections, we tackle each alteration presented in this taxonomy and propose a series of techniques for recovering the phase behavior to the point where repeated runs of an application are recognized as similar.

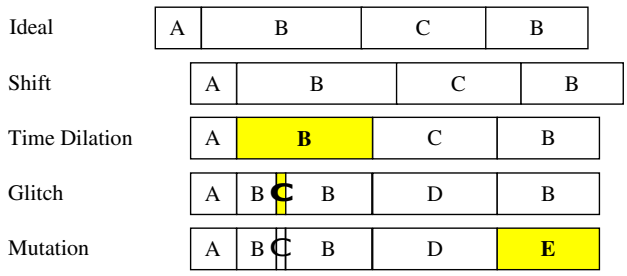


Figure 4. Effects of real system behavior variability on application phase distribution.

4 Using Phase Transitions as Application Signatures

In this Section, we propose a representation for application phase behavior that is an alternative to the prior value-based (*VBP*) approach. The goal of this representation is to be more resilient to real-system variations. We suggest tracking phase transitions, instead of tracking phases themselves, and show that transitions are more effective in detecting recurrent workload behavior. We identify phase transitions at runtime by comparing the current and the previous sample vector, and by evaluating their similarity based on Equation 1. This transition-based representation of phase behavior, in comparison to the original *VBP* representation, is much more successful in identifying a program from its phase signature and in rejecting other application signatures based on the tested features.

One way to evaluate our claim—that tracking phase transitions instead of phases is more successful in detecting recurrent behavior—is by computing correlations. If two phase traces vary together, they have a high correlation coefficient. Therefore one would expect high correlations between two runs of the same application, and much lower correlations among different applications.

To perform this comparison, we enumerate *VBP* sequences with positive integers, where phase numbers are assigned to encountered different phases in increasing order. This corresponds to the original value-based representation. For the same stream, we can also represent the transition information as a binary stream, assigning 1 to phase transitions and 0 to stable regions. This is our initial proposed transition-based phase (*TBP*) representation. We call these binary sequences *Initial Transitions* (TBP_{init}).

In Figure 5, we present the resulting correlation coefficients for two different cases. In both plots, the lighter lines plot the correlation coefficients for the original *VBP* traces. The darker lines show the results for the transition (TBP_{init}) traces. Figure 5.a shows the "matching" case for two separate runs of `gcc`. Here, since we are correlating phase sequences for two runs of the same program, a good phase assignment will show a high correlation spike when the two runs are properly time-aligned. Figure 5.b shows the "mismatch" case with `gcc` and `equake`. Here, we are correlating two unrelated phase sequences, so we do not expect a high spike.

In the correlation plots, we show the results for a range of time shifts to consider the probable lag between two runs. For instance, if two traces are identical, one would expect a peak (1) in $sample\ shift = 0$. If there is only a lag of x samples between traces, the peak will move to $+x$ or $-x$.

Figure 5 reveals that correlating value-based phase sequences does not produce good discrimination among benchmark signatures. In comparison, transitions provide much more useful results. Notably, we can distinctly see a peak in the TBP_{init} `gcc` vs. `gcc` case with a time-shift of 6 samples, while there is no observable peak from *VBP* correlations. Furthermore, correlating the transition traces of `gcc` and `equake` gives very low correlations as expected. The *VBP* correlations are also lower than their `gcc`-`gcc` counterpart, but transitions perform observably better, with roughly 0 correlation.

This distinguishable peak in the correlation trace for the transition-based TBP_{init} representation proves to be very useful in identifying benchmarks from their signatures. Starting with the next section, we look into these initial transitions in more detail, demonstrating how we can further improve and use this information to match application signatures under real workload variability.

5 Working with Transitions

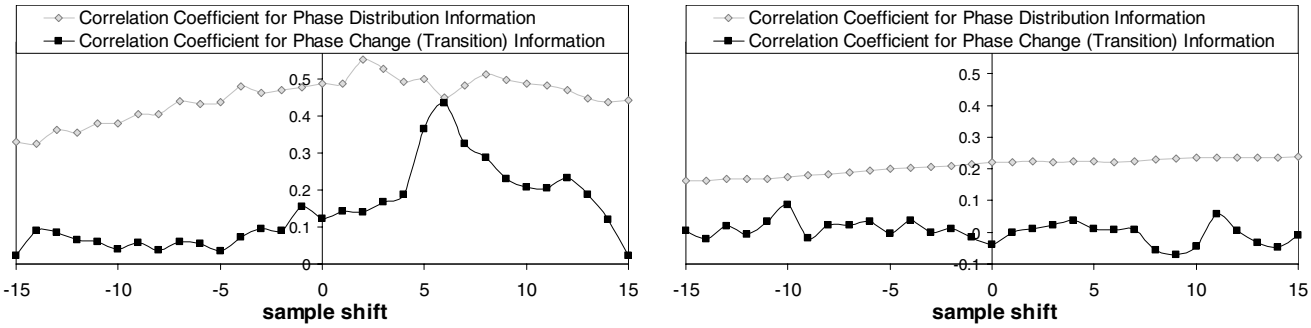
5.1 Removing Sampling Effects on Transitions with Glitch and Gradient Filtering

Our starting point for defining phase transitions was to say that they are sample points where the next interval's phase is different from the current phase. These transitions can be identified on-the-fly by evaluating the similarity metric in Equation 1 for the current and previous power vector and comparing against a similarity threshold. While Figure 5 illustrates that this TBP_{init} approach is already useful for phase detection, we improve on it here. In particular, we note that sampling and stability effects impede the effectiveness of transitions for representing phase behavior; we address that here.

We characterize these effects as *glitches* and *gradients* (Figure 6). Following the stability definitions of Dhodapkar and Smith [9], we define a *glitch* as one or more consecutive unstable sampling intervals, where the *before* and *after* of the glitch belong to the same stable phase. Because glitches are short and unstable, their single sample phase information is not likely to be suitable to exploit with dynamic management techniques.

A *gradient* is one or more consecutive unstable samples, where the *before* and *after* of the gradient belong to different stable phases. These regions correspond to an actual phase transition. However some phase transitions do not happen instantaneously in a single sampling interval, but instead can actually have multiple samples along the transition gradients.

In the context of our work, glitches are false transitions and gradients are duplicated transitions. To remove these spurious effects, we propose a more intelligent transition analysis that works to filter the transitions deemed to be glitches and gradients. In *Glitch/Gradient Filtering* extraneous transitions corresponding to glitches are discarded. Single or multi-cycle gradients en route to a new phase are



(a) Correlation of two gcc runs.

(b) Correlation of gcc and equake.

Figure 5. Correlation coefficients for a range of shifts between two *different* gcc runs (a) and separate gcc and equake runs (b) (y axis shows the computed correlation coefficient values).

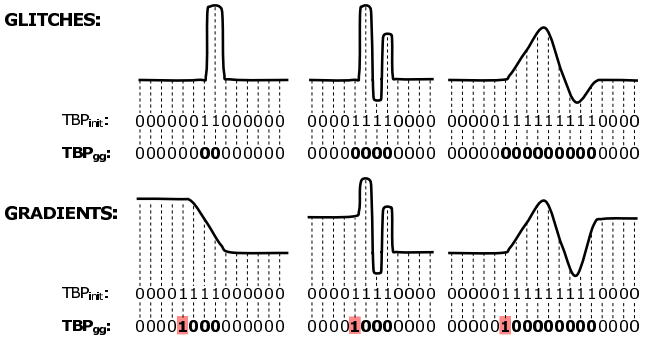


Figure 6. Initial transitions, TBP_{init}, with different types of glitches and gradients, and refined transitions, TBP_{gg}, after glitch/gradient filtering.

converted into a single stable phase change.

In Figure 6, we show the generic scenarios for the glitches and gradients. The upper rows depict the initial TBP_{init} traces. (Recall from Section 4 that '1' denotes a transition and '0' denotes stability.) The lower rows denote the refined transition traces after we apply our glitch/gradient filtering. We refer to these transitions with glitch/gradient removal as *refined transitions* or TBP_{gg}.

Our filter implementation identifies each initial transition by monitoring the phase stream, and forms the initial binary representation TBP_{init}. From the TBP_{init} stream, we construct TBP_{gg} in the following manner. Each burst of transitions is replaced by either no transitions—if they are glitches—or a single transition—if they form a gradient. We do not allow multiple consecutive transitions in the refined TBP_{gg} signature and all gradients have a prior transition adjacent to them.

In Figure 7, we show the application of glitch/gradient filtering to gcc benchmark. In the figure, we show the refined transitions, as well as the regions identified as glitches and gradients, for a zoomed-in execution region. For gcc, the initial 212 transitions reduce to 82 once glitch/gradient filtering is applied.

5.2 Discerning Phase Behavior with Time Shifts

Initially we have quantitatively shown the quality of matching with transitions using computed correlation co-

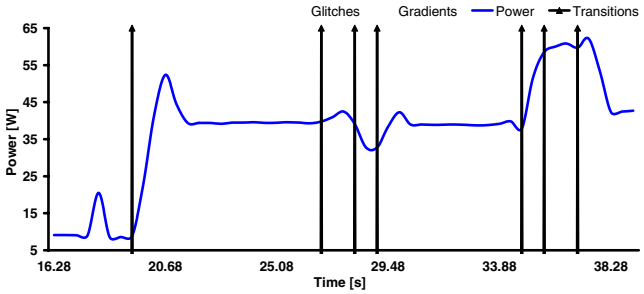


Figure 7. Transitions, glitches and gradients for gcc after glitch/gradient filtering.

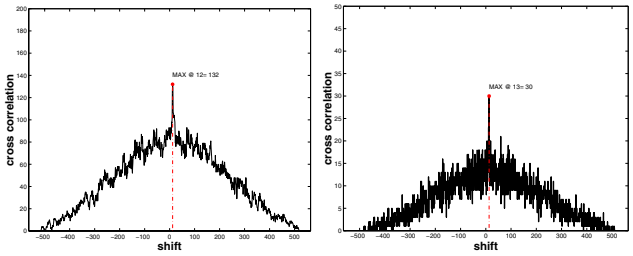
efficients for a range of shifts. However, this method is computationally expensive and not suitable for runtime applicability. As the generated transition features now contain simple binary information, a simpler metric to use is cross-correlation. Correlators can be easily implemented in hardware and can be applied continuously to the incoming data stream online.

In Figures 8 and 9 we again demonstrate the "matching" and "mismatch" cases. In the first case, we show how well a new gcc run can be matched to a previous gcc signature. In the second case, we run equake and examine the severity of a false alarm. We show the results for refined (TBP_{gg}) and initial (TBP_{init}) transitions in both cases.

For the two gcc runs, refined transitions show a significant peak, proving a good match between the two signatures for a shift of 13 samples. For gcc and equake, the cross correlation of transitions produces no significant peak, which suggests the signatures do not match. Thus, we can see the spike behavior in case of signature match is retained with refined transitions and with application of cross correlations.

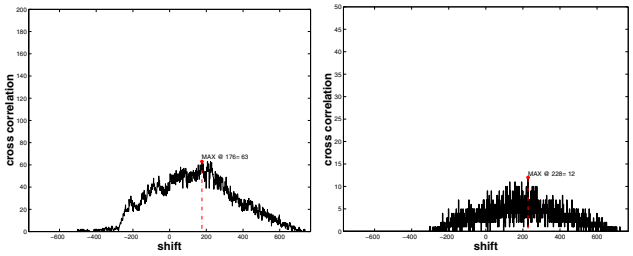
5.3 Handling Time Dilations with Near-Neighbor Blurring

In addition to glitches and gradients, time dilation between runs is a common problem. Recognizing the similarity of an original phase trace with a time-dilated one is a problem with similarities to many other research domains. Examples include matching a warped image in image recognition or pitch tracking in humming recognition [31]. These high-level methods can afford high complexity



(a) Cross-corr. of initial transitions. (b) Cross-corr. of refined trans-ns.

Figure 8. Matching of transition signatures for two `gcc` runs.



(a) Cross-corr. of initial transitions. (b) Cross-corr. of refined trans-ns.

Figure 9. Matching of `equake` transition signatures to `gcc`.

and they can store vast libraries of training data. In contrast, our goal is to implement an approach with simple correlators and table lookup on a small set of recent signatures.

In Table 1, we demonstrate the potential problems that time dilations pose on the transition guided phase detection scheme. In Table 1.a, we show the high matching of processed transition information (lower trace) to a previous baseline signature (upper trace) in the absence of time dilations. In Table 1.b the lower transition trace is dilated, which shows the negative effect of time dilations on detecting recurrent behavior.

a. No Dilation	b. With Dilation
0 0 1 0 0 1 0 1 0 0 0 0 1 0	0 0 1 0 0 1 0 1 0 0 0 0 1 0
0 0 1 0 0 1 0 0 0 0 0 0 1 0	0 1 0 1 0 0 0 0 0 0 0 0 1 0
$\surd$ (match)	$\times$ (mismatch)

Table 1. Effect of time dilations in detecting recurrent behavior.

This matching problem results from considering transition information to be sharply associated with a particular deterministic sample point, while the actual transition times in each run are instead probabilistic with a modest distribution around an average. (See Figure 1.c for examples.) To remedy this problem, we propose a *near-neighbor blurring* solution, which is fundamentally similar to blurring image edges for image matching. With near-neighbor blurring, we consider transitions as distributions along the time axis centered at their encountered locations. With this probabilistic approach, subtle time dilations are not penalized altogether, but instead are scaled according to their proximity to the exact location.

Tolerance: We use this metric to define the "spread" of the distribution we assume around an encountered transition time point. We define this in terms of samples. For example, a tolerance of x samples means that a transition at time sample t is considered to have a distribution in the sample range of $[t-x, t+x]$.

In our implementation, we choose a relatively primitive model, where we scale the near neighbors of transitions linearly from 1 to 0, based on the chosen sample tolerance. Further research could investigate other suitable distributions to characterize phase transitions. To apply near-neighbor blurring, the baseline refined signature (TBP_{gg}) is altered from its raw form to generate the distributions. The second live TBP_{gg} stream, on the other hand, is not altered to avoid the runtime cost. We show in Table 2 how the example of Table 1 is altered for a tolerance of 4 samples. With near-neighbor blurring, previous mismatch due to time dilations is now detected as a strong match.

Applying near-neighbor blurring to TBP_{gg} results in similar cross correlations as in Figures 8 and 9. For the remainder of this paper, we refer to TBP_{gg} augmented with near-neighbor blurring as TBP_{ggN}. In Section 5.4 which follows, we quantify these results for our overall algorithm, using a quality metric we refer to as the *matching score*.

5.4 Quantifying Signature Matching with Matching Score

Matching Score: In order to quantify the success of a matching, we define the *matching score* metric, m, which provides a measure for the strength of matching between two signatures. Our goodness measure is the strength of the cross-correlation peak at the best alignment. Therefore, we define m as the ratio of best match value to the average of its closest 10 best matchings. As this value will always be greater than 1, we subtract 1 from the final value to remove this offset.

For our previous experiments with two `gcc` runs—the *matching* case,—the matching scores for initial transitions TBP_{init}, refined transitions TBP_{gg} and near-neighbors TBP_{ggN} are 0.22, 0.55 and 0.32. Corresponding values for the `gcc` vs. `equake` comparison—the *mismatch* case—are 0.054, 0.16 and 0.036. Therefore, TBP_{gg} performs best for signature matching as it produces the highest matching score between the two runs of `gcc`. On the other hand, TBP_{ggN} performs significantly superior for signature rejection, as it has a much lower matching score for the signatures of `gcc` and `equake`.

5.5 Summary of Methods

Before presenting the general quantitative results of our transition-guided recurrent phase detection method, here we provide a brief summary of our applied techniques, as depicted in Figure 10. First, we sample PMCs during application runtime and represent benchmark execution as a stream of vectors (Section 3.1). Then, evaluating the similarity between each current and previous vector sample, we identify initial transitions (Section 4). This process converts the application execution into the binary stream TBP_{init}. Next, we apply glitch/gradient filtering to TBP_{init} streams and convert them into refined transitions, TBP_{gg} (Section 5.1). In

baseline (refined):	0	0	1	0	0	1	0	1	0	0	0	0	1	0
baseline (near-neighbor):	0.6	**0.8**	1	**0.8**	0.8	1	0.8	1	0.8	0.6	0.6	0.8	**1**	0.8
new run with time dilation:	0	**1**	0	**1**	0	0	0	0	0	0	0	0	**1**	0
							$\surd$ (match)							

Table 2. Detection with near-neighbor blurring under time dilation.

addition, for the first run, we apply near-neighbor blurring to TBP_{gg} and generate the baseline signature TBP_{ggN} (Section 5.3). After this point, any newly observed TBP_{gg} trace is cross-correlated with this baseline TBP_{ggN} to detect a signature match (Section 5.2). A match is determined based on the strength of an observed peak in the cross-correlation sequence, which we quantify with our matching score metric (Section 5.4).

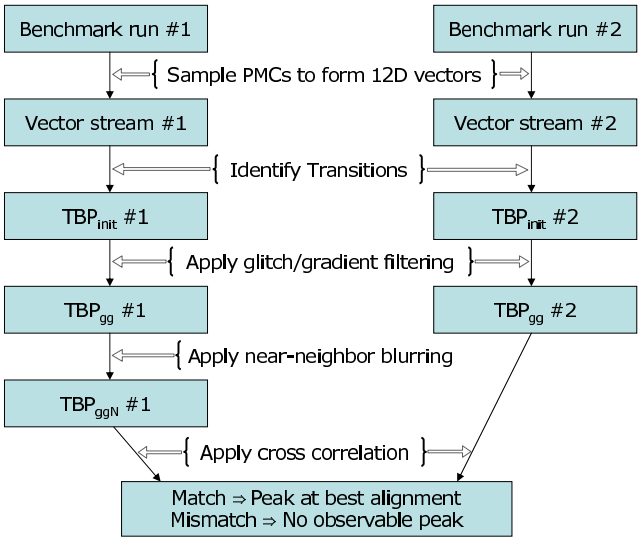


Figure 10. Flow of our methods.

6 General Recurrent Phase Matching Results and Evaluation of Methods

In Section 5, we have described our complete transition-guided phase detection and evaluation methodology, which tackles all described repeatability problems. In this section we present the general results for our technique.

6.1 Phase Detection Results

We present our phase detection results for a spectrum of benchmarks that include SPEC and other mainstream applications. We choose a subset of SPEC benchmarks that exhibit distinct phases in terms of power and performance metric behavior. Most of these benchmarks have high metric variability, with varying transitions across different runs. Additional non-SPEC applications share interesting phase characteristics. `Convert` is a general file conversion program that converts a large postscript file into pdf. `Convert` shows significant phases depending on the contents of the input file. We use the `lame` MP3 encoder to encode a wave file under varying quality settings. The power levels increase with finer recurrent phases at higher quality settings.

In our experiments, we run each application twice on our measurement setup. During the first run, we collect the phase transition information and apply glitch/gradient removal as they are identified. In our analysis, we consider a range of near-neighbor tolerances as well as the refined transition signatures—i.e. the outputs of glitch/gradient filtering, without near-neighbor blurring. In the second run, we only generate refined transitions without any blurring.

Table 3 presents the matching scores for the experimented application pairs. The diagonal entries show the matching scores for the two runs of the same application—the *matching* cases. The non-diagonal entries show the matching scores between two different applications—the *mismatch* cases. The baseline signatures correspond to the columns of Table 3. The transitions for the second runs are represented in the rows of the table. Therefore, the matching scores read along a column show how well a baseline signature can characterize a repeatable application phase behavior. In Table 3, we present the matching scores corresponding to the tolerances that maximize the matching score ratio to the highest mismatch score.

As an example, for `gzip`, the baseline signature has near-neighbor blurring with a tolerance of 1 samples as indicated by the value in parentheses. Reading the `gzip` column shows, a second run of `gzip` produces a matching score of 1.0784 to the baseline `gzip` signature. However, the same baseline signature produces much lower matching scores for the runs of other benchmarks, with an average of 0.13. Among these other benchmarks, `equake` has the highest matching score to `gzip` baseline signature with 0.2587. This is significantly lower than `gzip`'s matching score of 1.0784. Thus, our transition-based scheme successfully detects the 2nd run of `gzip` from its transition signature, while strongly rejecting signatures of the other benchmarks. In general, for all the benchmarks, we see the same trends. In all cases, the highest matching scores correspond to the second runs of the same application (diagonal entries), while the matching scores for different applications (non-diagonal entries) are significantly lower.

Most benchmarks have their best matching under a few levels of tolerance (1-3 samples) due to their small dilation magnitudes. The only exception is `convert` with an optimal tolerance of 7. As `convert` has only 17 transitions in its signature, each extra hit in the the spread has greater relative impact, thus favoring higher tolerances. Note that, the 0 tolerance case is equivalent to the TBP_{gg} signatures, without any blurring. Only for `mcf`, is the best matching condition achieved by TBP_{gg}.

In general, the outcomes of our detection method are very useful. We can detect specific recurrent phase sequences under different kinds of variability, with a moderately simple technique that can be implemented at runtime with negligible overhead. In most cases, considering tran-

	bzip2 (1)	equake (2)	gap (1)	gcc (3)	gzip (1)	mcf (0)	vortex (1)	convert (7)	lame (2)
bzip2	**0.4419**	0.0506	0.0744	0.0517	0.1494	0.1765	0.0791	0.0901	0.1481
equake	0.145	**0.391**	0.2766	0.0594	0.2587	0.25	0.0929	0.0405	0.0757
gap	0.2	0.2195	**0.7857**	0.072	0.0959	0.3333	0.0417	0.0519	0.1152
gcc	0.0526	0.0364	0.0547	**0.188**	0.0299	0.0526	0.1565	0.0373	0.117
gzip	0.0995	0.0959	0.1875	0.0549	**1.0784**	0.1594	0.1014	0.0338	0.0687
mcf	0.1834	0.1828	0.2319	0.0434	0.1558	**6.1429**	0.1706	0.0783	0.0802
vortex	0.2333	0.1047	0.1165	0.0132	0.1144	0.0784	**1.9297**	0.0303	0.048
convert	0.2057	0.1719	0.2632	0.06	0.1404	0.25	0.0884	**0.2165**	0.1284
lame	0.1211	0.1111	0.117	0.0382	0.1261	0.2	0.0645	0.0194	**0.214**

Table 3. Matching scores for different applications. Benchmarks in each column represent the base signatures that we apply near-neighbor blurring. The matching scores represent how well the refined phase transition signatures of the row benchmarks match to these base signatures. The values in parentheses next to benchmarks show the optimum tolerance.

sitions as distributions via near-neighbor blurring improves our results further, with the choice of small tolerance levels.

6.2 Receiver Operating Characteristics

As with any detection scheme, our matching scores are also prone to *misses* and *false alarms* for a particular *detection threshold*. That is, for all applications, a matching score above this single detection threshold is considered as a detected 'hit'. For instance, for the runtime detection scheme of Table 3, if we use a threshold of 0.188, we would be able to identify all the hits. However, out of the 72 possible mismatches we would also have detected 11 of them as hits. Thus, for this scenario, we would have a hit detection probability of 1. However, this would also incur a *false alarm* probability of $11/72 \approx 15.3\%$. If we increase the detection threshold, the probability of false alarms reduce, while this, in turn will incur some hits being *missed*. It is common practice in pattern classification to demonstrate this effect in terms of *Receiver Operator Characteristic* (ROC) curves. The detection function is drawn for its hit probability with respect to the probability of false alarms [11]. We show the ROC curves for our detection technique in Figure 11. To present the probabilities, the axes are shown from 0 to 1. However, for absolute measures, 1 on the *hit* axis represents 9 detected hits for the 9 benchmarks; and 1 in the *false alarm* axis represents 72 falsely detected hits for the 72 possible different benchmark combinations. The intermediate values are linearly scaled for both axes. Each ROC curve in the figure corresponds to a TBP_{ggN} with a specific tolerance; and for each curve we first compute the matching score matrix , similar to Table 3, across the (9x9) benchmarks for the current tolerance value, and then compute the hit and false alarm probabilities for several detection thresholds with step sizes of 0.05 for the whole matching score range 0-6.15.

In the ROC curves, we see our detection scheme achieves high hit probabilities with small rate of false alarms. Among the applied tolerance levels, TBP_{ggN} with 1 sample tolerance perform best, which is followed by tolerances of 2 and 3. The zero tolerance case, which corresponds to TBP_{gg}, with no distribution, performs distinctly worse for signal rejection. This proves the effectiveness of our near-neighbor blurring technique. Our best detection method

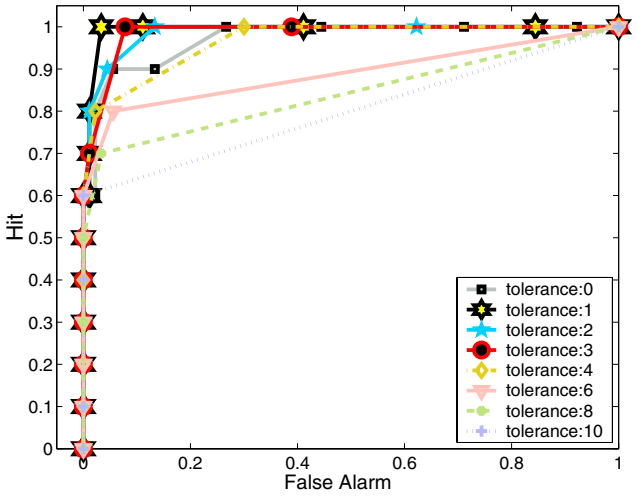


Figure 11. Receiver Operating Characteristic (ROC) curves for TBP_{ggN} with 0-10 range of tolerances.

achieves 100% hit detection with less than 5% false alarms.

6.3 Improvement in Signature Detection with Transition-Guided Approach

In Figure 12, we provide a final comparison of detection success between the original value-based phase representation (VBPs), refined transitions (TBP_{gg}); and final near-neighbor blurred transitions (TBP_{ggN}). For this comparison, we show the ratio of the matching score between two runs of the same application (matching case) to the highest matching score among all the different applications (worst mismatch case) for the same application—i.e. `vortex` for `bzip2`. Consequently, this quantifies how well each representation detects a matching signature, while rejecting other unmatching signatures. We show the individual results and average for the experimented benchmarks. The "break-even" line at $ratio = 1$ shows, below this point, a technique finds another application signature as a "better match" for the current benchmark, while ratios significantly higher than 1 represent an accurate detection of a signature.

In all cases, transition based methods perform much better than VBPs. In all cases except `mcf`, TBP_{ggN} shows significant improvement over TBP_{gg}. For `mcf`, both transition techniques perform equally well, as the best tolerance for `mcf` is 0. On average, our transition based, near-neighbor blurring technique provides a 6-fold improvement in recurrent behavior detection under variability, over the original value-based phases.

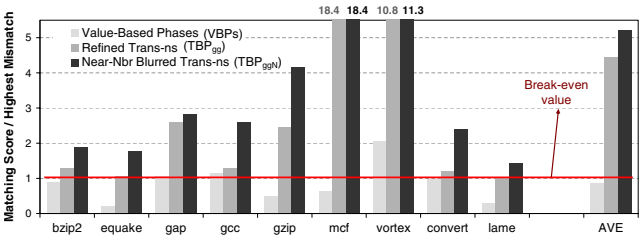


Figure 12. Improvement in phase detection efficiency with transition-based approach.

7 Related Work

Prior phase detection work operates at various domains and granularities using a variety of characteristic metrics to track phases. Dhodapkar and Smith [10], Sherwood et al. [27, 28], Lau et al. [21], Iyer and Marculescu [19], and Huang et al. [15] track the executed code characteristics such as basic blocks and subroutine IDs to detect phases. All these works are based on cycle-level simulations and, although useful for guiding representative simulation and architectural studies, they do not reflect the available real-system variability.

Some recent research also looks at executed code characteristics under real-system experiments. Patil et al. [24] and Lau et al. [20] use dynamic instrumentation to identify basic block based phases. Hu et al. [14] discuss compile time instrumentation to find basic block phases at runtime for power studies. Annavaram et al. [2] apply program counter sampling to find similar execution paths and investigate performance behavior similarity in these regions. These approaches also account for real-system variability. However, they do not consider detection of recurrent phase sequence signatures.

Another line of research explores performance behavior for phase tracking, using metrics such as IPC and memory references. Cook et al. [7] identify execution phases based on deterministic simulations. Todi [32] and Weissel and Bellosa [33] use runtime performance counter information on different platforms for workload characterization and reactive dynamic optimizations. Duesterwald et al. [12] also use performance counters to predict metrics such as IPC and L1 misses. Their work uses previous short-term sample history to predict behavior in the next sampling period. These run-time techniques also analyze application behavior under variability, but they do not aim to detect large-scale recurrent phase sequences. Shen et al. [25] also look at detecting recurrent phases, by observing reuse distance patterns. They use detailed program profiling and instrumentation to detect phases, while our work tries to identify phase transitions from runtime power vectors.

Finally, there is prior work both by ourselves and others that directly looks at program power behavior. In [17], we use runtime power measurement and estimation to identify phases, and Chang et al. [6] use a power profiling method triggered by consumed energy quanta to attribute software energy to processes. In comparison, this work uses detailed performance counter information to identify phase transitions, which are in turn used for detecting large-scale, recurrent behavior.

8 Discussion and Future Research

The motivation of our presented research is to demonstrate the problems of recurrent behavior detection particular to experiments on real-systems and to provide the detailed description of our *transition-based* solution. Nonetheless, there remain additional aspects of this research, which constitute our ongoing and future research directions.

Applications of Detected Recurrent Signatures: Signature detection can be applied to various workload dependent dynamic management strategies. Specifically for the large timescale program phases that we analyze, observed signatures can be used for thermally aware scheduling [4], workload balancing to meet power budgets and activity migration [13]. For these applications, observed signatures have to be coupled with their corresponding power and temperature properties. There have been several prior research efforts to estimate workload power and thermal behavior at runtime with performance monitoring [5, 18, 22]. Therefore, our work can vastly benefit from these methods to project the long-term power and thermal behavior for an observed signature. There are several additional interesting challenges to employing our method in the context of these applications; such as choice of signature sizes, time scales of valid power/temperature projections and baseline signature logging and replacement policies.

Comparison to Program Counter Based Techniques: Another interesting point of consideration is how this PMC sampling methodology compares to program counter (PC) based phase characterizations such as basic block vectors (BBVs) [27]. Although a direct comparison between full-blown BBVs and runtime PMC samples is not meaningful due to excessive sampling overhead, coarser PC sampling can be considered as a representation of execution paths [2]. Lau et al. [20] show that a strong correlation between sampled PC signatures and application behavior can be established by mapping PC samples to control flow blocks, such as loops and procedures. These PC based techniques are attractive for simulation and characterization oriented research. However, there is still research needed for an efficient runtime representation for a direct comparison of techniques on a real-system.

9 Conclusion

This paper presents a novel approach to phase behavior detection under real-system variability. Based on real-system measurements, we categorize the variability effects and provide methods to address these distortions of phase

behavior. We propose a *transition-based* phase representation and demonstrate its robustness against phase mutations and shifts with correlations. We develop *glitch/gradient filtering* to refine phase transitions from sampling effects and use *near-neighbor blurring* to handle observed moderate time dilations. By carefully discriminating these variability effects and application specific phase information, we are able to detect recurrent phase sequences prone to several real world transformations.

Overall, our results show that this fully-automatable flow of techniques can detect recurrent application phase signatures with good accuracy for SPEC and other benchmarks. Our best detection scheme, near-neighbor blurring with a tolerance of 1 sample, was able to detect all signatures with a false alarm probability less than 5%. In comparison to original value-based phase representation, transitions with near-neighbor blurring performed on average 6X better in detecting recurrent application signatures, while rejecting unmatching signatures.

This research has importance both in characterizing real-system variability effects and in addressing phase detection under this variability. While further work remains to be done, we believe the presented results represent a useful first step in program phase detection under system variability.

References

[1] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of 9th International Symposium on High Performance Computer Architecture (HPCA-9)*, Feb. 2003.

[2] M. Annavaram, R. Rakvic, M. Polito, J.-Y. Bouguet, R. Hankins, and B. Davies. The fuzzy correlation between code and performance predictability. In *Proceedings of the 37th annual International Symp. on Microarchitecture*, 2004.

[3] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. mei W.Hwu. Vacuum packing: extracting hardware-detected program phases for post-link optimization. In *Proceedings of the 35th International Symp. on Microarchitecture*, Nov. 2002.

[4] F. Bellosa, A. Weissel, M. Waitz, and S. Kellner. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'03), New Orleans*, Sept. 2003.

[5] W. Bircher, J. Law, M. Valluri, and L. K. John. Effective Use of Performance Monitoring Counters for Run-Time Prediction of Power. Technical Report TR-041104-01, University of Texas at Austin, Nov. 2004.

[6] F. Chang, K. Farkas, and P. Ranganathan. Energy driven statistical profiling: Detecting software hotspots. In *Proceedings of the Proceedings of the Workshop on Computer Systems*, 2002.

[7] J. Cook, R. L. Oliver, and E. E. Johnson. Examining performance differences in workload execution phases. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-4)*, 2001.

[8] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, pages 323–333, May 1968.

[9] A. Dhodapkar and J. Smith. Comparing program phase detection techniques. In 36th International Symp. on Microarchitecture, 2003.

[10] A. Dhodapkar and J. Smith. Managing multi-configurable hardware via dynamic working set analysis. In 29th Annual International Symposium on Computer Architecture, 2002.

[11] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification. Second Edition*. Wiley Interscience, New York, 2001.

[12] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *IEEE PACT*, pages 220–231, 2003.

[13] S. Heo, K. Barr, and K. Asanovic. Reducing power density through activity migration. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED), Seoul, Korea*, Aug. 2003.

[14] C. Hu, D. Jimenez, and U. Kremer. Toward an evaluation infrastructure for power and energy optimizations. In *Workshop on High-Performance, Power-Aware Computing*, 2005.

[15] M. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In *Proceedings of the International Symp. on Computer Architecture*, 2003.

[16] C. Hughes, J. Srinivasan, and S. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34)*, Dec. 2001.

[17] C. Isci and M. Martonosi. Identifying program power phase behavior using power vectors. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-6)*, 2003.

[18] C. Isci and M. Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *Proceedings of the 36th International Symp. on Microarchitecture*, Dec. 2003.

[19] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *Proceedings of Design Automation and Test in Europe, DATE*, Mar. 2001.

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

[21] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *11th International Symposium on High Performance Computer Architecture*, 2005.

[22] K. Lee and K. Skadron. Using performance counters for runtime temperature sensing in high-performance processors. In *Workshop on High-Performance, Power-Aware Computing*, 2005.

[23] T. Li and L. K. John. Run-time modeling and estimation of operating system power consumption. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2003.

[24] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *Proceedings of the 37th annual International Symp. on Microarchitecture*, 2004.

[25] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Oct. 2004.

[26] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[27] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior, 2002. In Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002. http://www.cs.ucsd.edu/users/calder/simpoint/.

[28] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA-30)*, June 2003.

[29] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[30] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, Jul/Aug 2002.

[31] D. Talkin. *A robust algorithm for pitch tracking (RAPT). Speech Coding and Synthesis*. Elsevier Science B. V., New York, 1995.

[32] R. Todi. Speclite: using representative samples to reduce spec cpu2000 workload. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-4)*, 2001.

[33] A. Weissel and F. Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2002), Grenoble, France,*, Aug. 2002.