Phase Behavior in Serial and Parallel Applications

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

Abstract

It is well known that most serial programs exhibit time varying behavior, for example, alternating between memory- and compute-bound phases. However, most research into program phase behavior has focused on the serial SPEC benchmark suite, with little investigations into large scale phase behavior in parallel applications.

In this study we compare and examine the time-varying behavior of the SPEC2006 (serial) and the PARSEC 2.1 (parallel) benchmarks suites, and investigate the program phase behavior found in parallel applications with different parallelization models. To this end, we extend a general purpose runtime phase desection library to handle parallel applications.

Our results reveal that serial applications have significantly more program phases ($2.4\times$) with larger variation in CPI ($1.5\times$) compared to parallel applications. While the number of phases are fewer in parallel applications, there still exists interesting phase behavior. In particular, we find that data-parallel applications have shorter phases with more threads. This makes phase-guided runtime optimizations (e.g., dynamic voltage frequency scaling) less attractive as the number of threads grows. Meaning it is much more difficult to exploit runtime optimizations in parallel applications.

1. Introduction

Most programs have time varying phase behavior [39, 40, 15, 13]. This insight has been exploited in various dynamic runtime optimizations, e.g., cache resizing [12, 41, 38, 28], dynamic voltage frequency scaling [21], scheduling [42] and phase-guided profiling [31, 37]. However, most of this research is based on the SPEC [18] benchmarks. In resent years, the focus in computer research has shifted to also include parallel applications. The motivation for this work is that we have seen a lot of interesting runtime optimizations for the SPEC benchmarks. We now want to see if those results are also representative for parallel applications. To do so, we examine the time varying behavior of parallel applications and compare it with that of traditional single-threaded applications.

We use the PARSEC 2.1 [8] benchmark suite to investigate phase behavior in parallel applications. PARSEC targets chip multiprocessors with shared memory, includes applications from emerging workloads and diverse application domains, and utilizes different parallelization models (e.g., data-parallel and pipeline-parallel) and implementations (i.e., pthreads and OpenMP). In contrast to SPEC [39, 40, 34, 2], only aggregated metrics (e.g., average cache miss ratio) have been used to characterize its behavior [8, 7, 5, 6], which can be misleading and hide the effects of program variation over time.

To compare and examine phase behavior in serial and parallel applications, we use the ScarPhase (Sample-based Classification and Analysis for Runtime Phases) library from previous work [36] to detect and classify program phases. It divides the execution into non-overlapping windows and assign a phase id to each window. However, ScarPhase only supports single-threaded applications. We therefore extend the library to detect phases in parallel applications. This provides us with low-overhead runtime desection of general-purpose program phases [41, 32, 36]. We then combine this phase information with runtime data (e.g., *cycles per instruction* (CPI)) collected during each window using hardware performance counters to characterize the program phase behavior in SPEC and PARSEC.

The main contributions of this paper are:

- A comparison of time varying behavior between serial (SPEC2006) and parallel (PARSEC 2.1) applications on real hardware. The results show that serial applications exhibit significantly more phases ($2.4\times$) and more variation ($1.5\times$) over time. This means that some earlier runtime-optimization proposals based on SPEC numbers can not easily be extended to parallel benchmarks.

- An extension of ScarPhase [36] to detect phases in parallel applications.

978-1-4673-4532-3/12/$31.00 ©2012 IEEE

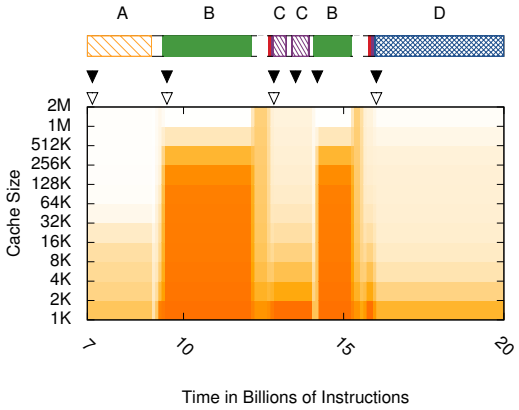


Figure 1. Miss-ratio (intensity) [37] as a function of time (x-axis) and cache allocation (y-axis) for gcc/166. The detected execution phases are shown above, with shorter phases shown in white for clarity. The black triangles shows when the phase detector notices a new phase and the runtime system applies the best optimization for the current phase.

- A study of program phase behavior in parallel applications and the effects of parallelization model (e.g., data-parallel and pipeline-parallel) on the phase behavior.
- A case study of how phase behavior changes when scaling the number of threads into the many-core (>16) region. The results show that the phases in data-parallel applications shrink (i.e., less work per thread) with an increase in number of threads. This makes phase-guided runtime optimizations (e.g., DVFS) less attractive as the number of threads grows, since the CPU frequency must be changed more frequently.

2. Phase-Guided Optimizations

Before characterizing the program behavior, we give a short background on how we detect phases at runtime, and an overview of how program phases has been used in the past to implement different phase-guided runtime optimizations.

2.1. Detecting Program Phases

We use the ScarPhase [36] library to detect and classify phases. ScarPhase is an execution history based, low overhead (2%), online phase desection library. It is based on the application's execution path, and detects hardware independent phases [40, 33]. Such phases can be readily missed by performance counter based phase desection.

To detect phases, ScarPhase monitors executed code, based on the observation that changes in executed code reflect changes in many different metrics [39, 40, 13, 41, 23]. To accomplish this, execution is divided into non-overlapping windows. During each window hardware performance counters are used to sample conditional branches using Intel Precise Event Based Sampling (PEBS) [29, 19]. The address of each branch is hashed into a vector of counters called a conditional branch vector (CBRV), similar to a basic block vector (BBV) [39] but with only conditional branches. Each entry in the vector shows how many times its corresponding conditional branches were sampled during the window. The vectors are then used to determine phases by clustering them together using an online clustering algorithm [14]. Windows with similar vectors are then grouped into the same cluster and considered to belong to the same phase.

2.2. Phase-Guided Runtime Optimizations

Phase-guided runtime optimizations exploit the heterogeneous nature of an application's phase behavior. They monitor executed phases and apply the best runtime optimization for each phase. To illustrate how this works, we have zoomed in on a short part of gcc/166's execution in Figure 1 [37]. The Figure shows the miss-ratio (intensity) as a function of time (x-axis) and cache allocation (y-axis). The detected execution phases are shown above, with shorter phases shown in white for clarity. The black triangles indicate when the phase detector notices a phase change and when the runtime system can apply an optimization (e.g., DVFS) for the new phase.

Phase-guided cache resizing/dynamic voltage frequency scaling [12, 41, 38, 28, 21, 30] is used to reduce the energy consumption without sacrificing performance. The optimal settings (i.e., cache size, voltage and frequency) is found for each phase. When the application changes phase, the appropriate settings are applied. For example, when gcc changes phase from A to B in Figure 1, the cache size should be increased, or the frequency lowered, since gcc entered a more memory bound phase with a larger working set. This can be seen in the figure by the increase in cache miss-ratio.

Phase-guided scheduling [42] is used to exploit heterogeneous multi-processors to improve throughput. When the application enters a memory bound phase, it is migrated to a slower core/chip, and when it returns to a compute intensive phase, it is migrated back to the faster core. For example, when gcc changes phase from B to D, it becomes more compute bound, and should therefore be migrated back to a faster core.

Phase-guided profiling [31, 37] is a method to reduce the overhead of profiling without sacrificing accuracy, by

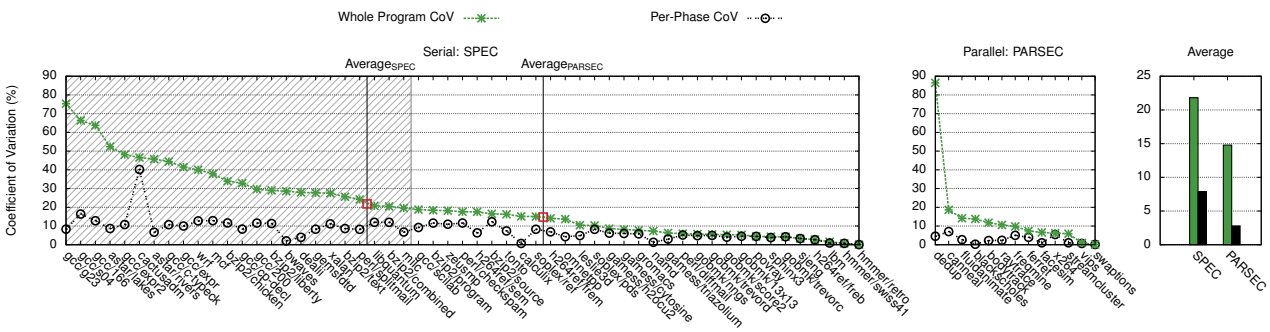


Figure 2. The CPI CoV (standard deviation divided by the mean). The figure shows how the CPI varies over the whole program execution (Whole Program CoV) and within a phase (Per-Phase CoV). The higher the CoV, the higher variation in CPI. The gray area highlights the difference between the benchmarks suites. All SPEC programs in the gray region have a higher CoV than all PARSEC programs except dedup. *This shows that with the exception of dedup (discussed in Section 4.1), the serial SPEC benchmarks shows significantly more program variation than the parallel PARSEC benchmarks (1.5× on average).*

taking advantage of the (nominally) uniform behavior of each program phase. This method only profiles a small part of each phase, and then use that profile for all other windows belonging to the same phase. The white triangles in Figure 1 shows where phase-guided profiling decides to profile the applications. For example, the first time gcc executes phase B, the profiler profiles one window from phase B. The next time phase B is executed, after phase C, the profiler already knows the behavior of phase B, and reuses the previous profile of phase B.

2.3. Prediction

The phase of each window is only known after the window has been executed. But, a runtime decision is needed before the window starts to execute. To circumvent this limitation, the phase of the next window can be predicted to make a runtime decision for the next window. Advanced history predictors have been proposed [15, 41, 27, 21], where the prediction is based on previously seen behavior. If the phase pattern has not been seen, it falls back on last value prediction. In this work, we use the run length encoded markov predictor described in [41, 27] with a 256 entries lookup table, a run length encoding of 2 and a confidence threshold of 1.

3. Methodology

We ran our analysis on all benchmarks in the SPEC2006 [18] and PARSEC 2.1 [8] benchmark suites. However, we also investigated other parallel benchmarks suites. We found that the NAS [4] benchmarks exhibited a very limited set of phases behaviors. Most of them had execution behaviors very similar to facesim from PARSEC. Due to space limitation we do not include those results in this paper. The SPEC and PARSEC experiments were run on an Intel Xeon E5620 (Nehalem), 4-core machine, with a window size of 100 million[1] instructions. All benchmarks were run from start to completion with reference and native input for SPEC and PARSEC respectively. We used Linux perf_events [1] to collect runtime data (e.g., CPI, L3 miss ratio, etc.) during each window.

4. Serial Phase Behavior

In this section we compare the serial phase behavior between SPEC and PARSEC[2]. To limit the amount of data we only examine the serial versions of PARSEC. Parallel versions of PARSEC are examined in the next sections.

4.1. Time Varying Behavior

To characterize an application's time varying behavior, we use the *Coefficient of Variation* (CoV) [40, 41, 23], that is the standard deviation divided by the mean, a common metric for evaluating the accuracy of phase classification. The higher the CoV, the more heterogeneous the behavior is. In this section we are interested in the overall performance behavior, and we therefore use the CPI CoV. We look at both the behavior for the whole program (i.e., the CoV is

[1] We chose a window size of 100M instructions because it has been used extensively in other works to evaluate phase behavior in the SPEC benchmarks [39, 40, 11, 3, 16, 26, 35, 36, 37]. We did preliminary tests with a widow size of 10M instructions but found similar results.

[2] The parallel versions of PARSEC benchmarks may create more threads than specified in the input parameter. We therefore use the serial version of the benchmarks.

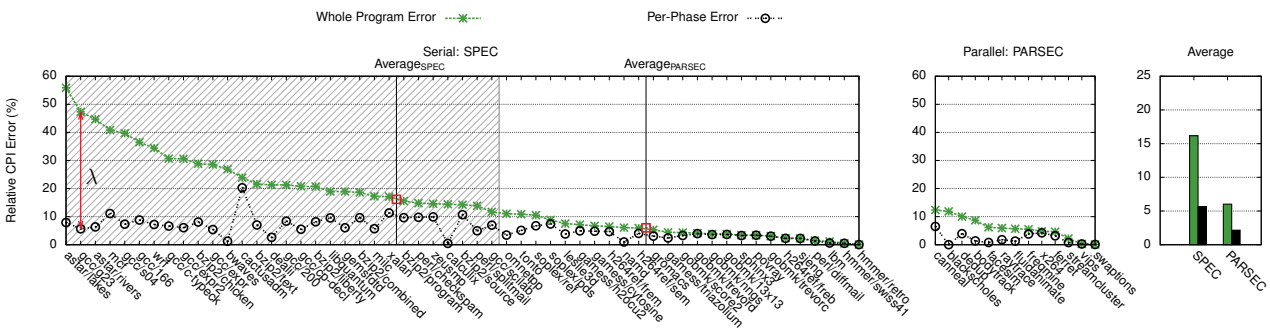


Figure 3. The relative CPI error between the CPI in each window and the average CPI for the whole program (Whole Program Error) and average CPI per phase (Per-Phase Error). All SPEC programs in the gray region have higher CPI error than all the PARSEC programs. The figure shows the importance of using phase-guided runtime optimizations. The larger the difference (λ) between the top line (Program) and the bottom line (Phase), the more important it is to use phase information for runtime optimizations. *This shows that tool developers will find phase-guided runtime optimizations to be more effective for the serial SPEC benchmarks than the parallel PARSEC benchmarks, because SPEC has more time varying behavior.*

calculated using all windows thereby ignoring phases), and within a phase (i.e., the CoV is calculated per phase, then weighted with the size of the phase).

Figure 2 presents the CPI CoV for the benchmarks. The benchmarks suites have been sorted in descending order based on whole program CPI CoV. The gray area highlights the difference between the benchmarks suites. All SPEC benchmarks in the gray region (42% percent of SPEC) have higher program CPI CoV than all the PARSEC benchmarks except dedup. On average, the whole program CPI CoV is 22% and 15% for SPEC and PARSEC respectively. With the exception of dedup (discussed later), the SPEC benchmarks have significantly more time varying behavior than PARSEC. As expected, the CPI variations within phases (7.9% SPEC and 2.8% PARSEC) is much lower compared to the whole program. This means that examining application behavior in terms of phases is far more accurate than just looking at the program average for both benchmark suites. However, the results show that it is more important to examine phases in SPEC than PARSEC since it has larger variation in CPI.

The dedup benchmark has a set of phases in the beginning and the end of its execution (see Figure 9) with a much higher CPI than the rest of the execution. This results in a large standard deviation, and thus a high CPI CoV. However, while this is important for understanding the program behavior (e.g., phase-guided profiling), in terms of other phase-guided runtime optimizations (e.g., DVFS), the phases are short and can sometimes be averaged out. To examine the effects of phase-guided runtime optimizations on SPEC and PARSEC, we consider a hypothetical runtime system. The runtime system can make a decision at every window. It can either base the decision on the average CPI for the whole program execution (Program) (e.g., for DVFS, the frequency is set once when the application starts) or the average CPI of the phase the window belongs to (Phase) (e.g., the frequency is changed when the application changes phase).

Figure 3 presents the relative CPI error for the benchmarks, and it shows the importance of using phase-guided runtime optimizations. The larger the difference (λ) between the top lines (Whole Program Error) and the bottom lines (Per-Phase Error), the more important it is to use phase information for runtime optimizations. As expected, the dedup benchmark has a relative low program CPI error compared to the CPI CoV. All SPEC benchmarks in the gray region (52% percent of SPEC) have higher program CPI error than all the PARSEC benchmarks. On average, the program CPI error is 16% and 6% for SPEC and PARSEC respectively, and 5.6% and 2.2% for phases. This means that phase-guided runtime optimizations (e.g., DVFS) will have a larger impact on SPEC than PARSEC compared to static approaches where the optimization is done once per application. For example, swaptions (PARSEC) has negligible variations in CPI. Setting the optimal CPU frequency once at the start of the execution will therefore provide similar results as setting it per phase, but astar/lakes (SPEC) on the other hand has large variation in CPI so that setting the optimal frequency per phase will provide much better results than once at the start of the execution.

4.2. Phase Behavior

In the previous section we examined the CPI and how performance varies over time. However, several phases can have similar CPI but different behavior in other metrics. To understand how the phases changes over time, and not just CPI, we look at the number of detected phases, and the

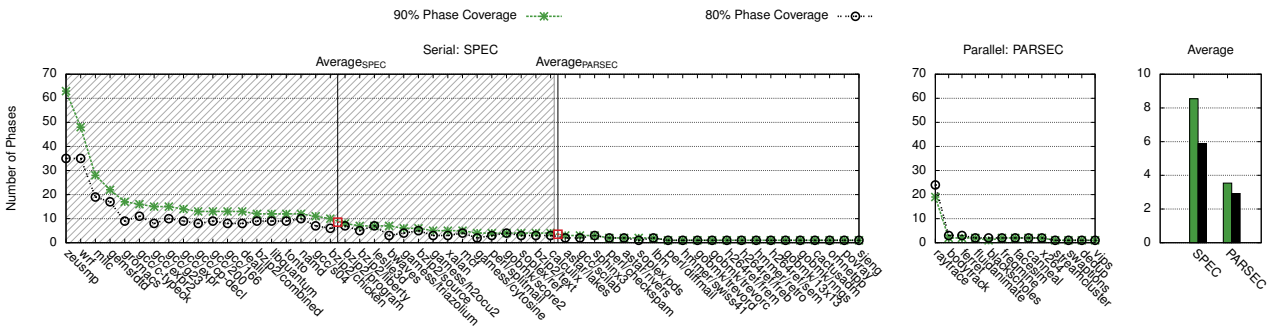


Figure 4. The number of phases that are needed to cover 80% and 90% of the program execution. The gray area highlights the difference between the benchmarks suites. All SPEC programs in the gray region have more phases than all the PARSEC program except raytrace. *This shows that SPEC has more phases (2.4× on average). This is important for the overhead with phase-guided profiling (i.e., profile each phase once), which is proportional to the number of phases. Profiling a SPEC benchmark will therefore on average take 2.4× longer then a PARSEC benchmark.*

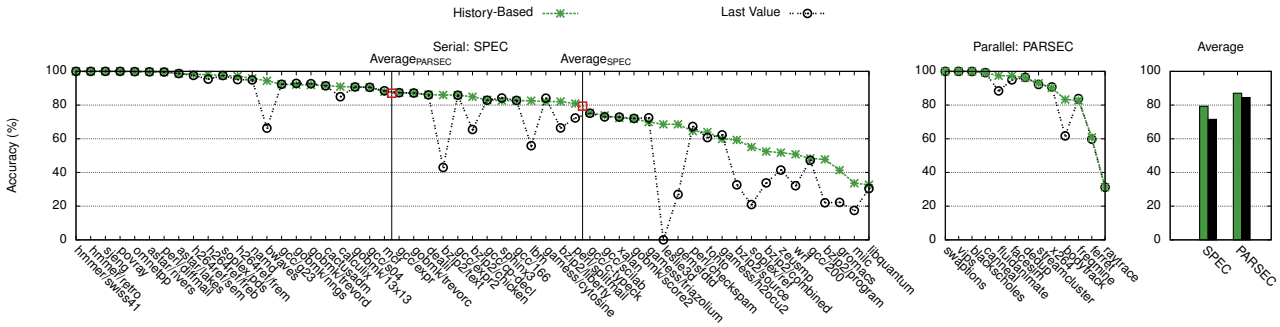


Figure 5. The prediction accuracy of predicting the phase id of the next window, using last value prediction (LV) and history based prediction with run length encoding (RLE). *The two benchmarks suites show similar accuracy, with a slight advantage (i.e., easier to predict) to PARSEC.*

corresponding pattern the phases appear in.

Figure 4 presents the number of phases that are needed to cover 80% and 90% of the program execution[3]. The figure shows that SPEC has significantly more program phases than PARSEC. All SPEC benchmarks in the gray region (59% percent of SPEC) have more phases than all the PARSEC benchmarks except raytrace. On average, the number of detected phases for 90% of the execution is 8.5 and 3.5 for SPEC and PARSEC respectively, and 5.9 and 2.9 for 80% of the execution.

A consequence of this is for example the overhead of phase-guided profiling which is proportional to the number of detected phases (i.e., one window from each phase is profiled). This means that it takes on average 2.4× longer to profile and understand the behavior of 90% of the program execution for a SPEC benchmark than a PARSEC benchmark, and 2× to understand 80% of the program execution.

Applying phase-guided runtime optimizations to an application that frequently changes phase can be more difficult than one with fewer phase changes but the same number of phases (e.g., A, A, B, B vs. A, B, A, B). For example, the phases must be long enough to change frequency (DVFS) or cache size (dynamic cache resizing). Figure 5 presents the prediction accuracy of predicting the phase of the next window using last value prediction and history-based prediction. In addition to simply comparing the accuracies, the last value predictor also describes how often the application changes phase (i.e., high prediction accuracy means few phase changes), while the history-based prediction shows how complex the behavior is (i.e., low prediction accuracy means a more complex pattern). History based prediction shows as expected a better accuracy compared to last value. For leslie3d (SPEC), nearly every window is a phase change, hence a very low accuracy for last value[4]. Overall, the two

[3]We do not consider 100% because it will include more transition-phases [27], that is phases with windows that may appear between phase changes and contain code from two distinct phases. The transition phases, are few however, and can be miss leading so we ignore them in this analysis.

[4]This can happen due to aliasing and when the window size does not

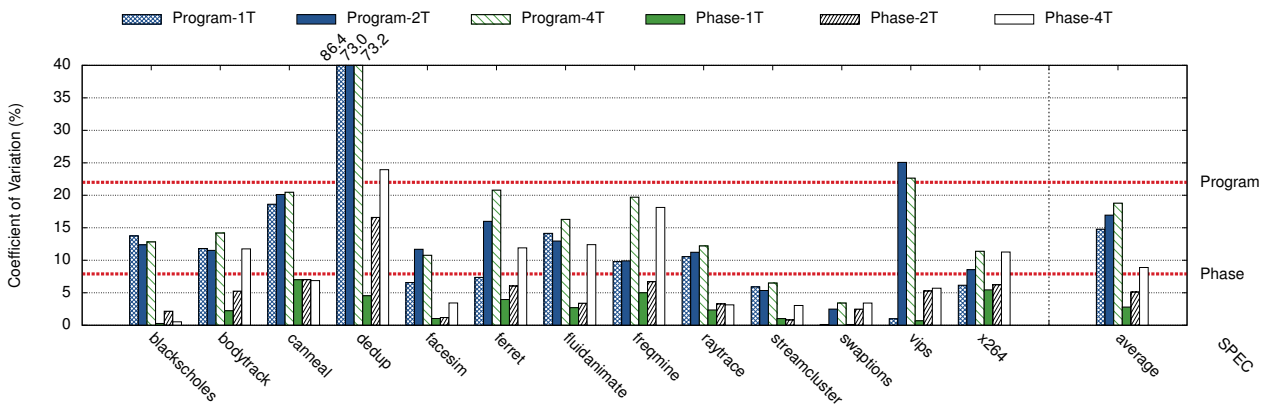


Figure 6. The CPI CoV (standard deviation divided be the mean) for 1, 2 and 4 threads. The figure shows how the CPI varies over the whole execution (Whole Program CoV) and within a phase (Per-Phase CoV). *The CPI variations increases slightly with more threads for both the whole execution and within phases due to competition of shared resources between threads.*

benchmarks suites show similar accuracies, with a slight advantage (easier to predict) to PARSEC. On average, the prediction accuracy for last value prediction is 72% and 84% for SPEC and PARSEC respectively, and 85% and 87% for history-based prediction.

4.3. Summary

The serial phase behavior characterization of SPEC and PARSEC shows that the SPEC benchmarks have both more program phases and exhibit larger variation in CPI. On average, SPEC has $2.4\times$ more phases than PARSEC for 90% of the execution. Using only PARSEC for testing and evaluation can therefore be dangerous since all the effects of phase variations might not be noticed to the same extent.

5. Parallel Phase Behavior

In this section we characterize the parallel phase behavior in PARSEC when running 1 (serial), 2 and 4 threads. To detect phases in parallel applications, we extend the ScarPhase library to monitor and detect phases at runtime in multiple threads. To do so, ScarPhase asynchronously detect phases in the application. Whenever a thread finishes executing a window, ScarPhase classifies what phase the window belongs to using the same method as in the serial version, then waits for the next window to be finished, and so on. ScarPhase will thus alternate between threads when detecting phases. For example, if thread 1 executes windows A_1, A_2, A_3 and thread 2 executes B_1, B_2, B_3, ScarPhase may classify the windows in the following order, $A_1, B_1, A_2, B_2, A_3, B_3$. Important to remember is that the execution is divided into *executed* instructions. This means that two windows can take different amount of time to complete. ScarPhase may therefore instead classify the windows in the following order, $A_1, A_2, B_2, A_3, B_2, B_3$, if phase A executes faster than phase B.

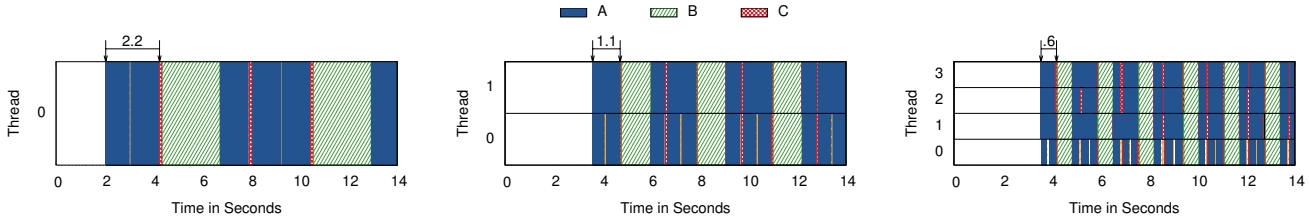
In addition to using shared data structures for phase classification, the prediction lookup tables for history based prediction can also be shared between threads. For example, if thread 1 executes phases A, A, A, B and thread 2 executes A, A, A then we can predict that thread 2 will execute phase B. We found however no advantage of using shared lockup tables, the two methods produced similar results with an average accuracy of $\approx 90\%$.

Figure 6 shows how the CPI varies (CPI CoV) over the whole execution (Program) and within a phase (Phase) for different number of threads. The serial versions of blackscholes and dedup have more program variations than their parallel versions. However, because of more interference between threads for shared resources, the overall CPI variations for all benchmarks increases slightly with more threads for both the whole program execution and within phases. On average, the whole program CPI CoV is 15%, 17% and 19% for 1, 2 and 4 threads respectively. As expected, the CPI variations within phases (2.8%, 5.1% and 8.9%) are lower compared to the whole program.

5.1. Parallelization Models

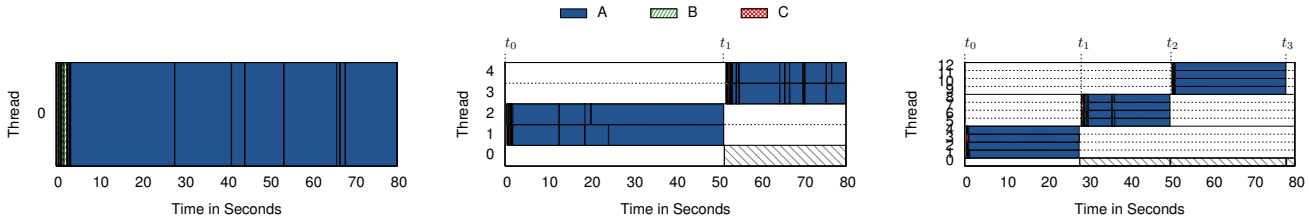
The PARSEC benchmarks utilizes two different parallelization models. The benchmarks dedup and ferret are pipeline-parallel while the rest are data-parallel. To highlight the difference between the two models we plotted the phase behavior over time for facesim in Figure 7, streamcluster in Figure 8 and dedup in Figure 9. The figures show the detected program phases (color) as a function of time (x-axis) for the different threads (y-axis). The largest phases

match the underlying phase bahavior [25].



(a) 1 threads (b) 2 threads (c) 4 threads

Figure 7. The detected program phases (color) using ScarPhase as a function of time (x-axis) for the beginning of facesim's execution. The largest detected phases are colored and named above, with shorter (fewer executed instructions) phases shown in white for clarity. *The facesim benchmark is data-parallel and has two primary phases, A and B, executed in an alternating pattern. Data-parallel application divide the work between threads. The length of the phase will therefore shrink with more threads. For example, the first instance of phase A executes for 2.2 seconds with one thread, but only for 1.1 seconds with two threads (i.e., linear speedup).*



(a) 1 threads (b) 2 threads (c) 4 threads

Figure 8. The detected program phases (color) using ScarPhase as a function of time (x-axis) for the beginning of streamcluster's execution. The largest detected phases are colored and named above, with shorter (fewer executed instructions) phases shown in white for clarity. *The streamcluster benchmark creates new threads in each iteration (i.e., thread 1 and 2 starts at t_0 in Figure 8b and stops at t_1). Using thread private clusters for phase classification would therefore create a significant amount of duplicated phase ids.*

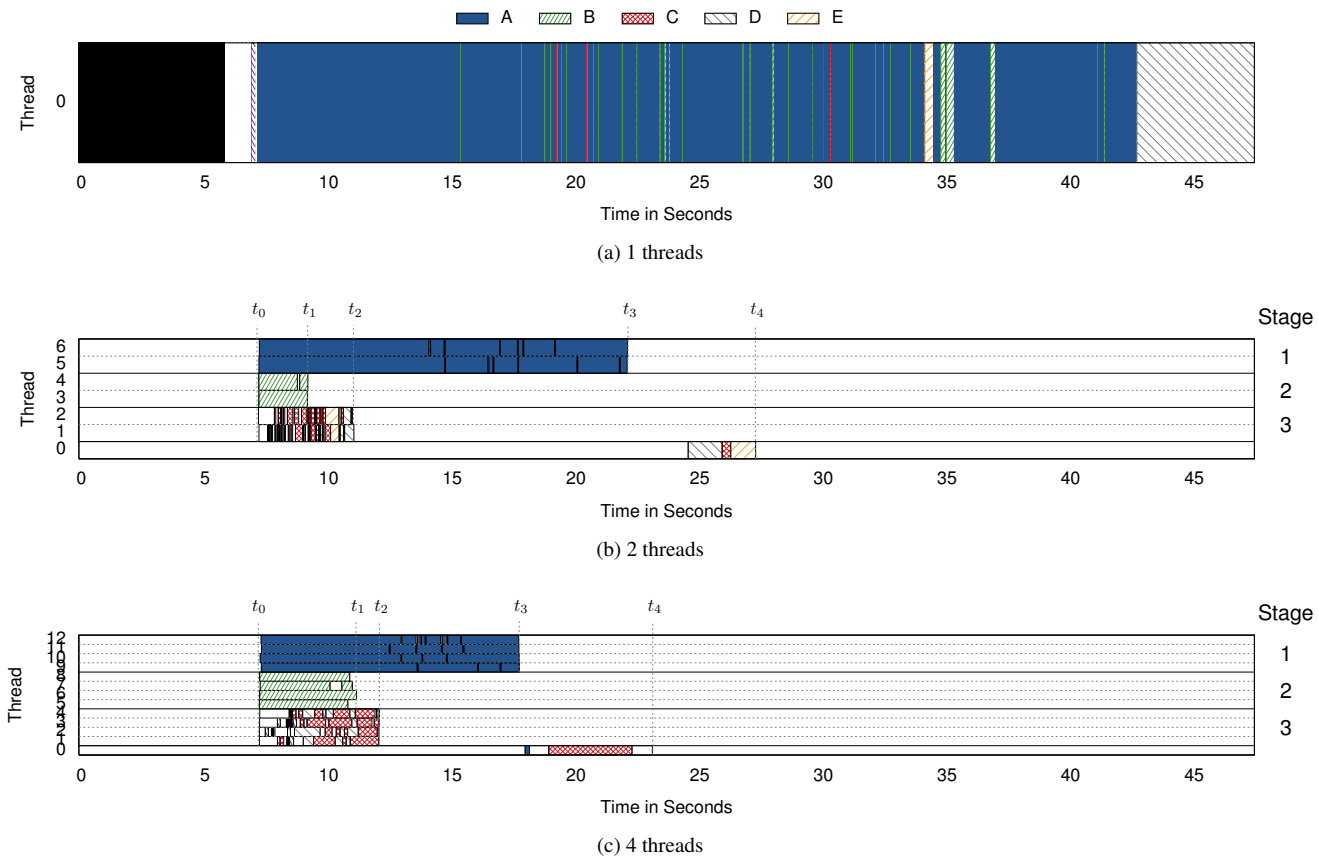
are colored and named above, with shorter (fewer executed instructions) phases shown in white for clarity. We record when windows start and stop executing, and plot the phase for each window. However, since the windows are measured uniformly in executed instructions, they can take different amounts of time to complete. For example, windows can be executed with different speeds depending on phase, or the kernel can put the thread to sleep. A control thread (e.g., thread 0 in streamcluster and dedup) that only start work-threads and then goes to sleep will have few execution windows (colored in white) but the thread will take a long time to complete.

Data-parallel. Figure 7 shows the detected program phases for facesim. It has two primary phases, A and B, executed in an alternating pattern. Because data-parallel application split the work between threads, the length of each phase will shrink with more threads, as can be seen in the figure. For example, the first instance of phase A has a linear speedup from 1 to 2 threads. It executes for 2.2 seconds with 1 thread, but only 1.1 seconds with 2 threads. Another characteristic of data-parallel applications is that all threads usually execute the same phases. However, the phases are not necessary aligned in time. Meaning, thread 1 could execute phase A at the same time as thread 2 execute phase B.

The benchmark streamcluster is also data-parallel, but it has noticeably different behavior. Figure 8 shows how streamcluster creates new work-threads in each iteration. For example in Figure 8b, threads 1 and 2 start to execute at t_0 and they terminate at t_1, where thread 3 and 4 starts.

Pipeline-parallel. Figure 9 shows the detected program phases for dedup's whole execution. It executes different stages (phases) in different threads. For example, in Figure 9c, Stage 1 has 2 threads and executes phase A, while Stage 2 executes phase B. The three stages starts to execute at t_0, and they stop at t_1, t_2 and t_3 for stage 2, 1 and 3 respectivly. The program finally terminates at t_4. It oversubscribes the system for load balancing (i.e., stage 1 executes much longer than stage 2 and 3). While the phase behavior in stage 1 and 2 is homogeneous, stage 3 has some phase changes.

Only one phase is executed in stage 1 and 2. This means that setting the frequency (DVFS) once per thread



(a) 1 threads

(b) 2 threads

(c) 4 threads

Figure 9. The detected program phases (color) using ScarPhase as a function of time (x-axis) for the whole execution of dedup. The largest detected phases are colored and named above, with shorter (fewer executed instructions) phases shown in white for clarity. *The dedup benchmark is pipeline-parallel and executes different stages (phases) in different threads. It oversubscribes the system for load balancing (i.e., stage 3 executes much longer than stage 2). The three stages starts to execute at t_0, and they stop at t_1, t_2 and t_3 for stage 2, 1 and 3 respectivly. The program finaly terminates at t_4. While the phase behavior in stage 1 and 2 is homogeneous, stage 3 shows that pipeline-parallel programs can still benefit from phase-guided runtime optimizations (e.g., DVFS).*

in those stages will produce similar results as setting the frequency per phase. To see if this applies to the other programs as well, we have plotted the CPI variations (CPI CoV) for the whole program (Program), within threads (Thread), within phases (Phase) and within phases per thread (Phase+Thread) (i.e., the CPI CoV is calculated per phase using windows from one thread, then averaged across all threads) in Figure 10.

The figure shows that the CPI variations within threads are much lower than the variation within the whole execution for blackscholes, streamcluster and the two pipeline-parallel benchmarks dedup and ferret. The benchmark blackscholes has 1 control thread and 2 computation threads. The CPI is very different between the control and computation threads, but rather homogeneous in each thread. Streamcluster, on the other hand, creates 10 computation threads (Figure 8). The CoV is lower just as a consequence of dividing the execution into smaller pieces. As expected, for facesim[5], there is no difference between the variations within threads and within the whole execution as can be seen in Figure 7. However, dividing the execution into phases provides better results across all benchmarks, including pipeline-parallel applications. On average, the CPI CoV is 17%, 9%, 5% and 4% for Program, Thread, Phase and Phase+Thread respectively.

5.2. Summary

The overall CPI variations and number of detected phases are lower for PARSEC compared to SPEC as seen in section 4. However, the PARSEC benchmarks show a diverse

[5]If the results from the NAS [4] benchmarks were to be included, the overall behavior would be more similar to that of facesim.

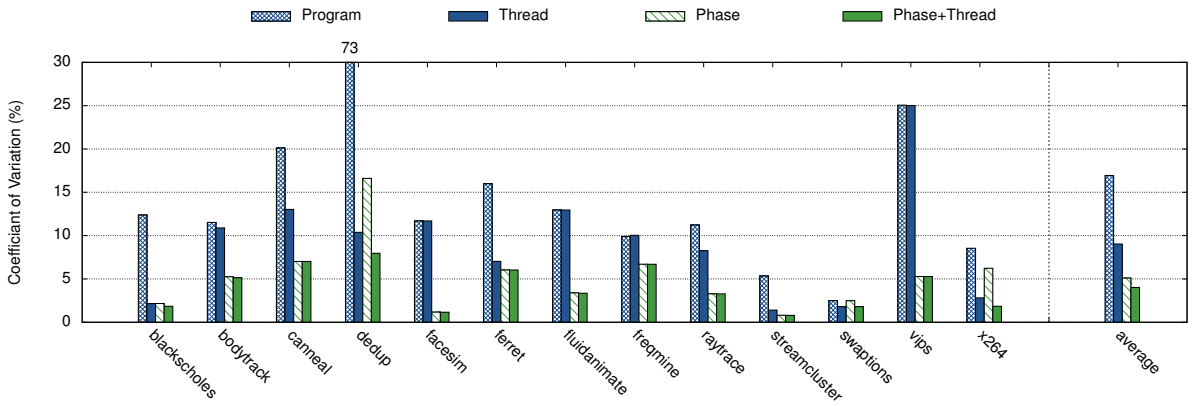


Figure 10. The CPI CoV for the whole program (Program), per thread (Thread), per phase (Phase) and per phase within each thread (Phase+Thread) using 2 threads. ***Dividing the execution into threads provides accurate results for blackscholes, streamcluster and for the two pipeline-parallel benchmarks dedup and ferret. However, dividing the execution into phases provides better results across all benchmarks, including pipeline-parallel applications.***

set of phase behaviors which are important to understand when developing new runtime optimizations for parallel applications.

6. Phases in the many-core era

In the previous section we ran PARSEC with 1 to 4 threads. However, next generation processors will have many more cores. In this section, we investigate how phase-guided optimizations are affected when scaling the number of threads (i.e., strong scaling) into the many-core region. To approximate the behavior of a many-core machine, we used a Intel Xeon X6550 (Nehalem) 8 sockets machine with 8 cores per chip (64 core machine).

We examine dedup (pipeline-parallel) and fluidanimate (data-parallel). The benchmarks fluidanimate and facesim have similar phase behavior, but fluidanimate has shorter phases which makes it easier to analyze. Figure 11 shows the next window prediction accuracy for different number of threads, using last value prediction (LV) and history based prediction (RLE). The prediction accuracy remains relatively unchanged for dedup since Stage 1 and 2 (Figure 9) does not have any phase changes[6]. However, the length of fluidanimate's phases shrinks (lower prediction accuracy for last value) as the number of threads increases in Region A (i.e., it divides the work between more threads (see Figure 7)). When the length of the phases shrinks below the windows size, the different phases are merged into one phase (100% prediction accuracy) in Region B. We make three interesting observations and discuss them below.

[6]Both last value prediction and history based prediction have similar prediction accuracies since history based prediction automatically falls back to last value prediction when there is no phase patterns (i.e., only one long phase for Stage 1 and 2).

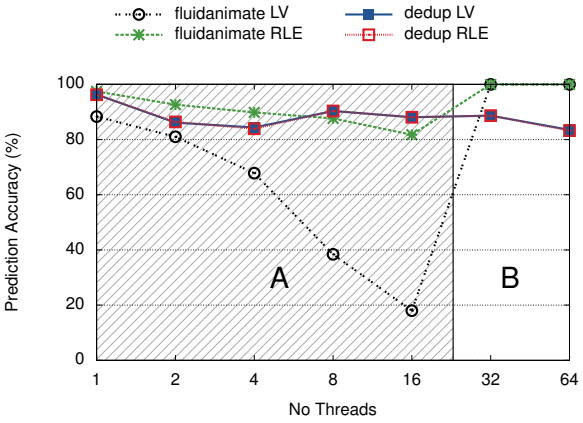


Figure 11. Next window prediction accuracy for different number of threads for dedup (pipeline-parallel) and fluidanimate (data-parallel), using last value prediction (LV) and history based prediction with run length encoding (RLE). ***The length of the phases shrinks (lower prediction accuracy for last value) as the number of threads increases in Region A. When the length of the phase shrinks below the windows size, the different phases are merged into one phase (100% prediction accuracy) in Region B.***

Prediction. The history based predictor has a more stable and a higher prediction accuracy than the last value predictor. This means that more advanced phase predictors are needed for reliable prediction across executions with different number of threads or systems that can change number of threads at runtime.

Phase change frequency. Phase changes occurs more frequently since the phases shrink. Usually there is a cost associated when some phase-guided runtime optimizations changes a setting (e.g., power and time to turn on and off parts of the cache, or the cost of migrating a thread). This means that the overhead of the runtime system will increase with more threads, since the cost will be payed more frequently.

Homogeneity. The final observation is that only one phase is detected when using more than 32 threads. This means that the runtime behavior *appears* homogeneous across the whole execution. Phase-guided runtime optimizations will therefore be less useful when running many threads. For example, setting the frequency once for the whole execution will be the same as setting it per phase.

One solution to these observations is to shrink the window size when executing more threads. However, some runtime optimizations have a fixed limit on how fast they can react (e.g., time before the new CPU frequency can take effect). Another solution is to also use weak scaling (i.e., scale the problem size when using more threads). The amount of work per thread would therefore remain the same, meaning that the length of the phases would not change. However, scaling the problem size is not always feasible or desirable (e.g., encoding a movie with x264). Both the number of threads (strong scaling), problem size (weak scaling) and the speed (window size) of the runtime optimization must therefore be considered when implementing phase-guided runtime optimizations for parallel applications.

7. Related Work

Perelman et al. [35] extended SimPoint [40] to detect phases in parallel applications, and they used it to find architecture simulation points in the OpenMP version of the NAS [4] benchmarks. We also investigated the NAS benchmarks, but found that they exhibited a very limited set of phase behaviors. Most of them had execution behaviors very similar to facesim. Due to space limitation we do not include those results in this paper.

Biesbrouck et al. [9, 10] suggested a co-phase matrix to reduce the overhead of simulating symmetric-multithreading (SMT). The idea is to only simulate each phase combination once. However, they looked at multi-process workloads using SPEC (e.g., co-schedule gcc with mcf) and not multi-threaded applications. A co-phase matrix could be combined with ScarPhase to find unique phase combination across the threads, which we plan to investigate in future work.

Ipek et al. [20] extended hardware-based phase tracking [41, 27] for parallel processors with distributed shared-memory. They observed that the relationship between executed code and CPI decreased with the number of threads. Whether a co-phase matrix would solve this problem was not investigated, instead they proposed to also track data contention and data distribution along with the executed code.

Various related phase researchers [39, 24, 2] have observed that phase behavior depends on the size of the sampled windows. Dividing the execution into windows effectively averages the execution: the smaller the windows are, the larger the variations will be, and vice versa. Transition phases (i.e., windows between two phases) can also be misleading, since they contain code from two phases. One solution is to not divide the execution into windows, but to instead monitor the call stack [17, 22, 28], and divide the execution when the call stack changes depth (i.e., phase).

Bienia et al. [7] compared PARSEC with SPLASH-2 [43]. However, they only looked at aggregated values and focused on metrics related to multi-processors. Bhadauria et al. [5] examined PARSEC using a range of different performance metrics on several multi-processors and Bhattacharjee et al. [6] characterized the TLB behavior.

8. Conclusions

In this paper we have compared the difference in runtime phase behavior between SPEC and PARSEC. We found that the SPEC benchmarks have many more program phases ($2.4\times$) and larger variations in CPI ($1.5\times$). Using only SPEC for evaluating phase-guided runtime optimizations may therefore be misleading, and not show all the possible performance improvements. For example, a new DVFS optimization will have more opportunities to change the frequency in SPEC than PARSEC, and it gets worse with more threads. In the future, we plan to look at other parallel workloads (e.g., commercial and database applications).

We then extended the ScarPhase library to detect phases in parallel applications and used it to characterize the phase behavior in PARSEC. Even though the CPI variations are not as significant as SPEC's, it contains a diverse set of phase behaviors.

Finally, we performed a case study to investigate how phase-guided optimizations are effected when scaling the number of threads into the many-core region. We showed that as the number of threads increases, the phases shrink until all phases are smaller than the window size. The runtime behavior will then appear homogeneous which can prevent phase-guided runtime optimizations.

References

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

[2] K. K. Agaram, S. W. Keckler, C. Lin, and K. S. McKinley. Decomposing memory performance: data structures and phases. In *Int. Symposium on Memory management*, 2006.

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

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. *Int. Journal of Supercomputer Applications*, 1991.

[5] M. Bhadauria, V. M. Weaver, and S. A. McKee. Understanding parsec performance on contemporary cmps. In *Int. Symposium on Workload Characterization*, 2009.

[6] A. Bhattacharjee and M. Martonosi. Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors. In *Int. Conf. on Parallel Architectures and Compilation Techniques*, 2009.

[7] C. Bienia, S. Kumar, and K. Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Int. Symposium on Workload Characterization*, 2008.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Int. Conf. on Parallel Architectures and Compilation Techniques*, 2008.

[9] M. V. Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *Int. Symposium on Performance Analysis of Systems and Software*, 2004.

[10] M. V. Biesbrouck, L. Eeckhout, and B. Calder. Considering all starting points for simultaneous multithreading simulation. In *Int. Symposium on Performance Analysis of Systems and Software*, 2006.

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

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

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

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

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

[16] L. Eeckhout, J. Sampson, and B. Calder. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *Int. Symposium on Workload Characterization*, 2005.

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

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

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

[20] E. Ipek, J. Martinez, B. de Supinski, S. McKee, and M. Schulz. Dynamic program phase detection in distributed shared-memory multiprocessors. In *Int. Symposium on Parallel and Distributed Processing*, 2006.

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

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

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

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

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

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

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

[28] J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. In *Int. Symposium on Code Generation and Optimization*, 2006.

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

[30] K. Meng, R. Joseph, R. P. Dick, and L. Shang. Multi-optimization power management for chip multiprocessors. In *Int. Conf. on Parallel Architectures and Compilation Techniques*, 2008.

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

[32] P. Nagpurkar, C. Krintz, M. Hind, P. F. Sweeney, and V. T. Rajan. Online phase detection algorithms. In *Int. Symposium on Code Generation and Optimization*, 2006.

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

[34] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *Int. Conf. on Parallel Architecture and Compilation Technique*, 2003.

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

[36] A. Sembrant, D. Eklov, and E. Hagersten. Efficient software-based online phase classification. In *Int. Symposium on Workload Characterization*, 2011.

[37] A. Sembrant, D. Black-Schaffer, and E. Hagersten. Phase guided profiling for fast cache modeling. In *Int. Symposium on Code Generation and Optimization*, 2012.

[38] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2004.

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

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

[41] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Int. Symposium on Computer Architecture*, 2003.

[42] T. Sondag and H. Rajan. Phase-based tuning for better utilization of performance-asymmetric multicore processors. In *Int. Symposium on Code Generation and Optimization*, 2011.

[43] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Int. Symposium on Computer Architecture*, 1995.