UPPSALA
UNIVERSITET

IT Licentiate theses
2012-009

Efficient Techniques for Detecting and Exploiting Runtime Phases

ANDREAS SEMBRANT

UPPSALA UNIVERSITY
Department of Information Technology

Efficient Techniques for Detecting and Exploiting Runtime Phases

Andreas Sembrant
andreas.sembrant@it.uu.se

December 2012

Division of Computer Systems
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

Dissertation for the degree of Licentiate of Philosophy in Computer Science

ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden

Abstract

Most applications have time-varying runtime phase behavior. For example, alternating between memory-bound and compute-bound phases. Nonetheless, the predominant approach in computer research has been to categorize an application based on its average runtime behavior. However, this can be misleading since the application may appear to be neither memory nor compute bound.

In this thesis we introduce tools and techniques to enable researchers and software developers to capture the true time-varying behavior of their applications. To do so, we 1) develop an efficient technique to detect runtime phases, 2) give new insight into applications' runtime phase behaviors using this technique, and, finally, 3) explore different ways to exploit runtime phase detection.

The results are ScarPhase, a low-overhead phase detection library, and three new methods for exploring applications' phase behaviors with respect to: 1) cache performance as a function of cache allocation, 2) performance when sharing cache with co-running applications, and finally, 3) performance and power as a function of processor frequency. These techniques enable us to better understand applications' performance and how to adapt different settings to runtime changes. Ultimately, this insight allow us to create new faster and more power efficient applications and runtime systems that can better handle the increasing computation demands and power constraints of tomorrow's problems.

Acknowledgments

I would like to thank all the people who made this thesis possible. To begin with, I would like to thank my two advisers, David Black-Schaffer and Erik Hagersten, for providing guidance and support. In addition, I would like to thank all my co-authors, Andreas Sandberg, David Eklöv, Muneeb Khan, Stefanos Kaxiras, and Vasileios Spiliopoulos, who made it *fun* to write the papers included in this thesis. Furthermore, I would like to thank the rest of the people in our research group and all the people on level two for providing a good working environment. Without these people, it would have been a very long and tedious process to complete this thesis.

However, life is not *all* about science and research. It is about sports, and pushing yourself to the limit (*maybe*). Therefore, I would also like to thank friends and family who have helped me to finish both a *Swedish Classic*[1] and to become an *Ironman*[2] during my time working on this thesis. This have been two great adventures to take my mind of research, and I have already signed up for more (*hint* another Ironman).

Finally, a special thanks to my parents for there never ending support. Be it driving me to the airport for conferences or sheering me on during sports events. They have always been able to help when needed. In the same regard, I would also like to thank my two sisters and my niece. Without them, I would have a much harder time to concentrate and focus on research and other things.

All in all, many thanks to the people above and to all those who have helped but have not been mentioned in person.

[1] An award for finishing four races in different disciplines within 12 months (Vasaloppet (90km cross-country skiing), Vätternrundan (300km biking), Vansbrosimningen (3km swimming) and Lidingöloppet (30km cross terrain running)).

[2] A title for completing an Ironman triathlon event (3.86km swimming + 180.2km biking + 42.195km running, followed directly after each other).

This work was supported in part by the Swedish Foundation for Strategic Research through the *CoDeR-MP* (Computationally Demanding Real-Time Applications on Multicore Platforms) project and by the Swedish Research Council within *UPMARC* (Uppsala Programming for Multicore Architecture Research Center).

List of Papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals:

I Andreas Sembrant, David Eklov, and Erik Hagersten. "Efficient Software-based Online Phase Classification". In: *Int. Symposium on Workload Characterization (IISWC)*. 2011

II Andreas Sembrant, David Black-Schaffer, and Erik Hagersten. "Phase Behavior in Serial and Parallel Applications". In: *Int. Symposium on Workload Characterization (IISWC)*. 2012

III Andreas Sembrant, David Black-Schaffer, and Erik Hagersten. "Phase Guided Profiling for Fast Cache Modeling". In: *Int. Symposium on Code Generation and Optimization (CGO)*. 2012

IV Andreas Sandberg, Andreas Sembrant, Erik Hagersten, and David Black-Schaffer. "Modeling Performance Variation Due to Cache Sharing". In: *Int. Symposium on High-Performance Computer Architecture (HPCA)*. 2013

V Vasileios Spiliopoulos, Andreas Sembrant, and Stefanos Kaxiras. "Power-Sleuth: A Tool for Investigating your Program's Power Behavior". In: *Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2012

All papers are reprinted verbatim and have been reformated to the one-column format of this book. Due to time contraints, Paper IV is a reprint of the submitted (accepted) version.

Comments on my participation:

I I am the principal author and principal investigator.

II I am the principal author and principal investigator.

III I am the principal author and principal investigator.

IV My main contributions were:

- I applied and shared my insights and results on runtime phases from Paper I, II and III.
- I helped to setup and use the phase detection infrastructure developed in Paper I, II and III.
- I ran experiments and collected all reference data.
- I wrote and created figures for different parts of the paper.
- I performed the final reorganization and rewrite of the paper.

V My main contributions were:

- I applied and shared my insights and results on runtime phases from Paper I, II and III.
- I modified, helped to setup and use the phase detection infrastructure developed in Paper I, II and III.
- I wrote and created figures for different parts of the paper.

Other publications by the author not included in this thesis:

A Muneeb Khan, Andreas Sembrant, and Erik Hagersten. "Low Overhead Instruction-Cache Modeling Using Instruction Reuse Profiles". In: *Int. Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2012

Contents

Glossary

Basic Block Vector (BBV) An architectural independent performance metric. Each entry in the vector describes how many times it's corresponding basic block was executed in a window.

Cache Pirating A hardware performance counters based method for measuring an application's cache miss rate, CPI, etc as a function of cache allocation.

Execution Window An application's execution is divided into non-overlapping fixed-sized windows of 100M executed instructions (also referred to as an *execution interval*).

Hardware Performance Counters Special-purpose registers found in modern computers for counting hardware related events (e.g., CPI or cache misses).

Last-Value Predictor A simple phase predictor that always predicts that the next window will belong to the same phase as the previous window. That is, it predicts no phase change.

Periodic Profiling A type of performance optimization that reduces the overhead of profiling by only profiling a randomly selected subset of application's runtime execution (i.e., the overhead is a function of profile/sample rate).

Phase Periods of execution with similar behavior (i.e., a set of phase instances with similar runtime behavior).

Phase Instance A period of execution with similar behavior (i.e., a set of adjacent windows with similar runtime behavior).

Phase Prediction A set of methods for predicting what phase the next window will belong to. This is needed since the phase of the current window in only identified after the window has been executed.

Phase-Guided Profiling A type of performance optimization that reduces the overhead of profiling by only profiling a small part of each phase (i.e., the overhead is proportional to number of phases).

Phase-Guided Runtime Optimization A type of runtime optimization that monitors applications' runtime phases and applies the best optimization to each phase.

Power-Sleuth A tool for profiling an application's power and performance phase behavior as a function of processor frequency.

Precise Event Based Sampling (PEBS) Intel specific extension to hardware performance counters to save register contents (e.g., the instruction pointer) when a counter triggers.

Run-Length Encoded Predictor An advanced phase predictor that predict what phase the next window will belong to based on previously seen behavior.

ScarPhase *(Sample-based Classification and Analysis for Runtime Phases).* A low overhead tool and library for detecting runtime phases.

StatCache A low overhead statistical cache model for modeling an application's cache miss-ratio as a function of cache allocation.

Abbrevations

BBV Basic Block Vector.

BRV Branch Vector.

CBRV Conditional Branch Vector.

CMP Chip Multiprocessor.

COV Coefficient of Variation.

CPI Cycles Per Instruction.

CPU Central Processing Unit.

DVFS Dynamic Voltage Frequency Scaling.

IPC Instructions Per Cycle.

L1 Level One Cache.

L2 Level Two Cache.

L3 Level Three Cache.

MBBV Mapped Basic Block Vector.

PGO Phase-Guided Runtime Optimization.

SLLC Shared Last Level Cache.

Chapter 1

Introduction

Most applications have time-varying phase behavior [43, 45, 11, 8]. One example of this is an application that alternates between memory-bound and compute-bound phases. Categorizing this application based on its average behavior can therefore be misleading since the application may appear to be neither memory nor compute bound.

Nevertheless, the predominant way in computer research has so far been to base many decisions on applications' average runtime behavior. One of the goals of this thesis is therefore to reduce the dependency on applications' average behavior. To do so, we 1) develop an efficient technique to detect runtime phases, 2) give new insight into applications' runtime behaviors, and, finally, 3) explore different ways to exploit runtime phase detection.

1.1 Runtime Phase Behavior

It can be misleading to only consider an application's average runtime behavior. To illustrate why, we can divide an application's execution into non-overlapping windows and examine the behavior in each window. Figure 1.1 shows the overall performance (CPI), memory performance (L2 cache miss ratio), and power behavior (Power), over time, for the whole execution of gcc/166[1] running on an Intel Nehalem machine [30]. Each data-point corresponds to the behavior in one window (100 million executed instructions). The dashed red line shows the average behavior.

The figure clearly demonstrates that the behavior is not constant, but instead changes significantly over time. For example, gcc has two periods of execution with very high CPI, one from 30 to 33 billion in-

[1] In this thesis, we use *<program>/<input>* notation to distinguish the application from input (e.g., gcc from SPEC2006 [16] with input 166.i).

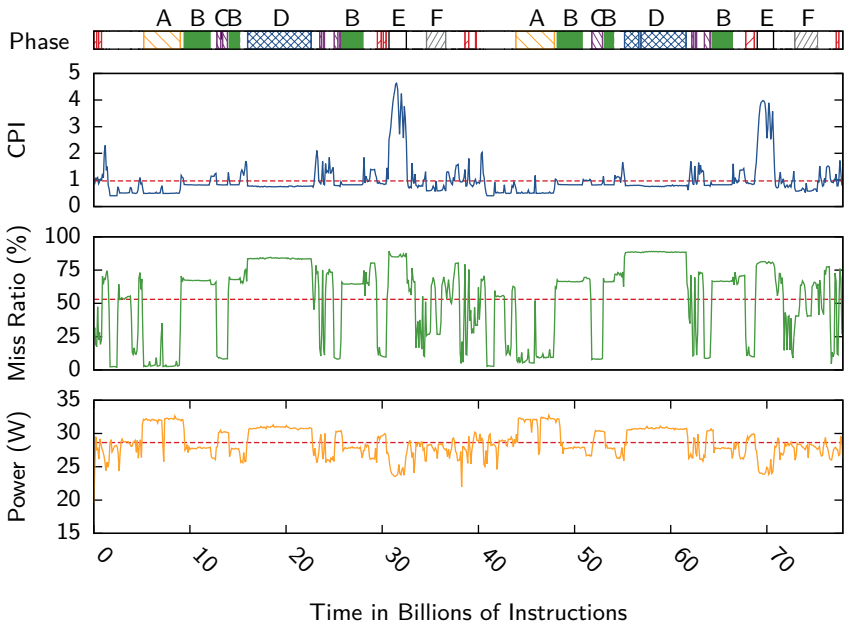


Figure 1.1: CPI, L2 cache miss ratio, and power consumption (y-axis), over time (x-axis), for the whole execution of gcc/166. Each data-point corresponds to the behavior in one window (100M instructions). The dashed red line shows the average behavior, and the colored bars above the graphs shows the detected runtime phases, with smaller phases grouped together in white for clarity. *This demonstrates that classifying an application based on its average behavior can be misleading since it hides transient periods of execution, for example, the high CPI in phase E.*

structions, and one from 69 to 71 billion instructions. However, this cannot be inferred from looking at the average CPI.

These changes in behavior over time are a result of how applications execute their code. For example, when an application starts executing a new function with more memory operations, or when it changes to a larger input set, we might see an increase in miss ratio. Applications with large and diverse code bases and inputs (e.g., gcc) will therefore exhibit large variations over time.

Looking at each individual window is usually not practical because there are just too many windows (e.g., 780 windows in Figure 1.1). Instead, we group windows with similar behavior together into phases. The colored bars above the graphs in the figure shows the detected runtime phases (explained later in Chapter 2), with smaller phases grouped

together in white for clarity.

An execution phase is defined to be a period of execution with stable behavior with respect to a given performance metric (e.g., CPI, or cache miss ratio). For example, gcc has a very stable behavior from 16 to 22 billion of executed instructions (phase D in Figure 1.1). That period is therefore regarded as one phase (i.e., all the windows in that period belongs to phase D). The same behavior can also reappear several times. Disconnected periods of execution with similar behavior are therefore also considered to belong to the same phase. For example, the period from 55 to 61 billion instructions also belongs to phase D.

Furthermore, an application's time varying behavior is not random, but changes in different phase patterns. For example, gcc has a set of phases with different behavior (i.e., phase A, B, C, D, E, and F), that are executed in a specific order (i.e., first phase A, then B, C, B, D, B, E, and finally F). This pattern occurs two times, once during the first half of the execution, and then a second time during the last half.

1.2 Phase-Guided Runtime Optimizations

Each of gcc's phases exhibit a unique set of runtime properties. By taking advantage of phase detection, we can exploit this by applying the best runtime optimization to each phase. This can be used to improve accuracy and performance of several different types of runtime optimizations (e.g., cache resizing [9, 44, 42, 27], dynamic voltage frequency scaling [21], scheduling [46], and phase-guided profiling [34, 40]).

For example, a CPU frequency governor (DVFS) that wants keep the total power consumption below a limit (e.g., for longer battary life or because of thermal limits), might lower the frequency when entering phase A and then restore it to a higher frequency when entering phase B. This enables the governor to be as close to the limit as possible without sacrificing performance since lowering the frequency decreases performance. Note that, if the average (28.6W) is used instead, a more conservative frequency must be chosen to guarantee that phase A does not exceed the limit (i.e., use a lower frequency than necessary which will negatively affect performance for low power phases).

1.3 Contributions

It is therefore important to detect and understand applications' phase behavior. In this thesis we address some of these problems by developing new efficient techniques for detecting and exploiting these runtime

phases. In short, we make contributions in the following areas:

1. **Detecting phases.** How to detect phases at runtime with low overhead using a software-based approach.

2. **Understanding phase behavior.** How applications behave over time, and how phase behavior differ between applications.

3. **Exploiting phase detection.** How to combine phase detection with other methods and optimizations.

1.4 Thesis Structure

This thesis is divided into two parts: detecting phases and exploiting phase detection. In the first part, we develop a tool, ScarPhase (*Sample-based Classification and Analysis for Runtime Phases*, Paper I), to detect runtime phases efficiently, and we use this phase detection tool to classify phase behavior in serial and parallel applications (Paper II). This enables us to easily take advantage of applications' phase behaviors.

The second part consists of three use cases (papers III, IV, and V) where we combine ScarPhase with other methods to improve the accuracy and performance of those methods, but also to give new insights with respect to applications' phase behaviors. This results in new insights into:

1. **Cache allocation.** In Paper III we study how an application's cache performance changes over time depending on how much cache it is allocated.

2. **Cache-sharing effects.** In Paper IV we explore how an application is affected by co-executing applications depending on how their phases overlap and how they compete for shared cache resources.

3. **Processor frequency.** And finally, in Paper V we investigate how an application's power/performance behavior changes over time depending on the processor's clock frequency.

Part I

Detecting Phases

Chapter 2

Detecting Phases

In this chapter we discuss how to detect runtime phases, and we present ScarPhase, a fast online phase detection implementation. We then use ScarPhase to characterize the phase behavior in serial and parallel applications.

2.1 Properties and Requirements

Efficient runtime phase detection techniques (e.g., ScarPhase) need to fulfill several requirements to be generally applicable and useful as tools. These requirements are discussed below.

1. **General purpose phases.** First, we want ScarPhase to detect general purpose phases. Remember that we previously defined a phase to be a period of execution with similar behavior. The behavior can however be similar in one metric but very different in another. For example, phase B and C have similar CPI in Figure 1.1, but noticeable different behavior in L2 cache miss ratio and in power. A potential phase detector that looks at only the CPI would therefore draw the wrong conclusion, and incorrectly classify phase B and C as just one phase instead of two distinct phases. ScarPhase should therefore detect *hardware independent* phases that reflect performance changes across several metrics.

2. **Insensitive to system interference.** Whenever more than one application is co-executed concurrently on the same system the different applications will affect each others' performance due to resource sharing. The phase detector should be insensitive to such interference. The same set of phases should be detected regardless whether the application runs in isolation or whether it is co-executed with other applications. This means that it should not detect false phase changes caused by performance variation due other applications (i.e., a phase change in one

application should not propagate into the other applications).

3. **Transparent and non-intrusive.** ScarPhase should also be easy to deploy. This means that it should be transparent and non-intrusive. For example, it should not require any recompilation of the analyzed application and it should work with dynamically generated code. Furthermore, it should be possible to run the phase detection on applications where the source code is unavailable.

4. **Online.** To detect runtime phases ScarPhase finds periods of execution with similar behavior and groups them together into phases. For ScarPhase to be usable as a component in a runtime systems (e.g., a scheduler), the phase detection must be performed at runtime while the application is running. For example, a scheduler should be notified when an application changes phase.

5. **Low overhead.** Finally, for ScarPhase to be practical as a tool, it must impose a minimal runtime overhead without loss of accuracy and fidelity (i.e., work with latency sensitive runtime systems). Moreover, the overhead must be low enough to not distort any runtime measurements done on the analyzed application.

2.2 Detecting Phases

It has been observed that changes in executed code reflect changes in many different metrics [43, 45, 8, 44, 25] (e.g., different functions have different CPI). To detect general purpose phases, ScarPhase and other previously related phase detection techniques [43, 45, 44, 25, 7, 1, 26, 28, 36] work by monitoring what code is executed, and groups period of execution that executes the code in similar ways into phases (this makes ScarPhase general purpose and insensitive to system interference). Figure 2.1 shows an overview of how this type of phase detection works. The phase detection is done in three steps which are discussed below.

A First, the execution is divided into fixed-sized non-overlapping windows, where each window is 100 million executed instructions long. For example, there are 5 windows (i.e., window $4-8$) in Figure 2.1.

B During each window, executed branch instructions (i.e., basic blocks) are sampled, where the address of the basic block is hashed into a frequency vector (commonly referred to as a *basic block vector* (BBV) [43]) that describes how many times a basic block has been executed during one window[1]. For example, branch B_1 is executed two times while B_2 is executed only once.

[1] A finite sized frequency vector (32 entries) is used to lower the overhead (i.e., the smaller the vector, the faster the clustering can be completed.).

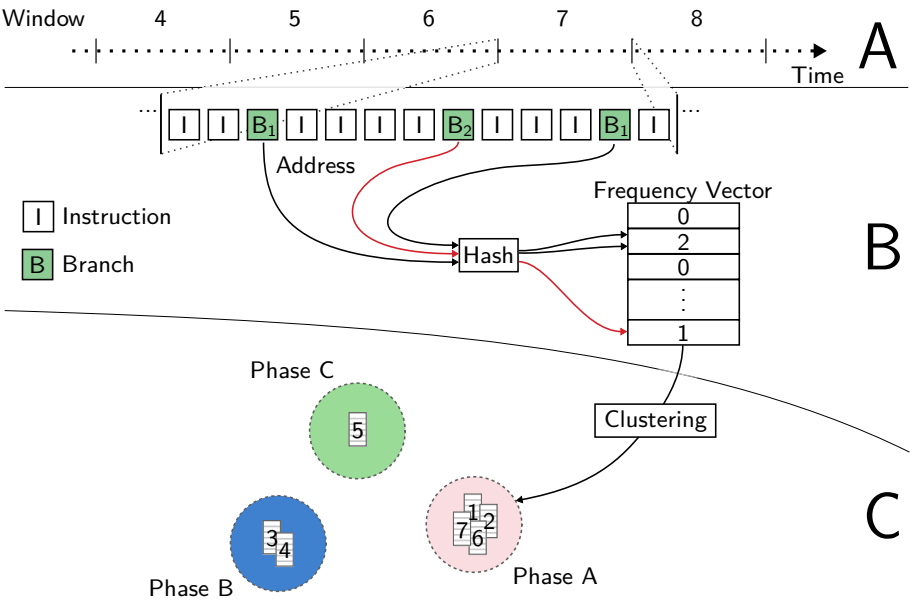


Figure 2.1: The phase detection is done in three steps (A, B, and C). In A, the execution is divided into fixed-sized non-overlapping windows (100M instructions). In B, branch instructions are sampled, where the address of the branch's target (i.e., the head of a basic block) is hashed into a frequency vector that describes how many times a basic block has been executed during one window. Finally, in C, similar frequency vectors are clustered together into phases (i.e., the distance between two vectors are below a threshold). In other words, periods of execution where the code is executed in a similar way (e.g., a loop) are classified as one phase.

C Finally, whenever a window has been executed, that window's frequency vector is clustered together with previous frequency vectors into phases using online clustering [10] (i.e., each cluster corresponds to a runtime phase). For example, if the Manhattan distance between two vectors are below a given threshold, they are classified as belonging to the same phase.

In other words, periods of execution where the code is executed in a similar way (e.g., a loop) are classified as one phase. For example, the vectors belonging to windows 1, 2, 6, and 7 are grouped together into phase A, and window 3, and 4 into phase B, and finally window 5 into phase C.

The colored bars above the graphs in Figure 1.1 show the phases that are detected using this method. We can clearly see that the detected

phase corresponds well to changes in the other metrics. For example, CPI, miss ratio and power change drastically at 10 billion instructions when gcc switches phase from phase A to B. Furthermore, phase B and D, which have similar behavior in CPI but not in miss ratio and power, are correctly detected as two different phases.

2.3 ScarPhase

ScarPhase (*Sample-based Classification and Analysis for Runtime Phases*) is a fast software-based implementation of the above method. ScarPhase is unique in that it is implemented solely in software, has a low overhead, works on real existing hardware, and detects phase at runtime. Other phase detections have been implemented using custom hardware [44, 28] (i.e., only work in simulators) or have been based on offline methods [43, 45, 25, 7, 1, 26, 36] (i.e., first profile the application, then find the phases, and finally, use the phase information) which adds extra overhead, and can not be used by runtime systems (e.g., a scheduler) since the phase detection is not done at runtime.

To keep the overhead at a minimum, ScarPhase uses hardware performance counters to monitor what code is executed (this makes ScarPhase transparent and non-intrusive, with low overhead). One performance counter is used to divide the execution into windows in step A, and another counter is used to sample branches using Intel's Precise Event Based Sampling (PEBS) [30, 18] in step B (i.e., the sampled program counter points to the address of the branch's target which is the head of the next basic block).

There is however, a tradeoff when sampling branches: the more branch samples ScarPhase collects, the better the accuracy, but the higher the overhead. In Paper I, we evaluate this trade off along with other performance improvement techniques (e.g., reducing the cost of collecting a sample), and how to reduce the number of samples that are needed without losing accuracy.

The results show that we can detect runtime phases efficiently with an overhead as low as 1.7%. This makes ScarPhase a very attractive tool to help improve accuracy and performance in a range of different situations since most phase-guided runtime optimizations (see Chapter 4) usually stand to gain more than 1.7%.

2.4 Predicting the Next Phase

The phase of each window is only known after the window has been executed and clustered. However, a runtime decision is sometimes needed

before the window starts to execute (e.g., to know if the application will change phase). To circumvent this limitation, the phase of the next window can be predicted in order to make a runtime decision for the next window. Advanced history predictors [11, 44, 28, 21] have been proposed, where the prediction is based on previously seen behavior (e.g., run-lenth predictor [28]). If the phase pattern has not been seen, it falls back on last-value prediction (i.e., the next window belongs to the same phase as the previous window).

ScarPhase thus includes different phase predictors to enable easy integration with runtime systems. For example, ScarPhase makes a prediction of what phase will be executed next which is then passed along to the runtime system. The best runtime optimization is then applied for the coming (predicted) phase.

2.5 Detecting Phases in Parallel Applications

Parallel applications execute several threads concurrently, where the same set of phases can appear in several threads. To detect phases in parallel applications, ScarPhase must be slightly modified. Step A and B (discussed in Section 2.2) remains unchanged and are performed on all threads independently. The classification step (Step C) must be modified so that the same phase will be identified in all the threads (i.e., share clusters between threads). This means that whenever a thread finishes executing a window, that window is clustered together with windows from other threads.

In Paper II, we use ScarPhase to investigate the phase behavior of parallel applications. The results show that the phase behavior is highly dependent on the parallelization model (e.g., pipeline vs. data-parallel). For example, in pipeline parallel applications, each stage usually executes a unique set of phases (i.e., a phase appear in one stage and not across all threads). This is different from data-parallel application where the same phase usually appears in all threads, but not necessarily at the same time.

2.5.1 Strong Scaling and Phase Detection

The amount of available parallelism in modern processors continues to grow. To investigate how phase detection will work on future machines and what consequences the increasing parallelism will have on phase-guided optimizations, we examine the phases detected with ScarPhase when scaling number of threads (i.e., strong scaling). Figure 3.1 shows the average phase length (i.e., average number of consecutive windows belonging to the same phase) for a data-parallel application (flu-

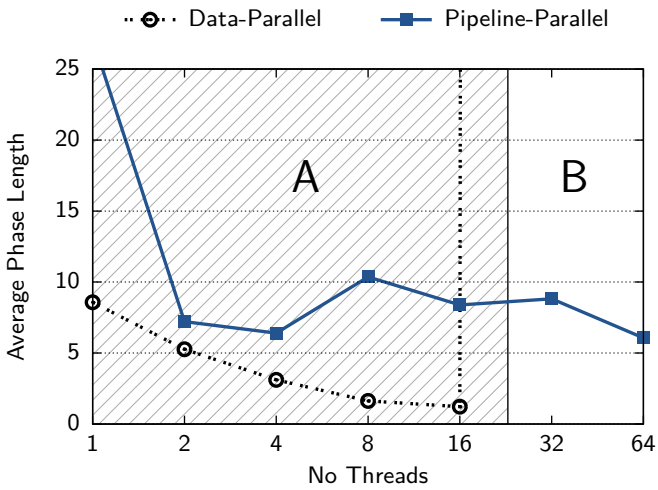


Figure 2.2: Average phase length in number of windows (y-axis) for fluidanimate (Data-Parallel) and dedup (Pipeline-Parallel) for different number of threads (x-axis). *The length of phases shrinks as the number of threads increases in Region A. When the length of the phases shrinks below the window size, the different phases are classified as only one phase in Region B. This means that the runtime behavior appears homegenous.*

idanimate) and a pipeline-parallel application (dedup) from the PARSEC 2.1 [4] benchmark suite. The length of the data-parallel application's phases shrinks with more threads since it divides the work between more threads in Region A. When the length of the phases shrinks below the window size, the different phases are detected as one phase in Region B. This means that the runtime behavior *appears* homegenous. However, this behavior is usually not noticeable for pipeline-parallel applications since each stage executes only one phase (i.e., no/few phase changes in a stage)[2].

In Paper II, we make three very interesting observations about data-parallel applications' phase behavior with regards to phase change frequency, prediction, and homogeneity.

Phase change frequency. Phase changes occur more frequently since the phases shrinks when the work is divided between more threads. Usually there is a cost associated with different runtime optimizations (e.g., power and time to change processor frequency or to migrate a

[2]dedup's the serial version's phase behavior is noticeable different from the parallel version's behavior (see Paper II). We therefore see a significant difference in phase lengths between 1 (serial) to 2 (parallel) threads.

thread). This means that the overhead for doing runtime optimizations will increase with more threads, since the phase transition cost must be paid more frequently.

Prediction. The last value predictor always predicts that an application will not change phase. The prediction accuracy will therefore decrease with more threads since phase changes will occur more frequently. More advanced phase predictors must therefore be used, however, this adds extra complexity and overhead.

Homogeneity. The final observation is that only one phase is detected for more than 32 threads. This means that the runtime behavior *appears* homogeneous across the whole execution. Phase-guided runtime systems will therefore be less attractive with more threads. For example, setting the frequency once at the start of the execution will have the same effect as setting it per phase.

These three observations show that it will be more difficult to exploit phase variation with increasing parallelism. One solution is to also shrink the window size. However, this is not always possible since some optimization have a fixed limit on how fast they can react (i.e., time to change processor frequency). Other possible ways to avoid this problem is to also use weak scaling (i.e., scale data so more works need to be done per thread) or to convert the application to use a different parallelization model (i.e., pipeline-parallel). Regardless, both the number of threads and the speed (window size) must be considered when implementing phase-guided runtime optimizations for parallel applications.

2.6 Summary

This chapter presents ScarPhase, a fast, accurate and general purpose phase detection tool that works on existing commodity hardware. This tool enable us to easily explore and take advantage of applications' runtime phase behaviors.

Chapter 3

Phase Behavior in Serial and Parallel Applications

There are several dynamic runtime optimizations that exploit an application's time-varying behavior (e.g., DVFS). However, most of this research has been based on phase behavior found in serial applications, and in particular the SPEC benchmarks. To investigate if these results are also representative for parallel applications we compare the phase behavior in serial (SPEC) and parallel (PARSEC) applications in Paper II.

The results show that serial applications have significantly more runtime phases and larger performance variations between phases (i.e., the CPI differs more between two phases for serial applications) than parallel applications. Figure 3.1 shows the number of phases that are needed to cover 90% of the execution[1]. Each point corresponds to one benchmark, where the benchmarks have been sorted in descending order with respect to number of phases. This shows that SPEC has significantly more runtime phases than PARSEC. All SPEC benchmarks in the gray region (59% percent of SPEC) have more phases than all the PARSEC benchmarks except raytrace (the leftmost point). On average, the number of detected phases is 8.5 and 3.5 for SPEC and PARSEC respectively.

One observation of this is that it takes a longer time to profile serial applications than parallel applications using phase-guided profiling (see Chapter 4), since the overhead is proportional to the number of detected phases (i.e., one window from each phase is profiled). On average, it will take 2.4× longer to profile and understand the runtime behavior of a

[1]We do not consider 100% because it will include more transition-phases [28], 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 misleading so we ignore them in this analysis.

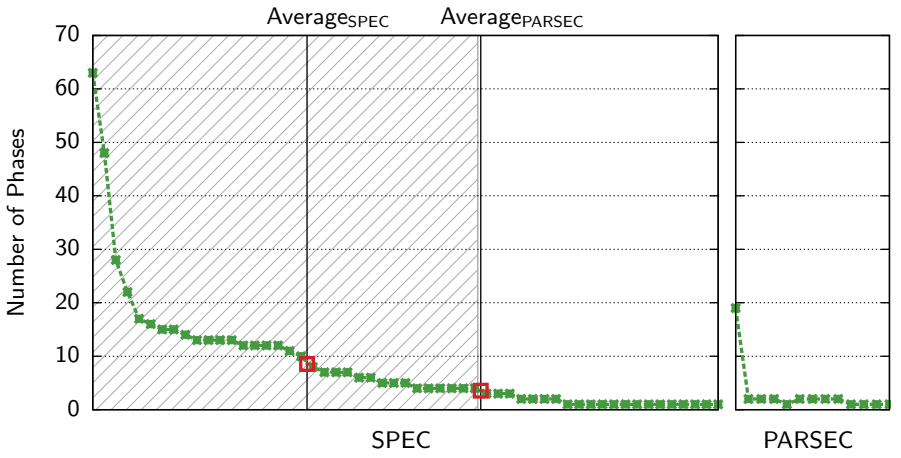


Figure 3.1: The number of phases that are needed to cover 90% of the program execution for serial applications (left, SPEC) and parallel applications (right, PARSEC). The gray area highlights the difference between the benchmarks suites. All SPEC programs in the gray region have more phases than all the PARSEC program except raytrace (the leftmost point). *This shows that SPEC has more phases (*2.4× *on average). One consequence of this is that it will take a longer time to profile an serial application than an parallel application with phase-guided profiling (i.e., profile each phase once), since the overhead is proportional to the number of phases. On average it will take* 2.4× *longer to profile an SPEC application than an PARSEC application.*

SPEC benchmark than a PARSEC benchmark.

The overall results in Paper II show that phase-guided optimizations have less potential with parallel applications because there are fewer phases, and it becomes harder with more parallelism since the phases shrinks. Nonetheless, parallel applications have phases, and so, phase detection will still remain a key component for improving performance of various methods that operates on parallel applications.

Part II

Exploiting Phase Detection

Chapter 4

Phase-Guided Runtime Optimizations

The previous part of this thesis described how to detect phases. While phase detection provides good insight into an application's runtime behavior, what is more important is how runtime phase detection can be used to improve applications' execution and to mitigate the cost of various runtime optimizations. In this chaper, we explore three use cases (papers III, IV and V) where we combine ScarPhase with other methods to improve accuracy and performance, and to provide new insights into applications' behaviors.

Before examining these use cases, we will first give an overview of how phase detection can be exploited. The final goal is to improve different methods' performance or accuary by taking advantage of applications' heterogeneous runtime behavior. Phase-guided runtime optimizations 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 4.1. The figure shows the CPI, L2 cache miss ratio, and power consumption (y-axis), over time (x-axis). The dashed red line shows the average behavior, and the colored bars above the graphs shows the detected runtime phases, with smaller phases grouped together 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, and, the white triangles show where phase-guided profiling (discussed below) decides to profile the application.

Example: Phase-Guided DVFS and Cache Resizing

Phase-guided dynamic voltage frequency scaling and dynamic cache resizing [9, 44, 42, 27, 21, 33] are used to reduce the processor's energy

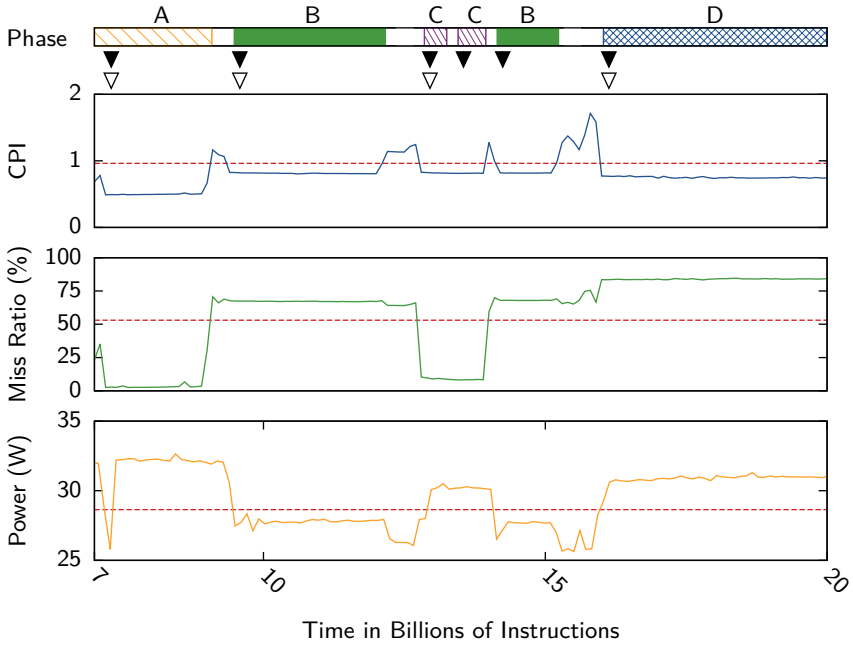


Figure 4.1: CPI, L2 cache miss ratio, and power consumption (y-axis), over time (x-axis). The dashed red line shows the average behavior, and the colored bars above the graphs shows the detected runtime phases, with smaller phases grouped together in white for clarity. The black triangles shows when the phase detector notices a new phase and when the runtime system can apply the best optimization for the new phase.

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, for dynamic cache resizing, when gcc changes phase from A to B in Figure 4.1, the L2 cache size can be increased to reduce the larger miss ratio. For DVFS, the CPU frequency can be raised, since gcc enters a low power phase that is under the power limit. This can be seen in the figure by the increase in cache miss ratio and the reduction in power.

Example: Phase-Guided Scheduling

Phase-guided scheduling [46] is used to exploit heterogeneous multiprocessors to improve throughput. When an application enters a memory-bound phase, it can be migrated to a slower core/chip, until it returns to a compute-intensive phase. For example, when gcc changes phase from B to D, it becomes more compute-bound (lower CPI), and should therefore be migrated back to a faster core.

Example: Phase-Guided Profiling

Phase-guided profiling [34, 40] is a method to reduce the overhead of profiling without sacrificing accuracy, by taking advantage of the (nominally) uniform behavior within each 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 4.1 shows where phase-guided profiling decides to profile the application. 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 can therefore reuse the previous profile of phase B. The overhead is thus proportional to number of phases, and not the length of the application.

4.1 Phase-Guided Profiling

The three use cases all uses ScarPhase to provide support for phase-guided profiling, albeit in slightly different ways. We will therefore take a closer look into how phase detection can be leveraged to improve different aspects of profiling.

The underlying goal is to understand an application's runtime behavior. This can be done by profiling every single window. However, this is usually not feasible in practice, since profiling is expensive, and the overhead is therefore usually too high.

An commonly used alternative to phase-guided profiling is periodic profiling (i.e., periodically select windows to profile). For example, every 10th window can be selected. The overhead would then be 10× lower than profiling every window. However, there are several drawbacks to this approach that phase-guided profiling avoids. By using phase-guided profiline, we get: better performance, better accuracy, and better understanding.

Better Performance

Phase-guided profiling can be much faster than periodic profiling since it avoids redundant profiling. For example, if a phase has already been profiled, it does not need to be profiled again. Periodic profiling is unaware of the application's phases and may therefore profile several windows from the same phase.

This can be a problem when we want to know an application's runtime behavior in detail (i.e., the behavior in all phases). To catch the smallest phases, the sample period must be conservative and short enough to match the granularity with which the phases occur. For

example, if we want to know gcc's behavior from 13 to 14 billion instructions (i.e., phase C) in Figure 4.1, a short period is needed. This will catch the behavior in phase C, however, several windows will then needlessly be reprofiled in phase D.

Better Accuracy

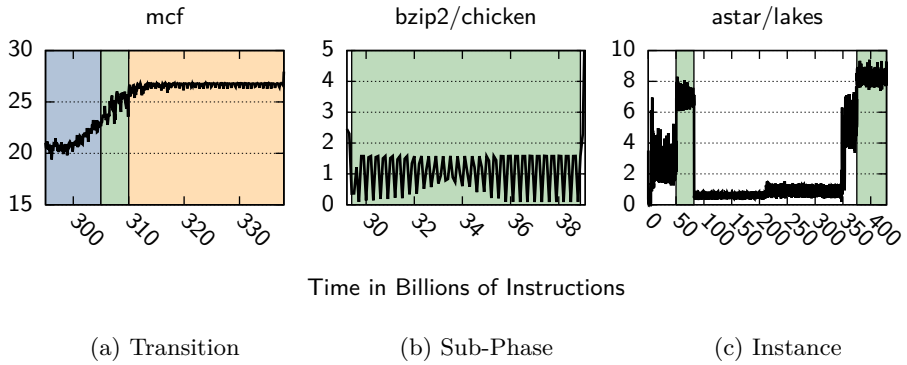
Periodic profiling also introduces gaps in our understanding of an application's runtime behavior between two profiled windows. That is, we only know the behavior in the windows we profile. One solution is to use interpolation to estimate the behavior in the non-profiled windows. However, this can be misleading since there can be several phases in between the profiled windows. Phase-guided profiling knows instead which phases are executed and can directly infer the behavior by examining the behavior of the phase each window belongs too.

Better Understanding

Both phase-guided profiling and periodic-profiling enable us to view an application's behavior over time with a lower overhead. However, applications can execute several thousand windows during one execution with repeating behavior. For example, gcc/166 (see Figure 1.1) is a short running application but it still executes many phases that reoccur several times. This makes it very difficult and inefficient to reason about individual or ranges of windows (e.g., windows $94-121$, $141-152$, ...). Instead, using phase-guided profiling we can say phase B, which is much easier.

Furthermore, the way ScarPhase detects phases enable us to easily map phases back to the source code since we already sample what basic blocks are executed. A programmer can then find problematic runtime phases (e.g., high CPI), and optimize the code that belongs to those phases.

In Paper III, we compare the accuracy and performance of phase-guided profiling against periodic profiling when applied to memory-access profiling for cache modeling. The results show that phase-guided profiling is $6\times$ faster and 39% more accurate than periodic profiling. This makes it possible to investigate applications with new methods that would otherwise have not been practical due to high overhead.



(a) Transition (b) Sub-Phase (c) Instance

Figure 4.2: Intra-phase variation in miss ratio for SPEC2006. Phases shown with shading. *While most phases show little intra-phase variation, as in gcc/166, some exhibit different types of intra-phase variation. Both Transition (Figure 4.2a) and Sub-Phase (Figure 4.2b) are artifacts of the tradeoff between phase size and the number of phases. Instance variations (Figure 4.2c), however, represent data-dependent changes in behavior for the same code path.*

4.2 Intra-Phase Variations

While the phases detected by ScarPhase have reasonably constant behavior within each phase, some applications exhibit intra-phase variation [43, 42]. In Paper III, we observe three different types of intra-phase variation: transition, sub-phase, and instance. These are illustrated in Figure 4.2. Both transition (Figure 4.2a) and periodic sub-phase (Figure 4.2b)[1] variations are artifacts of the tradeoff between phase size and the number of phases. Instance variations (Figure 4.2c), however, represent data-dependent changes in behavior for the same code path.

To limit the negative effects of intra-phase variations on the accuracy, instead of profiling a single window from each phase, we profile several windows randomly selected from within a phase. The average of those windows is then used to characterize the phase's runtime behavior. This slightly increases the overhead, but is still better than periodic profiling in terms of both performance and accuracy (see Paper III).

[1]Refereed to as just periodic in Paper III.

4.3 Phase-guided Multiplexing

Hardware performance counters are valuable tools to collect low-level performance runtime data (e.g., cache misses) with low overhead. However, there is a limited number of performance counters. For example, there are only 4 general purpose performance counters on Intel Nehalem (our evaluation system). This presents a problem when we want to track more than four performance metrics.

The standard approach to this limitation is to use a form of periodic profiling. The performance counters are turned on and off at regular intervals in a round robin schedule so that only 4 counters are active at the same time. For example, count cache misses in windows $1, 3, 5$ and then count something else in windows $2, 4, 6$ and so on.

This makes it possible to artificially use more performance counters than what is available, but at a cost in accuracy. In Paper V, we needed to use more than 4 performance counters. To improve the accuracy, we instead used ScarPhase to do phase-guided multiplexing of the performance counters. The same schema as above was used but at a per phase basis. That is, each phase has its own schedule on how to turn counters on and off. This allow us to easily use more performance counters than available without significantly affecting the accuracy.

4.4 Summary

The discussion in this chapter highlights the importance of detecting phases (e.g., ScarPhase). This enable us to improve other techniques by adapting to changes in runtime behavior. In addition, phase-guided profiling allow us to explore an application's behavior at a finer granularity that would otherwise have been impractical (i.e., too high overhead).

Chapter 5

Use Cases

We have seen that understanding an application's runtime phase behavior is important. However, phase information by itself is not very useful. More information is needed (e.g., cache and power). For such information to be usable in practice (e.g., for a developer), the information must be easy to acquire and accurate. We therefore combine ScarPhase with mathematical models to provide new insights into an application's cache performance, cache sharing sensitivity, and power behavior on a per-phase basis.

Figure 5.1 shows an overview of the use cases' workflows and how ScarPhase and phase information is used. The application is first profiled once in Box 1. ScarPhase is used here to improve the use case specific profilers' performance and accuracy. In the second step (Box 2), a model (e.g., a power model) is combined with phase information from ScarPhase to generate per-phase information. Box 2 is then repeated called for all inputs (e.g., available CPU frequencies).

5.1 Use Case A: Cache Performance

The goal of this use case is to develop an efficient method for understanding an application's cache behavior over time and as a function of its cache allocation. This type of information can be used to better understand performance [29], resource sharing [12, 6], and scheduling [13].

The ability to analyze an application's runtime behavior as a function of its cache allocation is particulary important for modern systems with shared caches where the cache allocation can change dynamically. This requirement makes it difficult to use data from hardware performance counters, as they only provide information for one particular cache allocation.

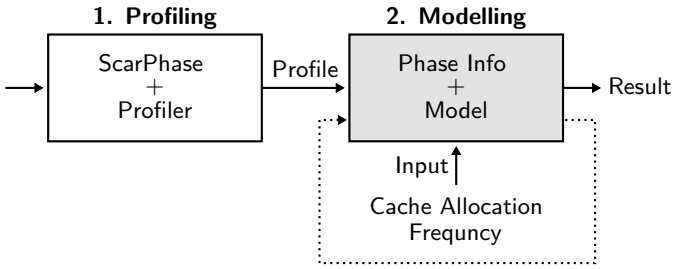


Figure 5.1: Overview of the use cases' workflows. First, the application is profiled once in Box 1. Then, a model is repeatedly called with different inputs (e.g., the target frequency) in Box 2 to generate per-phase per-frequency performance information.

We instead use StatCache [2, 3], a low overhead statistical cache model that can estimate an application's miss ratio for caches of arbitrary sizes. It answers the question: what is an application's miss ratio if it receives x amount of cache? The input to the model is an application profile containing sampled information about the application's memory accesses, which is then passed through the model to calculate the miss ratio.

However, StatCache only outputs an application's average miss ratio, which can be misleading. We therefore extend the model by combing it with ScarPhase so that the model is applied to each phase, instead of the average, by using phase-guided profiling. Using these two methods, we can accurately model application cache behavior as a function of time and allocated cache[1].

Figure 5.2 shows an example of what the new method can provide. It shows the miss ratio (intensity) over time (x-axis) as a function of cache allocation (y-axis), for the complete execution of gcc/166. The darker the points, the higher the miss ratio. The y-axis (cache size) indicates how the application's miss ratio is affected by its cache allocation. The vertical bar marked Average on the right shows the application's overall average miss ratio as a function of cache allocation. And, finally, the bars above the graph shows the phases detected by ScarPhase, with smaller phases grouped together in white for clarity.

The figure demonstrates that not only do we need to consider an application's behavior over time, but, we must also consider how much cache the application receives. For example, both phase A and B have similar miss ratio when they are allocated more than 1 MB cache, but very different behavior with less than 1 MB. This means that it is more

[1] In [23], we use a similar approach but for instruction caches.

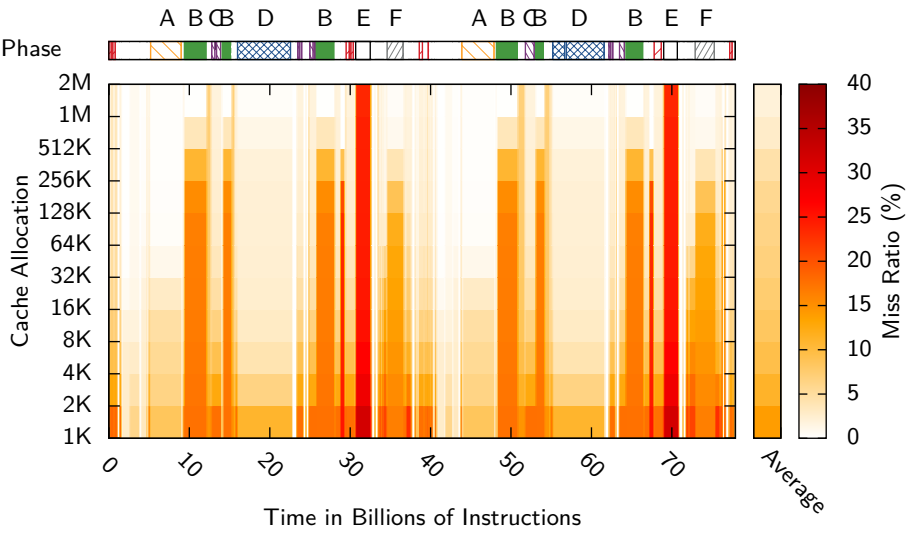


Figure 5.2: Miss ratio (intensity) as a function of time (x-axis) and cache allocation (y-axis) for the whole execution of gcc/166. The average miss ratio for the whole execution is shown on the right. The detected runtime phases are shown above, with shorter phases shown in white for clarity.

important to optimize phase B's cache performance if gcc is running on a machine with less than 1 MB cache, or if it is co-scheduled with a cache-hungry application.

The results in Paper III show that the new method using phase-guided profiling with ScarPhase is 39% more accurate and 6× faster than previous modeling techniques. This enable us to quickly provide valuable new insights into applications' cache performance.

5.2 Use Case B: Cache Sharing Effects

When several applications are co-executed together, they compete for shared resources (e.g., last level cache). Since different phases have different behavior, the performance will vary depending on how the applications' phases overlap with each over. The goal of this use case is to develop a fast method for understanding performance variations due to cache sharing as a result of how phases overlap.

Previous research has mostly focused on an application mix's average slowdown which can be misleading [50, 51, 32, 31]. Figure 5.3 shows gcc/166's slowdown distribution when gcc is co-executed with bwaves 100 times with different offsets in starting times. The average slowdown

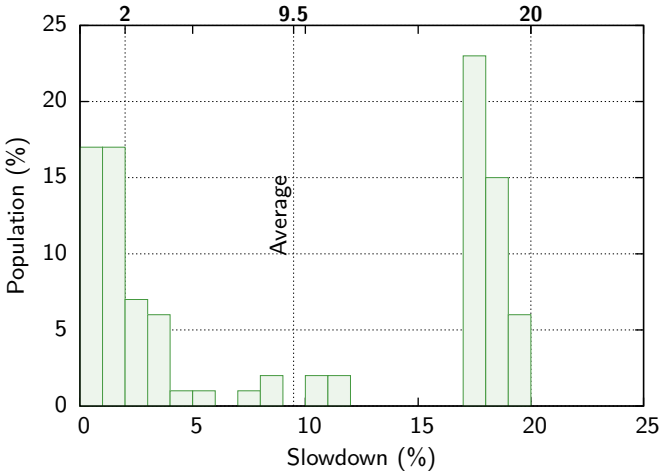


Figure 5.3: Performance distribution for gcc/166 co-running together with bwaves on an Intel Xeon E5620 based system. *Ignoring performance variability can be misleading, since the average (*9.5%*), hides the fact that the performance can vary between* 1% *and* 20% *depending on how the two applications' phases overlap.*

is 9.5%, however, it varies between 1% to 20% depending on how gcc's and bwaves's phases overlap. Furthermore, a developer that wants to evaluate gcc's performance when co-executed with bwaves might draw an incorrect conclusion when running the application for only a few times since there is a 34% probability that the slowdown will be less than 2% when in reality the slowdown can actually be as high as 20%.

To accurately estimate the performance of a mixed workload, the application mix must be run many times. This is both a time and resource-expensive process. In order to access the performance variability more quickly, we instead use a cache sharing model [37]. This cache model can predict an application's cache usage, CPI, and bandwidth when it is co-executed with other applications. However, the model can only handle phase-less applications and can as a result only output average performance slowdowns. We therefore extend the model to handle phases and we combine it with ScarPhase to generate performance distributions instead of average slowdowns.

The input to the cache sharing model is a set of application profiles containing information about how the miss rate and hit rate changes depending on the application's cache allocation. Unfortunately, the method in the previous use case described in Section 5.1 produces only miss-ratios (i.e., misses per memory access) and not miss-rates (i.e.,

misses per cycle). We therefore use another method, Cache Pirating [12], to collect the required input data. Cache Pirating uses hardware performance counters to measure an application's cache miss rate, CPI, etc, at runtime as a function of cache allocation. This is done by co-running a cache-intensive stress benchmark that steals a specified amount of shared cache.

As in the previous use case, the cache model is applied per phase. However, there is an even greater benefit with phase detection in this context: when evaluating the slowdown of several runs with different offsets in starting times, the same set of phase pairs will usually appear in several runs. For example, gcc's phase A will be executed together with bwaves's phase B in both run 1 and run 5. The cache sharing results can thus be reused not just in one run, but also across several runs.

The results in Paper IV show that we can evaluate an application mix's performance variations fast and accurately. It is on average 213× faster to use the new method than to run the target application the same number of times natively. This provide us with a method to quickly evaluate an application performance in a more genuine context, since applications are rarely run in isolation.

5.3 Use Case C: Processor Frequency

The previous two use cases investigated an application's performance when run on a processor that does not change behavior during runtime. However, modern processors are dynamic, for example, they can change voltage and frequency to conserve energy (e.g., DVFS). Considering only one frequency can therefore be misleading since changing the processor's clock frequency will also affect the application's runtime behavior. The goal of this use case is to provide developers with a method and a tool (Power-Sleuth) for understanding an application's performance and power behavior over time as a function of processor frequency.

Figure 5.4 shows an example of Power-Sleuth's output. It shows the power consumption (i.e., $Energy\,(\mathrm{J}) = Power\,(\mathrm{W}) * Performance\,(\mathrm{s})$) in one execution window (intensity) over time (x-axis) as a function of processor frequency (y-axis), for the complete execution of the gcc/166. The darker the points, the higher the power consumption. The y-axis (CPU Frequency) indicates how the application's power consumption is affected by the processor's frequency. The vertical bar marked Average on the right shows the application's overall average power consumption as a function of processor frequency. And, finally, the bars above the graph shows the phases detected by ScarPhase, with smaller phases

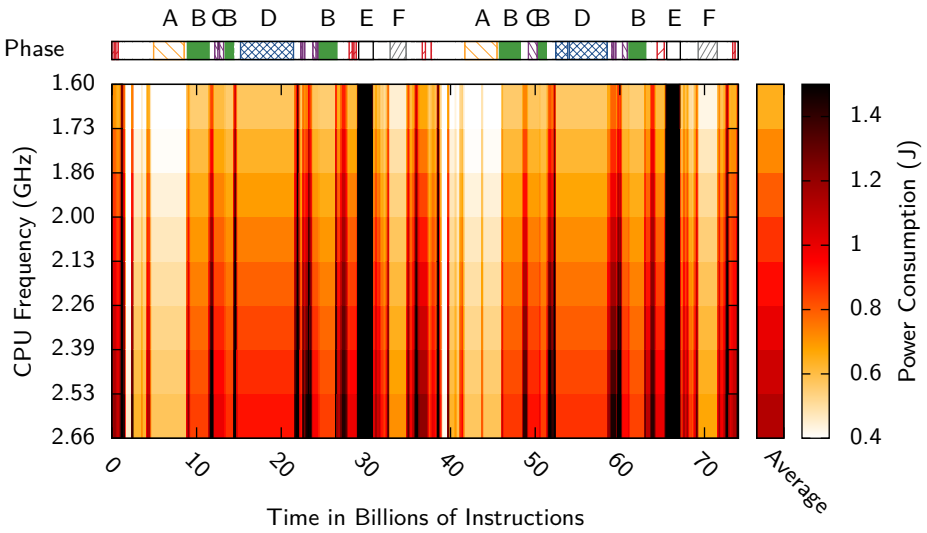


Figure 5.4: Energy consumed in one window (intensity) as a function of time (x-axis) and processor frequency (y-axis) for the whole execution of gcc/166. The average energy consumption for the whole execution is shown on the right. The detected runtime phases are shown above, with shorter phases shown in white for clarity. Phase E consumes much more energy ($1.8 - 5$J) than the other phases and has been capped for visibility.

grouped together in white for clarity.

The power consumption metric can vary a lot between phases. For example, Phase E has a much higher power consumption (1.8J and 5J at 1.6 GHz and 2.66 GHz respectively) than the other phases. To improve the visibility, we have capped phase E to increase the contrast between the other phases.

The figure shows that an application's energy behavior changes both over time and as a function of frequency. Traditional tools (e.g., Intel VTune [19]) have only focused on one frequency which can be a problem. For example, a developer that wants to reduce the total energy consumption, might start by finding the phase that consumes most energy (i.e., the sum of all windows belonging to the same phase), and try to optimize that phase's code. However, the most energy hungry phase can be different depending on frequency. For example, phase E consumes more energy at 2.66 GHz (167J) than phase D (118J) but less energy at 1.6 GHz (62J vs. 71J). This problem is not limited to just energy but can also occur for example when optimizing for performance (i.e., the longest running phase is different depending on frequency). This means

that a developer must look at all possible frequencies or the most likely runtime frequency, to find the right parts of the code to optimize.

To provide this kind of information, Power-Sleuth combines ScarPhase with a performance model [48] and a power model [22]. As before, ScarPhase is used to apply the models per phase. Furthermore, both of these models use hardware performance counters to collect input data. A problem here is that we need to use more performance counters than are available on modern hardware. To circumvent this limitation, ScarPhase is also used to do phase-guided multiplexing (see Section 4.3) of the performance counters so that each counter is collected at least once per phase.

This enable us to provide detailed power and performance analysis per phase for any given processor frequency. The results in Paper V show that we can do this fast and accurately, with an average errors of 3.5% and 3.9% for power and performance respectively.

5.4 Summary

These three use cases demonstrate that the average runtime behavior is not enough. To fully understand a modern application we must consider it's phase behavior. By combining phase detection with different model-based methods, we can provide new insights with respect to; cache allocation, co-executing applications and processor frequency. This enable us to quickly and accurately find problematic phases that can be targeted for optimization, but also to provide valuable information for decision-making in runtime systems.

Part III

Related Work and Conclusions

Chapter 6

Related Work

In this chapter, we discuss research related to phase detection and phases in general. We limit the discussion to just phases-related works since the scope of the use cases' related work is too broad to be summarized in this chapter. That information can instead be viewed in each of the individual papers (see Papers III, IV, and V).

Figure 6.1 presents an overview of different types of phase detection schemes. In broad terms, phase detection can be categorized along two dimensions: whether the detection is implemented in hardware or in software (x-axis); and whether the detection is done by analyzing a runtime metric or by analyzing the code's control flow (y-axis).

6.1 Runtime Analysis

Runtime-based phase detection (Q_1 and Q_2 in Figure 6.1) works by monitoring temporal changes in performance metrics. This can either be done in hardware (Q_1) or in software (Q_2, e.g., ScarPhase).

Q_1 (Hardware) implemented the method described in Chapter 2 with some extra support implemented in hardware. The advantage of using extra hardware is that every branch (i.e., basic block) can be sampled for better accuracy without impacting the application's performance. However, the requirement of extra hardware functionality puts severe restrictions on its usability.

In Q_{2a} (Software), the detection is done in software, for example with binary instrumentation (e.g., Pin [5]). Unfortunately, this introduces a significant overhead. To reduce the overhead, hardware performance counters are used in Q_{2c} for sparse sampling. However, both Q_{2a} and Q_{2c} use offline clustering to find phases. This means that the application is first profiled to find the runtime phases, then the application is run again with the collected phase information. This makes it impractical

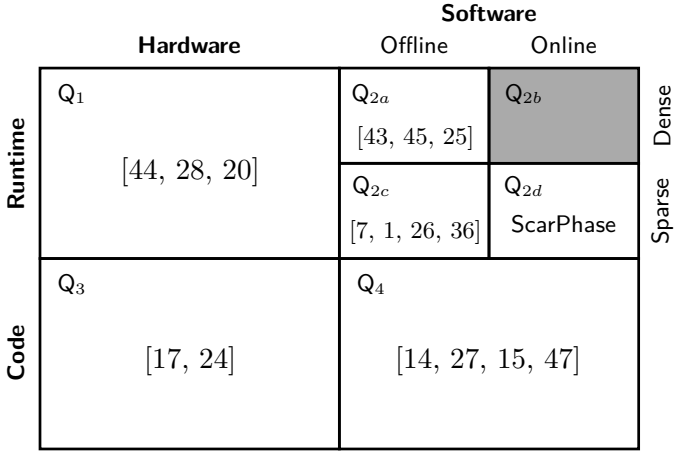


Figure 6.1: Overview of phase detection methods.

for many online uses (e.g., DVFS or cache resizing). However, the goal in Q_{2a} and Q_{2c} is to reduce the overhead of architecture simulation which is an order of magnitude more expansive than the cost of finding the application's phases.

ScarPhase combines the insights from online hardware phase detection in Q_1, with the hardware performance counter approach in Q_{2c} to detect phases at runtime with low overhead using software. This makes it possible to use it for phase-guided runtime optimizations on commodity hardware. Furthermore, online phase detection is needed for phase-guided profiling.

6.2 Code Analysis

Code-analysis-based phase detection (Q_3 and Q_4) works by analyzing an application's binary or its control flow graph. For example, if a function is large enough it is considered a phase. This can also be done in both hardware (Q_3) and software (Q_4).

Q_4 (Software) usually combines code analysis with profiling [27] to find which parts are executed the most. Each part is then considered a phase. Afterwords, extra runtime code is inserted into the binary at the phase boundaries to enable runtime decisions (i.e., they get triggered when the application changes phase). This makes the runtime phase detection very efficient since the only overhead comes from the inserted code. However, the application must first be profiled in order to find the phases which makes it input dependent (i.e., phase detection might

not work if the input is to different to the traning set). Furthermore, this adds extra overhead and it complicates the development cycle.

Another option is to do the profiling at runtime as well. This works well for interpreters (e.g., Python) and virtual machines (e.g., Java) [14], where it is easy to add this functionally since some form of profiling is usually done anyway during runtime (e.g., count function invocations to find hotspots for further optimizations). However, for natively compiled code (e.g., C/C++), this may add extra overhead, but also extra complexity since it can be difficult to do binary rewriting at runtime. Furthermore, there may also be a delay until all the phases are discovered during which phase-guided profiling is turned off.

Q_3 (Hardware) uses custom hardware to analyze an application's control flow and to detect phases at runtime. For example, a phase change occurs when the depth of the call stack exceeds a given threshold [24]. As before, this requires extra hardware which makes it less useful in many situations.

6.3 Phase Change Detection

A similar but less difficult problem is phase change detection and predicting performance changes [8, 11, 21, 35]. Peleg and Mendelson [35] showed that changes in the CPI can not be used as a metric for loaded systems. Instead, basic block vectors (BBV) and other architecture independent metrics have been used to detect performance behavioral changes. This is different from phase detection (e.g., ScarPhase) in that only the last windows are of interest (i.e., the applications changes phase if the difference between the last two windows is above a threshold). However, remembering reoccurring phases can improve performance by reusing configuration settings [11, 21].

6.4 Summary

As discussed, there are many different ways to detect runtime phases. Any one of these methods could have been used in the use cases. However, while the approach we have taken may only occupy a small part of Figure 6.1, it does provide some unique qualities (e.g., fast, software-only and online) and meets all the requirements in Chapter 2. This makes it possible to seamlessly integrate ScarPhase with other tools and methods.

Chapter 7

Conclusions

This thesis first presented ScarPhase, a new online method for detecting runtime phases with low overhead (2%). ScarPhase was then used throughout the thesis to provide phase information in various studies and use cases. For example, we used it to investigate the runtime phase behavior in serial and parallel applications and how the behavior differs. This provided us with new insights and a tool for exploiting runtime phase behavior, which we leveraged in three use cases to create new methods for exploring an application's cache performance, cache sharing effects, and power consumption along different dimensions (cache allocation and processor frequency).

In the use cases, we introduced new tools and methods that enable us, and other researchers and developers, to untangle the complexities of modern computer systems. We hope that the findings in this thesis will enable the development of faster and more power-efficient applications and runtime systems that can better solve tomorrow's problems.

Chapter 8

Bibliography

[1] Murali Annavaram et al. “The Fuzzy Correlation between Code and Performance Predictability”. In: *Int. Symposium on Microarchitecture (MICRO)*. 2004.

[2] E. Berg and E. Hagersten. “StatCache: a probabilistic approach to efficient and accurate data locality analysis”. In: *Int. Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2004.

[3] Erik Berg and Erik Hagersten. “Fast data-locality profiling of native execution”. In: *Int. Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 2005.

[4] Christian Bienia et al. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”. In: *Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008.

[5] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *Conference on Programming Language Design and Implementation (PLDI)*. 2005.

[6] Dhruba Chandra et al. “Predicting inter-thread cache contention on a chip multi-processor architecture”. In: *Int. Symposium on High-Performance Computer Architecture (HPCA)*. 2005.

[7] B. Davies et al. *iPART : An Automated Phase Analysis and Recognition Tool*. Tech. rep. IR-TR-2004-1-iPART. Intel Corporation, 2004.

[8] Ashutosh S. Dhodapkar and James E. Smith. “Comparing Program Phase Detection Techniques”. In: *Int. Symposium on Microarchitecture (MICRO)*. 2003.

[9] Ashutosh S. Dhodapkar and James E. Smith. “Managing multi-configuration hardware via dynamic working set analysis”. In: *Int. Symposium on Computer Architecture (ISCA)*. 2002.

[10] Richard O. Duda, Peter E. Hart, and David G. Stork. “Pattern Classification”. In: 2nd ed. Wiley-Interscience, 2001. Chap. 10.11. On-line Clustering, pp. 559–565. ISBN: 0-471-05669-3.

[11] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. “Characterizing and Predicting Program Behavior and its Variability”. In: *Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2003.

[12] David Eklov, David Black-Schaffer, and Erik Hagersten. “Fast modeling of shared caches in multicore systems”. In: *Int. Conference on High Performance and Embedded Architecture and Compilation (HiPEAC)*. 2011.

[13] Alexandra Fedorova et al. “Performance of multithreaded chip multiprocessors and implications for operating system design”. In: *USENIX Annual Technical Conference (USENIX)*. 2005.

[14] Andy Georges et al. “Method-level phase behavior in java workloads”. In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2004.

[15] Dayong Gu and Clark Verbrugge. “Phase-based adaptive recompilation in a JVM”. In: *Int. Symposium on Code Generation and Optimization (CGO)*. 2008.

[16] John L. Henning. “SPEC CPU2006 benchmark descriptions”. In: *SIGARCH Comput. Archit. News* (2006).

[17] Michael C. Huang, Jose Renau, and Josep Torrellas. “Positional adaptation of processors: application to energy reduction”. In: *Int. Symposium on Computer Architecture (ISCA)*. 2003.

[18] *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Volume 3B: System Programming Guide. 30.4.4 Precise Event Based Sampling (PEBS). Intel Corporation. 2010.

[19] *Intel VTune Performance Analyzer Homepage*. URL: `http://www.intel.com/software/products/vtune/.`.

[20] C. Isci and M. Martonosi. “Detecting recurrent phase behavior under real-system variability”. In: *Int. Symposium on Workload Characterization (IISWC)*. 2005.

[21] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. “Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management”. In: *Int. Symposium on Microarchitecture (MICRO)*. 2006.

[22] Stefanos Kaxiras and Margaret Martonosi. “Computer Architecture Techniques for Power-Efficiency”. In: Morgan and Claypool Publishers, 2008. ISBN: 0-471-05669-3.

[23] Muneeb Khan, Andreas Sembrant, and Erik Hagersten. “Low Overhead Instruction-Cache Modeling Using Instruction Reuse Profiles”. In: *Int. Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2012.

[24] Jinpyo Kim et al. “Dynamic Code Region (DCR)-based Program Phase Tracking and Prediction for Dynamic Optimizations”. In: *Int. Conference on High Performance and Embedded Architecture and Compilation (HiPEAC)*. 2005.

[25] J. Lau, S. Schoemackers, and B. Calder. “Structures for phase classification”. In: *Int. Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2004.

[26] J. Lau et al. “The Strong correlation Between Code Signatures and Performance”. In: *Int. Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2005.

[27] Jeremy Lau, Erez Perelman, and Brad Calder. “Selecting Software Phase Markers with Code Structure Analysis”. In: *Int. Symposium on Code Generation and Optimization (CGO)*. 2006.

[28] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. “Transition Phase Classification and Prediction”. In: *Int. Symposium on High-Performance Computer Architecture (HPCA)*. 2005.

[29] Alvin R. Lebeck and David A. Wood. “Cache Profiling and the SPEC Benchmarks: A Case Study”. In: *Computer* (1994).

[30] David Levinthal. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*. Tech. rep. Version 1.0. Intel Corporation, 2009.

[31] Jason Mars, Lingjia Tang, and Mary Lou Soffa. “Directly Characterizing Cross Core Interference Through Contention Synthesis”. In: *Int. Conference on High Performance and Embedded Architecture and Compilation (HiPEAC)*. 2011.

[32] Jason Mars et al. “Contention Aware Execution”. In: *Int. Symposium on Code Generation and Optimization (CGO)*. 2010.

[33] Ke Meng et al. “Multi-optimization power management for chip multiprocessors”. In: *Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008.

[34] Priya Nagpurkar, Chandra Krintz, and Timothy Sherwood. “Phase-Aware Remote Profiling”. In: *Int. Symposium on Code Generation and Optimization (CGO)*. 2005.

[35] Nitzan Peleg and Bilha Mendelson. “Detecting Change in Program Behavior for Adaptive Optimization”. In: *Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2007.

[36] Erez Perelman et al. “Detecting phases in parallel applications on shared memory architectures”. In: *Int. Parallel and Distributed Processing Symposium (IPDPS)*. 2006.

[37] A. Sandberg, D. Black-Schaffer, and E. Hagersten. “Efficient Techniques for Predicting Cache Sharing and Throughput”. In: *Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2012.

[38] Andreas Sandberg et al. “Modeling Performance Variation Due to Cache Sharing”. In: *Int. Symposium on High-Performance Computer Architecture (HPCA)*. 2013.

[39] Andreas Sembrant, David Black-Schaffer, and Erik Hagersten. “Phase Behavior in Serial and Parallel Applications”. In: *Int. Symposium on Workload Characterization (IISWC)*. 2012.

[40] Andreas Sembrant, David Black-Schaffer, and Erik Hagersten. “Phase Guided Profiling for Fast Cache Modeling”. In: *Int. Symposium on Code Generation and Optimization (CGO)*. 2012.

[41] Andreas Sembrant, David Eklov, and Erik Hagersten. “Efficient Software-based Online Phase Classification”. In: *Int. Symposium on Workload Characterization (IISWC)*. 2011.

[42] Xipeng Shen, Yutao Zhong, and Chen Ding. “Locality phase prediction”. In: *Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2004.

[43] T. Sherwood, E. Perelman, and B. Calder. “Basic block distribution analysis to find periodic behavior and simulation points in applications”. In: *Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2001.

[44] Timothy Sherwood, Suleyman Sair, and Brad Calder. “Phase tracking and prediction”. In: *Int. Symposium on Computer Architecture (ISCA)*. 2003.

[45] Timothy Sherwood et al. “Automatically characterizing large scale program behavior”. In: *Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2002.

[46] Tyler Sondag and Hridesh Rajan. “Phase-based tuning for better utilization of performance-asymmetric multicore processors.” In: *Int. Symposium on Code Generation and Optimization (CGO)*. 2011.

[47] Tyler Sondag and Hridesh Rajan. “Phase-guided thread-to-core assignment for improved utilization of performance-asymmetric multi-core processors”. In: *ICSE Workshop on Multicore Software Engineering*. 2009.

[48] Vasileios Spiliopoulos, Stefanos Kaxiras, and Georgios Keramidas. “Green governors: A framework for Continuously Adaptive DVFS”. In: *Int. Green Computing Conference (IGCC)*. 2011.

[49] Vasileios Spiliopoulos, Andreas Sembrant, and Stefanos Kaxiras. “Power-Sleuth: A Tool for Investigating your Program’s Power Behavior”. In: *Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2012.

[50] David Tam et al. “Managing Shared L2 Caches on Multicore Systems in Software”. In: *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*. 2007.

[51] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. “Addressing Shared Resource Contention in Multicore Processors via Scheduling”. In: *Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2010.

Part IV

Papers

Paper I

Paper I

Efficient Software-based Online Phase Classification

Andreas Sembrant, David Eklöv, and Erik Hagersten

In Proceeding of the
International Symposium on Workload Characterization
Austin, Texas, USA, November 2011

IISWC November 2011, Austin, Texas, USA 978-1-4577-2064-2/11/$26.00

Efficient Software-based Online Phase Classification

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

Abstract

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

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

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

I.1 Introduction

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

To capture such time varying behaviors, researchers have proposed detecting *program phases*. A program phase is defined to be a period of execution with a stable behavior. There are several important optimizations, both offline [2, 10, 27, 31, 32] and online [6, 12, 16, 24],

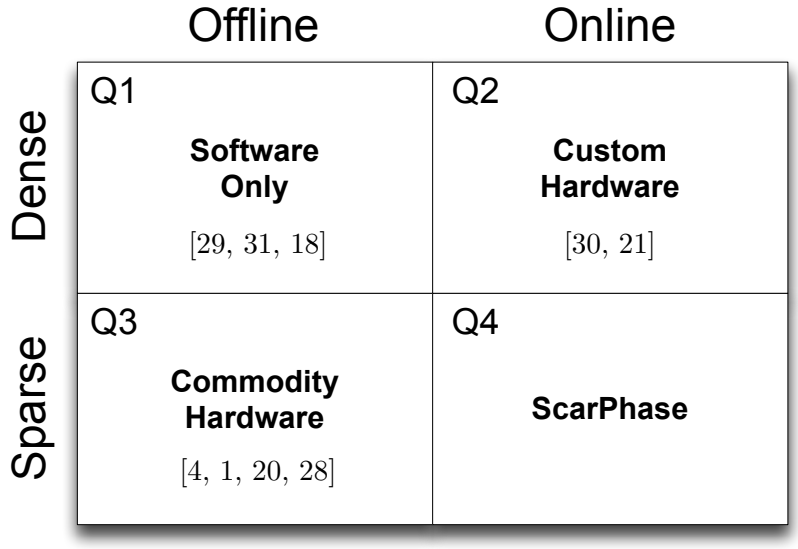


Figure I.1: Taxonomy of program phase classification methods.

which benefit from knowledge of program phases. For example, a CPU scheduler that knows the present phase of each runnable process can make better scheduling decisions and improve resource utilization [24, 32]. Other example include dynamic recompilation where knowledge of program phases can help determining when it is worth recompiling [2, 10, 26], efficient profiling where profiling is only performed for a phase with yet unknown behavior[25], or collection of representative simulation points to speed up simulation [31].

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

A large variety of program phase classification and prediction techniques have been studied in previous work [6, 5, 30, 18]. Most of these methods rely on the observation that the behavior of an application is highly correlated with the code it currently executes. This is typically captured using *basic block execution frequencies* that are collected dur-

ing the applications execution. The application's execution is divided into non-overlapping, adjacent, intervals for which execution frequencies are collected. If two intervals have similar enough execution frequencies they are classified as belonging to the same program phase.

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

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

The main contributions of this work are:

- We propose a new and efficient way to collect sparse frequency vector at a basic block granularity using Intel's Precise Event Based Sampling (PEBS) to sample branch instruction.

- We propose and evaluate a new sparse frequency vector based on conditional branches enabled by PEBS. This vector requires fewer samples and therefore reduces the overhead of data collection compared with previous work.

- We propose a method that allows us to dynamically adjust the sampling frequency, which reduces the runtime overhead by an additional factor of two.

- We have developed a general and easy to use library for online phase classification and prediction, called ScarPhase (Sample-based Classification and Analysis for Runtime Phases).

I.2 Phase Detection Algorithms

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

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

I.2.1 Execution Frequency Vectors

It has been observed that most performance metrics are strongly correlated with the code being executed [6, 5, 29, 31]. This observation has been the basis for many program phase classification techniques [18]. These methods typically use some form of *execution frequency vectors* to capture what code is executed during an execution interval, and define a measure of distance between the vectors. This distance is then used to compute the similarity of the intervals, and classify them into program phases.

Dhodapkar and Smith [6, 5] use a bit vector to keep track of what instructions have been executed during the execution intervals. When an instruction is executed its address is hashed into the bit vector, and the corresponding bit is set. Sherwood, Perelman, and Calder [29, 31]

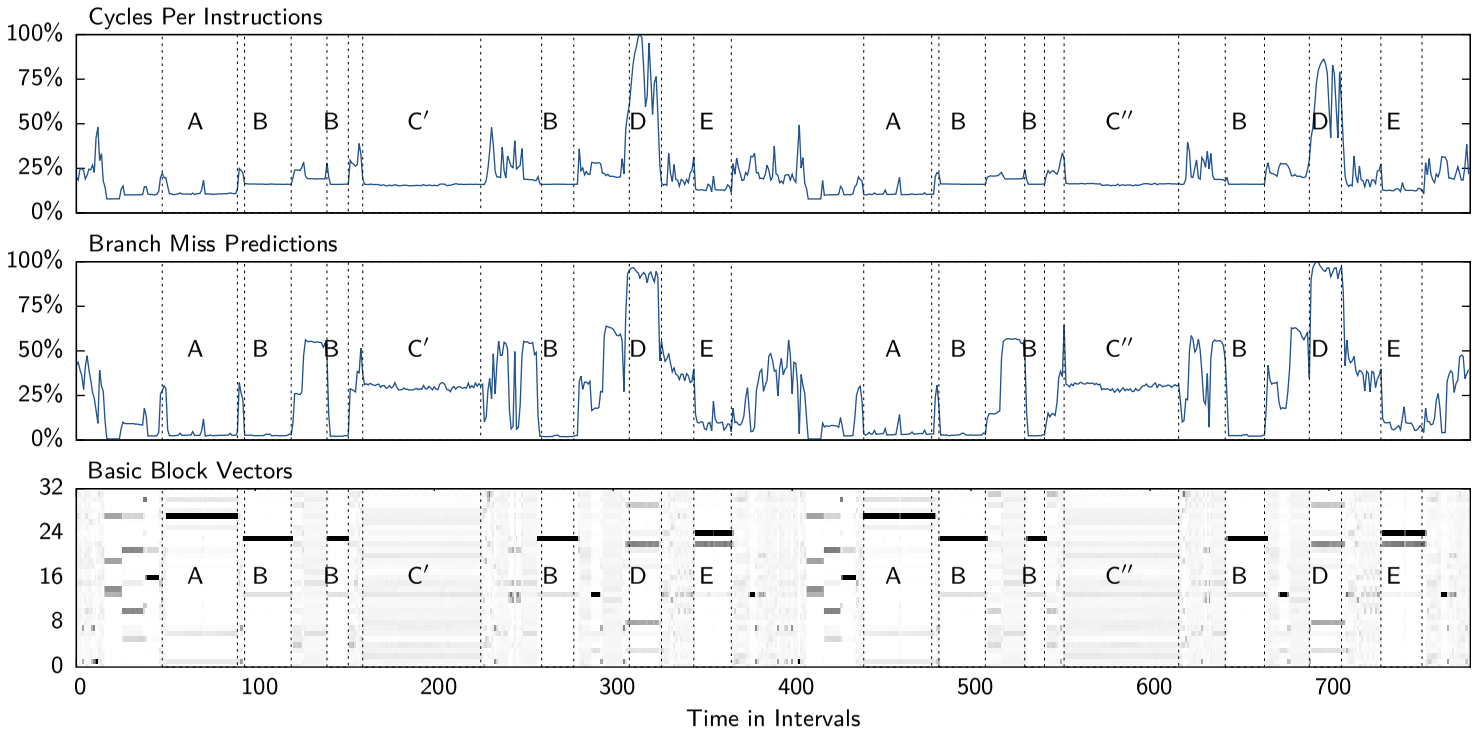


Figure I.2: The top and middle figures show how gcc/166's CPI and BMP changes over time, respectively, expressed as percentage of the maximum observed CPI/BMP. The bottom figure shows BBVs with 32 entries for each execution interval. The counters are shown on the y-axis, where darker shades of gray indicated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

keep track of the execution frequencies of basic blocks. They found that it is not necessary to have one counter for each basic block. Instead they use a small vector of counters, called a *Basic Block Vector* (BBV), and hash the basic blocks (using their addresses) onto the counters in the BBV. This has the benefit of reducing storage overhead and lowering cost of computing the distances between BBVs.

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

Lau et al. [19] evaluated the quality of phase classification based on the execution frequencies of a number of different program structures such as: function calls, loop branches, particular op-codes, register usage, memory access strides, and memory working sets. Their results suggest that none of these alternatives results in significant improvement over basic block execution frequencies.

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

I.2.2 Sparse Execution Frequency Vectors

To reduce the runtime overheads of collecting execution frequencies, Davies et al. [4] use hardware performance counters to collect sparse samples of instruction execution frequencies, which they use to estimate *Extended Instruction Pointer Vectors* (EIPV). EIPVs are similar to BBVs, but instead of capturing execution frequencies of basic blocks they capture execution frequencies of individual instructions.

Davies et al.'s original implementation uses VTune [15]. Our implementation, uses comparable features of Linux-perf_events [23], which has been available in the mainline Linux kernel since version 2.6.32. Using perf_events, the performance counters can be programmed to operate in two main modes. Either they count the occurrences of certain events, e.g., executed instructions, or they periodically trigger interrupts after a given number of occurrences of the event, called the sample period. These interrupts are caught by perf_events that in turn reacts to the interrupt in one of two different ways. Either it forwards the interrupt to user space in the form of a signal[1], or it records the state of the CPU, including its instruction pointer, at the time the interrupt is handled. These recorded CPU states can later be requested from user space.

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

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

While EIPVs can be collected fairly easily using standard hardware performance counter features, they are less accurate than BBVs. Since all the instructions in a basic block are executed the same number of times, it is more effective to count basic block execution frequencies and weight the count with the number of instructions in the basic blocks. Lau et al. [20, 28] showed that the results can be significantly improved by mapping the sampled instructions to their corresponding basic block. This however requires analysis of the program binary and is therefore not suitable for dynamically generated code. To leverage the low overhead data collection of Davies et al. but at the same time achieve the accuracy of BBVs, Lau et al. [20, 28] map instruction addresses to their corresponding basic blocks using the program binary. This however, has two problems: it increases the runtime data collection overhead, and it cannot be used to analyze dynamically generated code (e.g., JIT'ed code) as the mapping information is not readily available. We refer to these vectors as *Mapped Basic Block Vector* (MBBV).

I.2.3 Sparse Branch Vectors

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

Since each basic block ends with one branch instruction, we could attempt to record the addresses of sparsly selected branch instruction instead of the address of sparsely selected instructions. We could, for example, program perf_events to record the address of every Nth branch instruction, which at the end of the execution intervals would give us a sample of the executed branch instructions that directly corresponds to the BBVs. This, however, does not work due to performance counter skid [14].

If the performance counter is used to generate interrupts after N branch instructions, there will be a short delay between the time of the Nth branch and the time when the interrupt handler is invoked, during which the CPU keeps executing instructions. This delay is referred to

Table I.1: List of abbreviations

EIPV	Extended Instruction Pointer Vector [4]
MBBV	Mapped Basic Block Vector [20, 28]
BRV	BRanch Vector (New)
CBRV	Conditional BRanch Vector (New)

as *performance counter skid*. It is first when the interrupt handler is invoked that perf_events records the CPU state, which due to the skid can not be guaranteed to point to a branch instruction.

To reduce the impact of performance counter skid, Intel introduced *Precise Event Based Sampling* (PEBS) [13]. When PEBS[2] is enabled, the CPU saves its state at the time when the Nth event occurs. This saved state can then be read by perf_event's interrupt hander

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

I.2.4 Sparse Conditional Branch Vectors

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

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

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

[2]However, there is a small delay before the processor state is recorded, called shadowing [22]. For example, if we program a performance counter to trigger after the execution of 1000 branch instructions, the processor state might be recorded first when executing the 1001th instruction. This, however, does not present a problem for the work presented in this paper.

I.3 Experimental Setup

We performed our evaluation on *astar/lakes*, *bzip2/chicken*, *bwaves*, *dealii*, *gcc/166*, *mcf*, *perl/splitmail*, *wrf* and *xalan* from the SPEC 2006 [11] benchmark suite. We chose the programs above because they display the most interesting phase behavior. All benchmarks were run to completion with their reference input on an Intel Xeon E5620 (Nehalem) system.

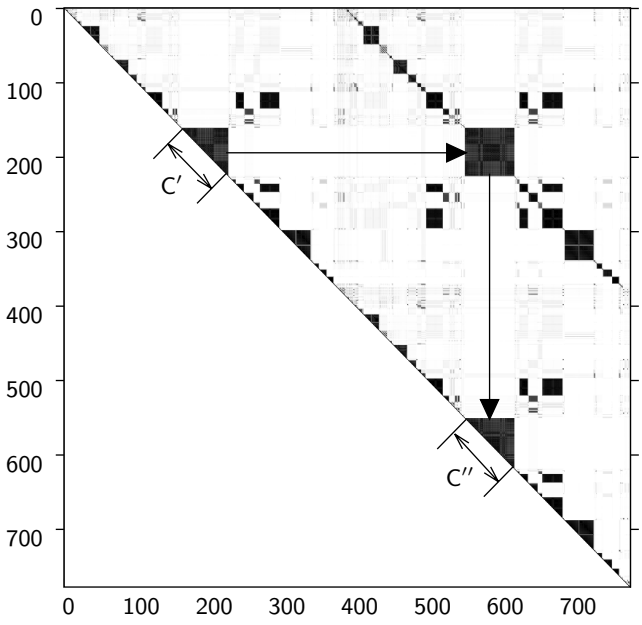
I.4 Case Study: GCC

In this section we evaluate the quality of the execution frequency vectors obtained using the methods discussed in section I.2.3 and I.2.4 by comparing them to execution frequencies obtained using a Pin [3] based reference implementation. The reference implementation dynamically instruments the benchmark applications and counts each execution of all basic blocks, and provides a ground-truth reference. We use an execution interval size of 100M instructions for both the reference and the evaluated methods.

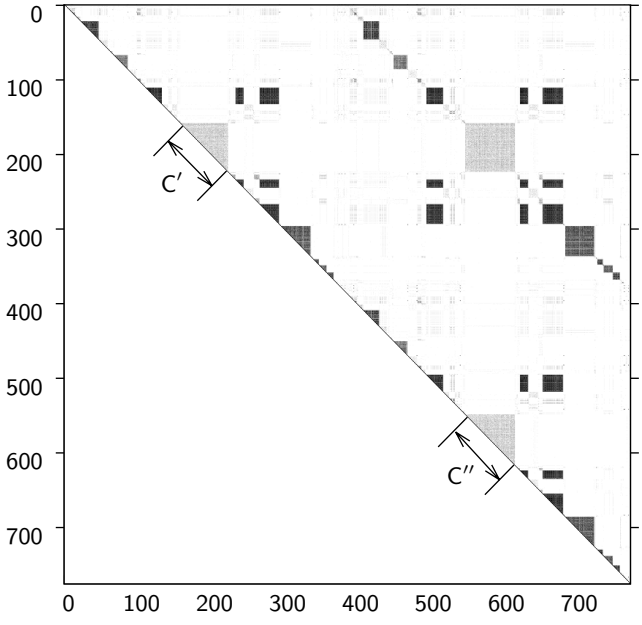
For the evaluation in this section we focus on gcc which is one of the SPEC CPU2006 benchmarks with the most complicated phase pattern. Figure I.2 shows how the CPI and branch miss predictions changes over time for gcc/166. The figure shows that the behavior of gcc changes greatly over time.

To quantify the difference between the captured execution frequency vectors and reference execution frequencies vectors, we compare their basic block similarity matrices [31]. A basic block similarity matrix is $N \times N$ upper triangular matrix M, where N is the number of intervals. Each element $M_{n,m}$ is the Manhattan distance between the execution frequency vectors of the nth and mth interval. Figure I.3a shows the similarity matrix for gcc/166 generated from the reference BBVs. The gray scale indicates the similarity between BBVs, where darker shades represents shorter Manhattan distances. The similarity matrix allows us to evaluate the quality of execution frequency vectors independent of the phase classification method.

We can interpret the basic block similarity matrix as follows. The application's execution progresses along the diagonal. The triangles above the diagonal indicate that neighboring intervals have similar execution frequency vectors and therefore belong to the same phase. For example, see Figure I.4, where we have labeled the two occurrences of phase C (C' and C''). To find possible reoccurrences of a phase, we start at the triangle marking the phase and move horizontally to the right across the



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



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

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

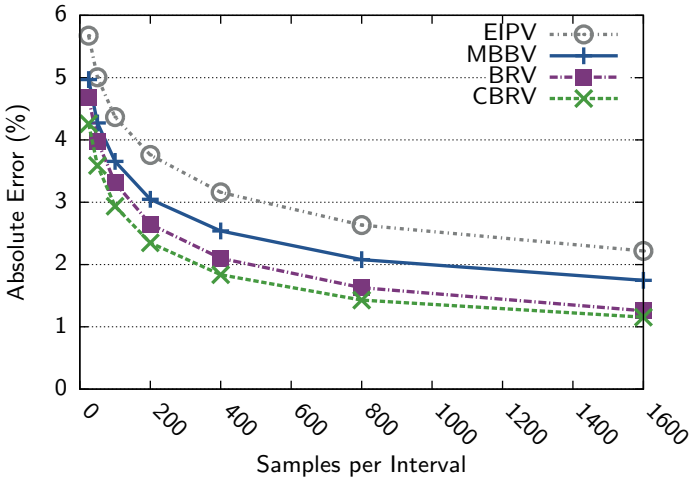


Figure I.4: The average absolute element-wise differences between the reference similarity matrix and the similarity matrices generated from data captured using EIPVs, MBBVs, BRVs and CBRVs for gcc/166. Lower is better.

matrix until we reach a dark rectangle. Then, from the rectangle, we continue straight down until we reach the diagonal. The triangle that we land on indicates the similarity of a set of intervals which belong to the phase where we started.

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

To quantify the difference between two similarity matrices we compute the average element-wise difference between the two matrices. Figure I.4 shows the average elemen-wise difference between the reference similarity matrix and the ones based on EIPVs, MBBVs, BRVs and CBRVs for a range of sample periods[3], i.e., the number of "sampled"

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

instructions/branches during execution intervals. As expected the similarity improves when the number of samples per interval is increased. However, the curves start to level of when more than 800 samples per interval is used, suggesting that there is no real benefit to use more than 800 samples for any of the methods.

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

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

I.5 Phase Analysis

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

For phase classification we use the leader-follower clustering algorithm [7]. It identifies cluster of similar execution frequency vectors, and we interpret each clusters as program phases. The algorithm works as follows. At the end of every execution interval, we apply leader-follower clustering to the interval's execution frequency vector. If the vector is close enough to the center of an existing cluster, it is added to the cluster, and the cluster's center is recomputed. Here, close enough means that the Manhattan distance between the execution frequency

vector and the cluster center is below a threshold. Otherwise a new cluster is created to which the vector is added. The interval is classified as belonging to the phase defined by the cluster to which it was added.

I.5.1 Online Phase Classification

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

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

To measure phase homogeneity we use the *Coefficient of Variation* (CoV) [30]. To compute the CoV we first measure the CPIs of all execution intervals. Then, for each phase, we compute both the average and the standard deviation of the CPIs of the intervals belonging to the same phase. The per-phase CoV is then the standard deviation divided by the average. Finally we compute the whole program CoV as the weighted average of the per-phase CoVs. Note that CoV is a lower-is-better metric.

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

a phase classification method that classifies all intervals as belonging to the same phase). This gives us a goodness metric that penalizes phase classification methods that identify too many phases. We call this new metric Corrected CoV (CCoV). Now, in the extreme case when all execution intervals are classified into different phases, they will all end up in the virtual phase and the CCoV will be equal to the CoV of the whole program, which is the desired result.

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

I.5.2 Runtime Overhead

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

To measure what it cost to use PEBS we measured the overhead of sampling every one thousand instructions with and without PEBS and divided the overhead by the number of samples. Figure I.6 (Buffered) shows the time in microseconds to collect one sample. For gcc/166, the cost to collect one sample using PEBS is 33% more expensive than without PEBS support.

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

Figure I.6 (Buffered vs. Unbuffered) shows the time to collect one sample with and without buffering. On average, the sample cost is reduced with a factor 11. This means that for the same overhead, 11 times more samples can be collected. It is therefore vital to buffer the samples in kernel space for online phase classification.

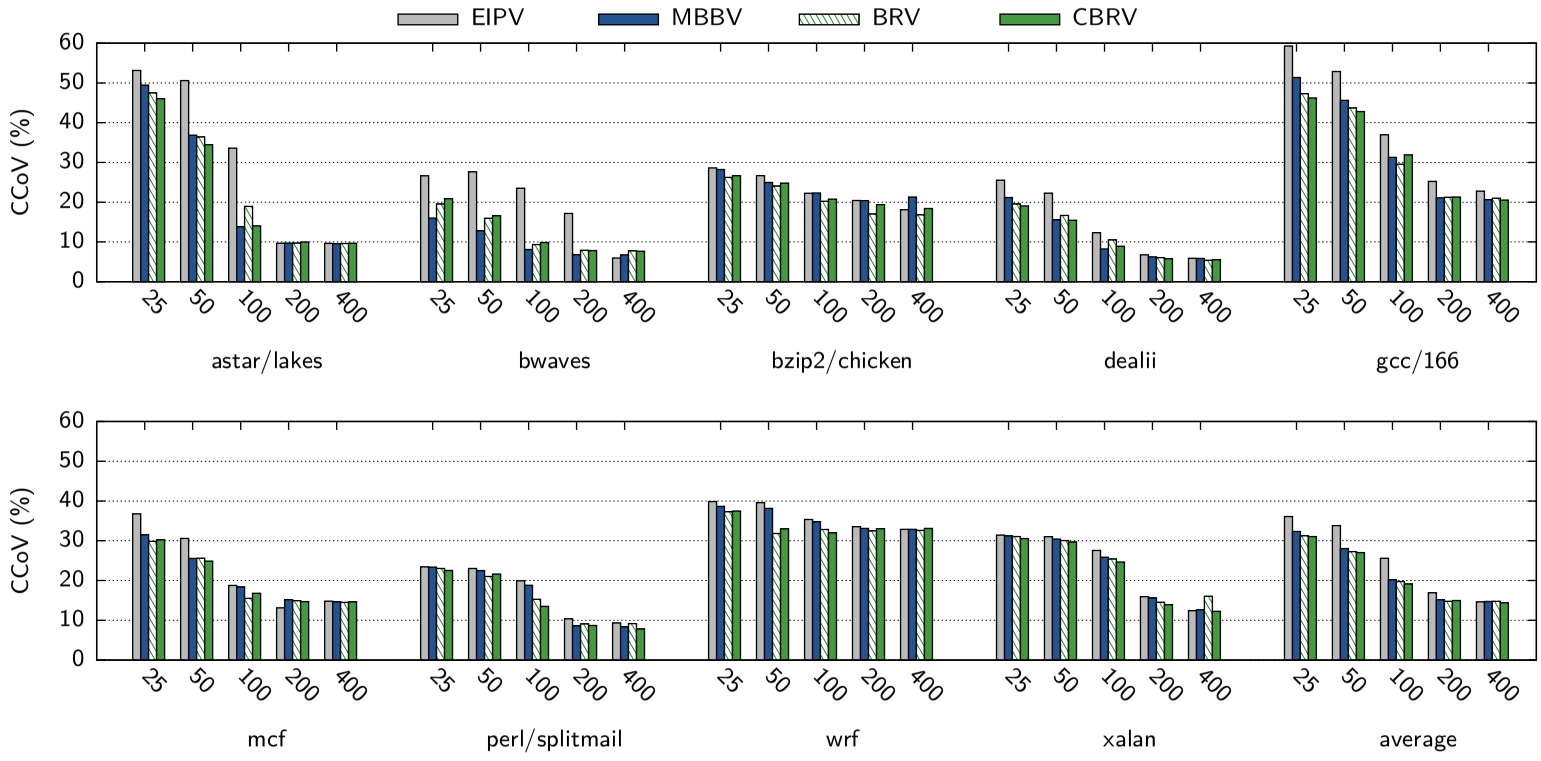


Figure I.5: CCoV for different number of samples per interval. Lower is better.

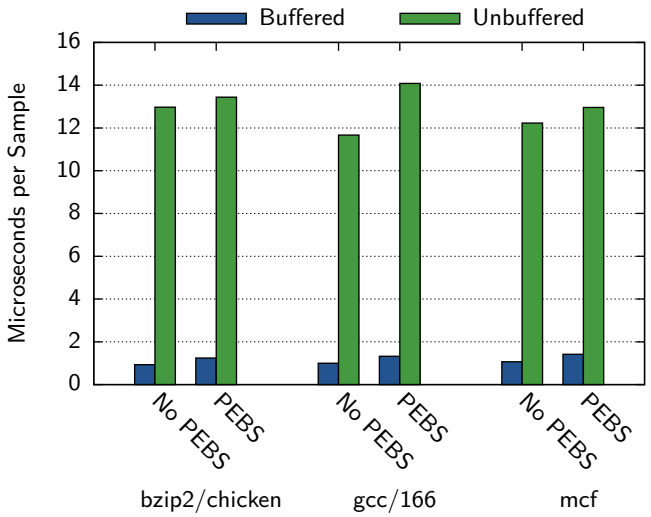


Figure I.6: The average time in microseconds to collect one sample. With and without using in kernel buffering, and with and without using PEBS.

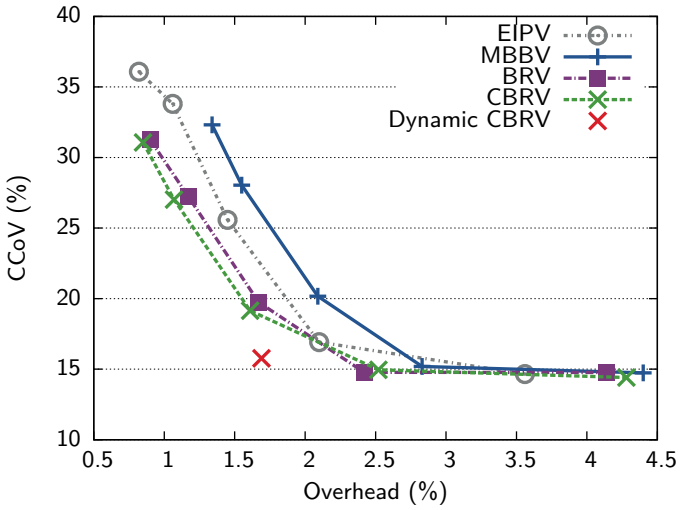


Figure I.7: Average CCoV vs runtime overhead.

In the previous section we evaluated the accuracy at different number of samples, however the cost of a sample varies between the methods. Figure I.7 shows the accuracy against the overhead. This shows that CBRV results in the most accurate phase classification for a given overhead and MBBV the worst. Comparing EIPV with BRV/CBRV shows that the benefit of using PEBS clearly outweighs the additional runtime

cost.

One design parameter of great importance for both accuracy and overhead, but not evaluated here, is the execution interval size. In practice, the most appropriate interval size may depend on how the phase information will be used. Using larger execution intervals would have an almost linear effect on the runtime overhead for all methods studied in this section (assuming that the number of samples per interval is held constant). Throughout this paper we use a execution interval size of 100M instructions as in most prior works [31, 4, 1, 28].

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

I.6 Dynamic Sample Rate

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

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

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

[4]When we look for phase changes and are sampling at a lower rate, we are more

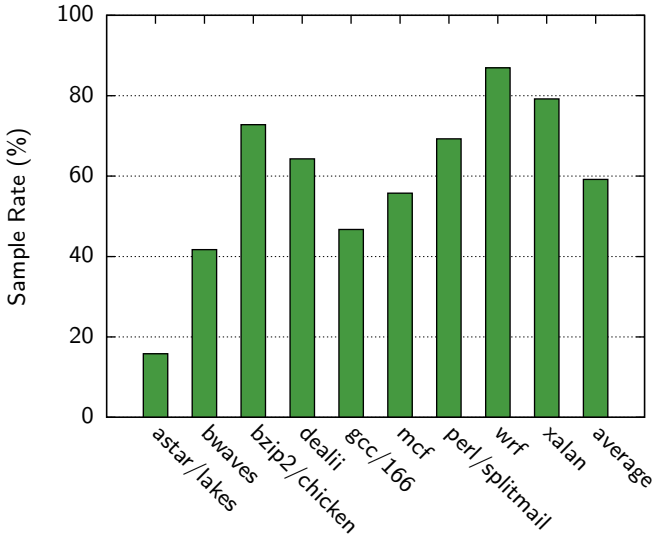


Figure I.8: Average number of samples per execution interval when using dynamic sample rate. The number of samples are shown as percentages of the maximum number of samples per interval (200).

sample rate. This is done exponentially until we reach a lower limit of 25 samples per interval. When a phase change is detected, we go back to sample 200 branches per interval.

There are two things to note about the above method. First, when detecting a phase change, we have to classify the current interval (i.e. the first interval in the new phase), however, as we have lowered the sample rate, we have less than 200 samples, and the classification might therefore be less accurate. To reduce the impact of this we use a next phase predictor that predicts the phase of the next interval. If it predicts that the next interval belongs to a different phase, we pessimistically increase the sample rate back to 200 samples per interval, and can now more accurately classify the interval. For this, we use a history-based predictor similar to the one used by Sherwood, Sair, and Calder [30]. Second, if the predictor incorrectly does not predict a phase change, it will not help us, and we need to classify the current interval based on less than 200 samples. This is particularly troublesome if we incorrectly classify the interval as a new phase (i.e. one that we have not previously encountered) as this increases the total number of detected phases. Therefore, whenever we detect a new phase while sampling with a lowered sample rate, we do not attempt to classify the interval, and

sensitive to sampling noise, and we therefore use a somewhat higher threshold than when doing full phase classification.

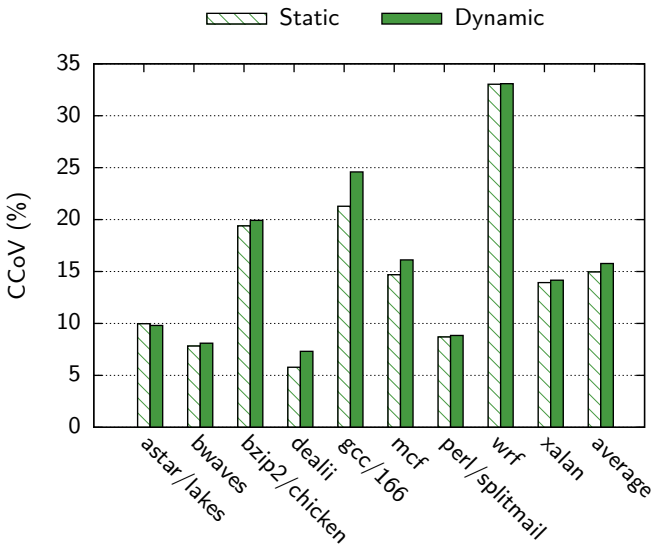


Figure I.9: CCoV when using static sample rate (left bar) and dynamically adjusted sample rate (right bar).

consider it as unclassified.

Figure I.8 shows the average sample rate in percentage of the maximum sample rate (corresponding to 200 samples per interval) for all benchmarks. astar/lakes has few phases with very long run lengths, the sample rate is therefore reduced to 15% (30 samples per interval). wrf on the other hand has many phase changes, and the sample rate is therefore only reduced to 87% (174 samples per interval). Using the dynamic sample rate adjustment, the sample rate is reduced on average to 59% (118 samples per interval), which results in an average runtime overhead of 1.7%.

Figure I.9 shows the CCoV for all benchmarks when using both static sample rate (i.e. 200 samples for all interval) and dynamic sample rates adjustment. When computing the CCoV for dynamic sample rate, we group all unclassified intervals into a "junk" phase. This phase will have a large CoV, as its constituent intervals belong to many different phases, and undesired unclassified intervals therefore contribute to an increased CCoV. As the figure shows, using dynamic sample rate does not significantly increase the CCoV. The benchmark where the CCoV increase the most is gcc/166, this is mainly due to gcc having the largest number of unclassified intervals, 1.7% of its intervals are unclassified. Across all benchmarks only 0.4% of the intervals are unclassified.

Figure I.10 shows the runtime overhead for all benchmarks, for both static and dynamic sample rate, and for collecting both MBBVs and

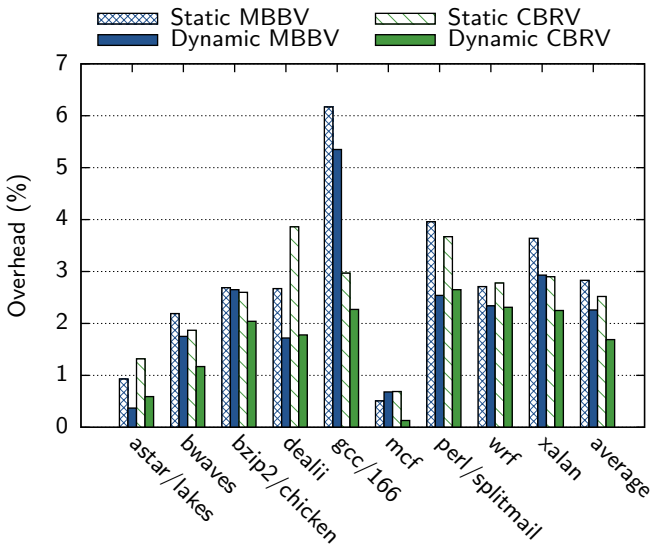


Figure I.10: Runtime overhead in percentage, when using both static and dynamically adjusted sample rate to collect MBBVs and CBRVs.

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

In summary, the results of this section shows that collecting CBRVs using dynamic sample rate outperforms all previous approaches, both in terms of accuracy and runtime overhead. This is clearly shown in Figure I.7 where we have marked the average CCoV and runtime overhead of CBRV based phase classification using dynamic sample rate with an $\times$.

I.7 ScarPhase

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

In the rest of this section we describe a practical usage study de-

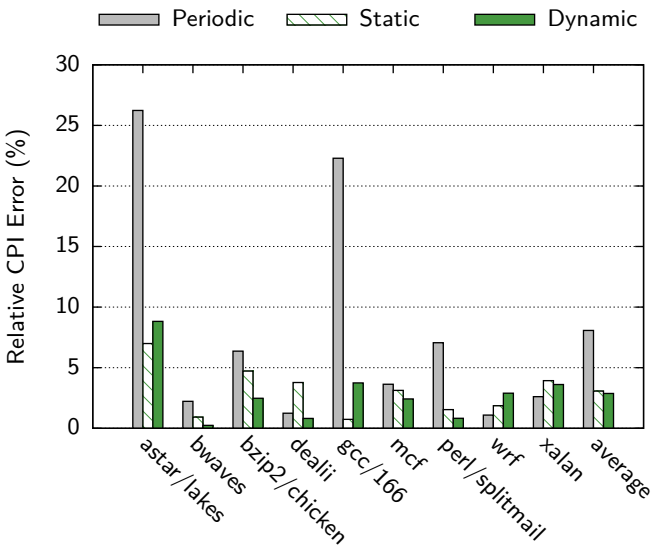


Figure I.11: The relative error between the measured CPI and the estimated average CPI based on periodic sampling and phase guided profiling using both static and dynamically adjusted sample periods. Lower is better.

scribing how ScarPhase is utilized to cheaply and accurately profile an application.

I.7.1 Use Case: Phase Guided Profiling

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

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

To implement the phase guided profiling, we use the library to predict the next interval; the interval is then profiled if it belongs to a phase that has not yet been profiled. (In this example, we simply measure the CPIs of the intervals.) To avoid small phases that do not make a sig-

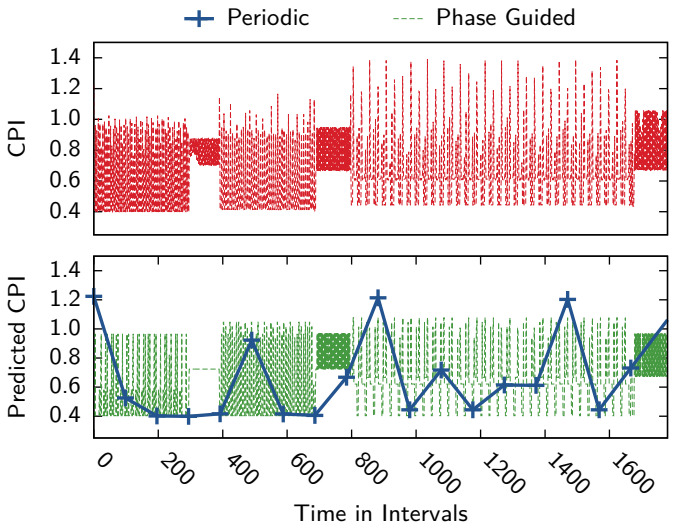


Figure I.12: CPI changes over time for bzip2/chicken (top), and the reconstructed program execution (bottom) using periodic sampling (front, bottom) and phases guided sampling (back, bottom). It should be noted that the period from 800 to 1600 is not a single phase, but a series of smaller phases.

nificant contribution to the average CPI, a phase is only profiled after it has been seen in a certain number of intervals. To estimate the overall CPI of the profiled application, we compute average CPI of the phases weighed with the number of intervals detected for each phase. (If the same phase was profiled more than once due to miss-prediction, we use the median CPI).

Figure I.11 shows the relative error between the measured CPI and the estimated average CPI, for both periodic sampling[5] (Periodic) and phase guided profiling. The phase guided profiling was done using both static (Static) and dynamic (Dynamic) sample period adjustment. Phase guided profiling can accurately estimate the CPI with an average error of 3%, while periodic sampling has an average error of 8%. Furthermore, the accuracy of periodic sampling varies a lot between the applications. For example, gcc/166 has an error of 22%, while the error for wrf is only 1%. The standard deviation for phase guided profiling and periodic sampling is 2 and 9.5 respectively.

With phase guided profiling we can cheaply reconstruct the profiled

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

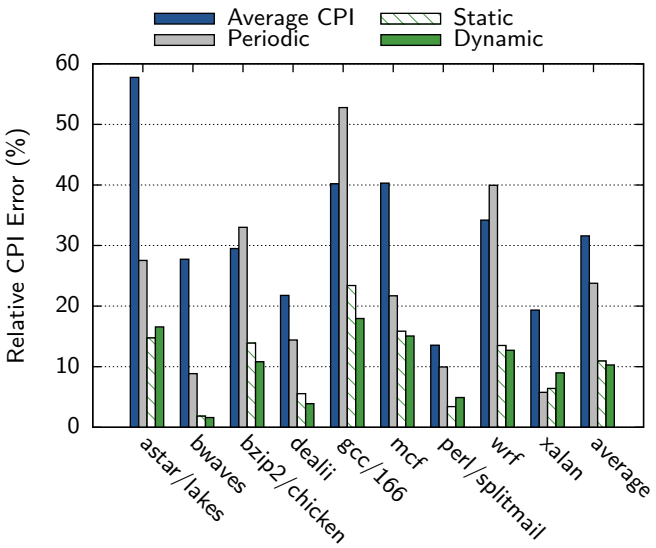


Figure I.13: The average relative difference between the measured CPI and the predicted CPI for each interval. The executions were reconstructed using both periodic sampling and phase guided profiling with both static and dynamically adjusted sample periods. Lower is better.

application's execution behavior over time. This cannot be easily done using periodic sampling[6]. To do this, the CPI of each interval is predicted to be the same as the CPI of the phase it belongs to. (The CPI of a phase is the CPI of the profiled interval.) Figure I.12 shows how the CPI changes over time for bzip2/chicken. The top figure shows the measured CPI, and the bottom figure shows the reconstructed CPI. The line labeled periodic in the bottom figure, shows an attempt to reconstruct the CPI using periodic sampling by interpolating between the profiled intervals. For phase guided profiling, only 0.7% and 0.8% of the execution intervals are profiled using static and dynamic sample period adjustment respectively, however the profiled intervals represent 90% and 91% of the application's execution.

Figure I.13 shows average relative difference between the measured CPI and the predicted CPI for each interval. Using phase guided profiling with and without dynamic sample rate results in an error of 10% and 11% respectively, while periodic sampling has an error of 24%. Using the average CPI to predict the behavior for the whole execution results in an error of 32%. This shows that predicting the program's behavior with an average can be misleading.

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

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

I.8 Related Work

In this paper we have focused on *temporal phases* [29, 6], however, there are alternative definitions of phases such as *code phases* [12, 9, 17, 10, 32]. Instead of observing the execution behavior over time, the program binary is analyzed, where parts of program's control flow is grouped into phases. For example, if the number of instructions in a function is above a threshold, the function is considered to be a phase. This has been used to instrument code at the phase boundaries for various optimizations. Both software [9, 10, 32] and hardware [12, 17] approaches have been suggested.

Gu and Verbrugge [10] used code phases to find the best optimization level for each function with respect to compilation/execution time for dynamic recompilation in Java virtual machines. Sondag and Rajan [32] instrumented phase boundaries for scheduling, and moved threads between heterogeneous processors when they changed phase. They found that phase guided scheduling can significantly improve the throughput compared to the standard Linux scheduler.

Temporal phases (see section I.2) divide the execution into intervals and group similar intervals into phases. This has been used to reduce the overhead of profiling [25] and finding representative part of the execution for architectural simulation [29, 31, 4, 28].

A sub set of temporal phase classification is phase change detection and predicting the applications performance behavior [5, 8, 16, 26]. Peleg and Mendelson [26] showed that changes in the CPI can not be used as a metric for loaded systems. Instead, BBVs, instruction working sets, and other architecture independent metrics have been used to detect performance behavioral changes. This is different from phase classification in that only the last intervals are of interest, i.e., the program changes phase if the difference between the last two intervals is above a threshold. However, remembering reoccurring phases can improve performance by reusing configuration settings [8, 16].

Dhodapkar and Smith [5] used instruction working set to reconfigure the size of caches at runtime when the application changes phase. Isci, Contreras, and Martonosi [16] predicted the ratio between mem-

ory transactions and micro operations to reduce the power consumption using dynamic voltage frequency scaling.

I.9 Conclusions

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

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

I.10 References for Paper I

[1] Murali Annavaram et al. "The Fuzzy Correlation between Code and Performance Predictability". In: *Int. Symposium on Microarchitecture*. 2004.

[2] Ronald D. Barnes et al. "Vacuum packing: extracting hardware-detected program phases for post-link optimization". In: *Int. Symposium on Microarchitecture*. 2002.

[3] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Conf. on Programming Language Design and Implementation*. 2005.

[4] B. Davies et al. *iPART : An Automated Phase Analysis and Recognition Tool*. Tech. rep. IR-TR-2004-1-iPART. Intel Corporation, 2004.

[5] Ashutosh S. Dhodapkar and James E. Smith. "Comparing Program Phase Detection Techniques". In: *Int. Symposium on Microarchitecture*. 2003, pp. 217–.

[6] Ashutosh S. Dhodapkar and James E. Smith. “Managing multi-configuration hardware via dynamic working set analysis”. In: *Int. Symposium on Computer Architecture*. 2002.

[7] Richard O. Duda, Peter E. Hart, and David G. Stork. “Pattern Classification”. In: 2nd ed. Wiley-Interscience, 2001. Chap. 10.11. On-line Clustering, pp. 559–565. ISBN: 0-471-05669-3.

[8] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. “Characterizing and Predicting Program Behavior and its Variability”. In: *Int. Conf. on Parallel Architecture and Compilation Techniques*. 2003.

[9] Andy Georges et al. “Method-level phase behavior in java workloads”. In: *Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*. 2004.

[10] Dayong Gu and Clark Verbrugge. “Phase-based adaptive recompilation in a JVM”. In: *Int. Symposium on Code generation and Optimization*. 2008.

[11] John L. Henning. “SPEC CPU2006 benchmark descriptions”. In: *SIGARCH Comput. Archit. News* (2006).

[12] Michael C. Huang, Jose Renau, and Josep Torrellas. “Positional adaptation of processors: application to energy reduction”. In: *Int. Symposium on Computer Architecture*. 2003.

[13] *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Volume 3B: System Programming Guide. 30.4.4 Precise Event Based Sampling (PEBS). Intel Corporation. 2010.

[14] *Intel VTune Amplifier XE 2011 Getting Started Tutorials for Linux* OS*. Section Key Concept: Event Skid. 2010. Chap. Key Concept: Event Skid, p. 15.

[15] *Intel VTune Performance Analyzer Homepage*. URL: `http://www.intel.com/software/products/vtune/.`.

[16] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. “Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management”. In: *Int. Symposium on Microarchitecture*. 2006, pp. 359–370.

[17] Jinpyo Kim et al. “Dynamic Code Region (DCR)-based Program Phase Tracking and Prediction for Dynamic Optimizations”. In: *Int. Conf. on High Performance Embedded Architectures and Compilers*. Springer Verlag, 2005.

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

[19] J. Lau et al. “Motivation for Variable Length Intervals and Hierarchical Phase Behavior”. In: *Int. Symposium on Performance Analysis of Systems and Software*. 2005.

[20] J. Lau et al. “The Strong correlation Between Code Signatures and Performance”. In: *Int. Symposium on Performance Analysis of Systems and Software*. 2005.

[21] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. “Transition Phase Classification and Prediction”. In: *Int. Symposium on High-Performance Computer Architecture*. 2005.

[22] David Levinthal. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*. Tech. rep. Version 1.0. Intel Corporation, 2009.

[23] *Linux perf_events*. URL: `Linux/include/linux/perf\_event.h`.

[24] Yangchun Luo et al. “Dynamic performance tuning for speculative threads”. In: *Int. Symposium on Computer Architecture*. 2009.

[25] Priya Nagpurkar, Chandra Krintz, and Timothy Sherwood. “Phase-Aware Remote Profiling”. In: *Int. Symposium on Code generation and Optimization*. 2005.

[26] Nitzan Peleg and Bilha Mendelson. “Detecting Change in Program Behavior for Adaptive Optimization”. In: *Int. Conf. on Parallel Architecture and Compilation Techniques*. 2007.

[27] Cristiano Pereira et al. “Dynamic phase analysis for cycle-close trace generation”. In: *Int. Conf. on Hardware/software Codesign and System Synthesis*. 2005.

[28] Erez Perelman et al. “Detecting phases in parallel applications on shared memory architectures”. In: *Int. Parallel and Distributed Processing Symposium*. 2006.

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

[30] Timothy Sherwood, Suleyman Sair, and Brad Calder. “Phase tracking and prediction”. In: *SIGARCH Comput. Archit. News* (2003).

[31] Timothy Sherwood et al. “Automatically characterizing large scale program behavior”. In: *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. 2002.

[32] Tyler Sondag and Hridesh Rajan. “Phase-guided thread-to-core assignment for improved utilization of performance-asymmetric multi-core processors”. In: *ICSE Workshop on Multicore Software Engineering*. 2009.

Paper II

Paper II

Phase Behavior in Serial and Parallel Applications

Andreas Sembrant, David Black-Schaffer, and Erik Hagersten

In Proceeding of the
International Symposium on Workload Characterization
San Diego, California, USA, November 2012

IISWC November 2012, San Diego, California, USA 978-1-4673-4532-3/12/$31.00

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×) with larger variation in CPI (1.5×) 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.

II.1 Introduction

Most programs have time varying phase behavior [39, 41, 14, 11]. This insight has been exploited in various dynamic runtime optimizations, e.g., cache resizing [12, 40, 38, 26], dynamic voltage frequency scaling [20], scheduling [42] and phase-guided profiling [31, 36]. However, most of this research is based on the SPEC [17] 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 [7] 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, 41, 34, 1], only aggregated metrics (e.g., average cache miss ratio) have been used to characterize its behavior [7, 6, 4, 5], 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 [37] 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 [40, 32, 37]. 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×) and more variation (1.5×) 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 [37] to detect phases in parallel applications.

- 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

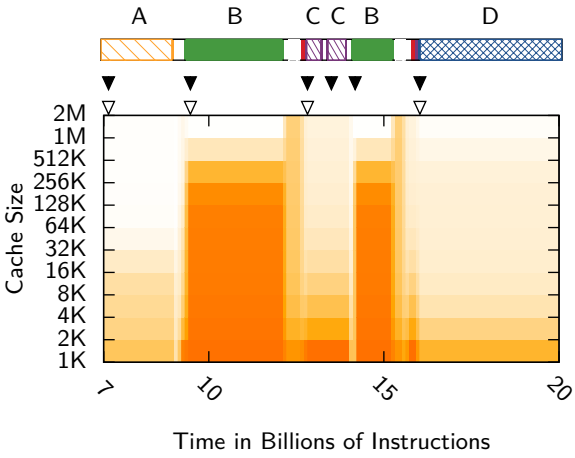


Figure II.1: Miss-ratio (intensity) [36] as a function of time (x-axis) and cache allocation (y-axis) for gcc/166. The detected execution phases are shown above, with shorter phases shown in white for clarity. The black triangles shows when the phase detector notices a new phase and the runtime system applies the best optimization for the current phase.

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.

II.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.

II.2.1 Detecting Program Phases

We use the ScarPhase [37] 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 [41, 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 differ-

ent metrics [39, 41, 11, 40, 22]. 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) [28, 18]. 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 [13]. Windows with similar vectors are then grouped into the same cluster and considered to belong to the same phase.

II.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 II.1 [36]. 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, 40, 38, 26, 20, 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 II.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, 36] is a method to reduce the overhead of profiling without sacrificing accuracy, by 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 II.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.

II.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 [14, 40, 27, 20], 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 [40, 27] with a 256 entries lookup table, a run length encoding of 2 and a confidence threshold of 1.

II.3 Methodology

We ran our analysis on all benchmarks in the SPEC2006 [17] and PARSEC 2.1 [7] benchmark suites. However, we also investigated other parallel benchmarks suites. We found that the NAS [3] 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 [29] to collect runtime data (e.g., CPI, L3 miss ratio, etc.) during each window.

[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, 41, 10, 2, 15, 25, 35, 37, 36]. We did preliminary tests with a widow size of 10M instructions but found similar results.

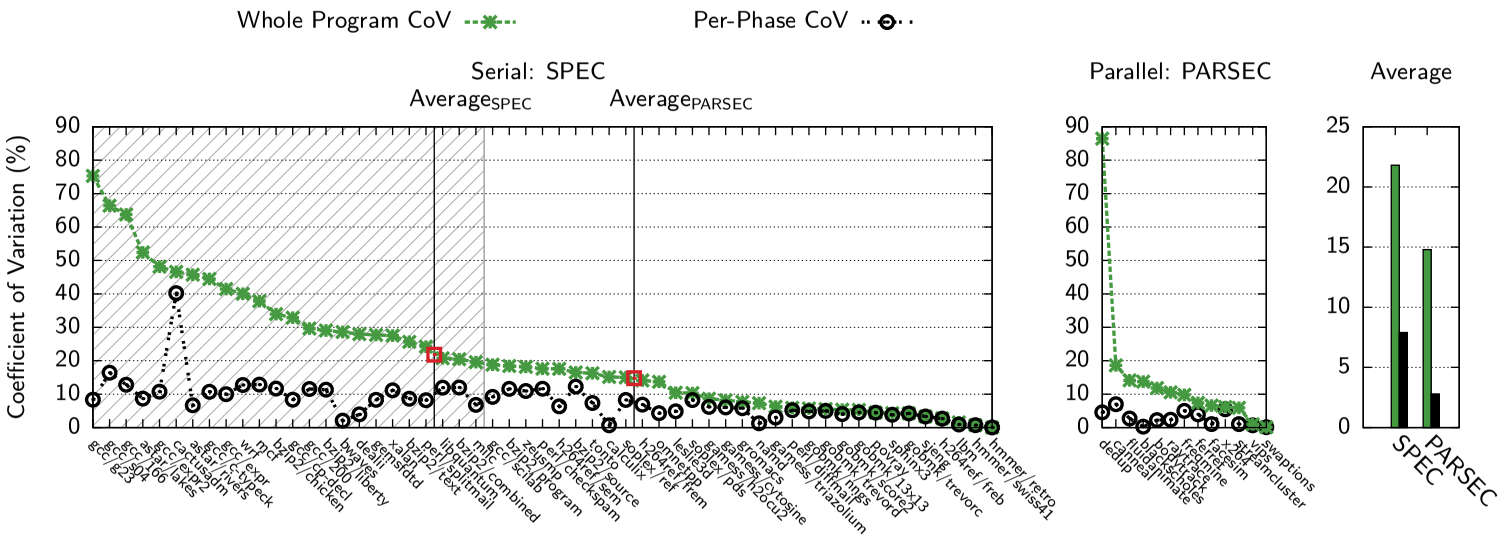


Figure II.2: The CPI CoV (standard deviation divided by the mean). The figure shows how the CPI varies over the whole program execution (Whole Program CoV) and within a phase (Per-Phase CoV). The higher the CoV, the higher variation in CPI. The gray area highlights the difference between the benchmarks 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 II.4.1), the serial SPEC benchmarks shows significantly more program variation than the parallel PARSEC benchmarks* (1.5× *on average).*

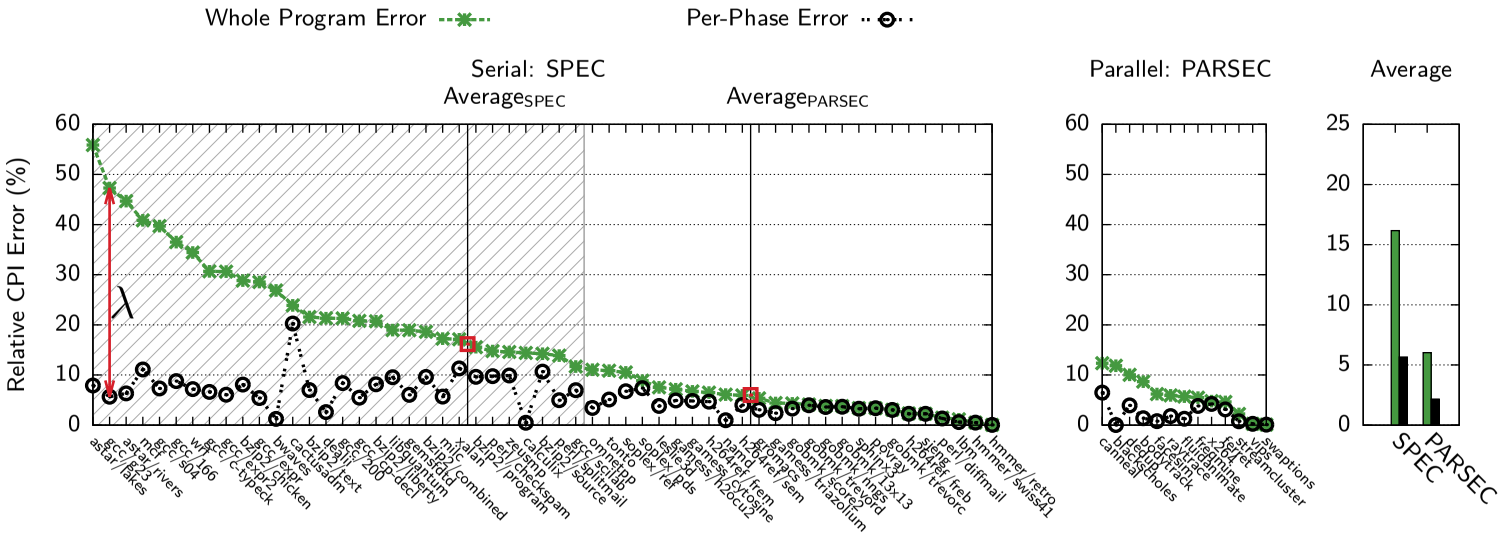


Figure II.3: The relative CPI error between the CPI in each window and the average CPI for the whole program (Whole Program Error) and average CPI per phase (Per-Phase Error). All SPEC programs in the gray region have higher CPI error than all the PARSEC programs. The figure shows the importance of using 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.*

II.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.

II.4.1 Time Varying Behavior

To characterize an application's time varying behavior, we use the *Coefficient of Variation* (CoV) [41, 40, 22], 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 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 II.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 II.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-

[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.

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 II.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 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.

II.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 corresponding pattern the phases appear in.

Figure II.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

[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.

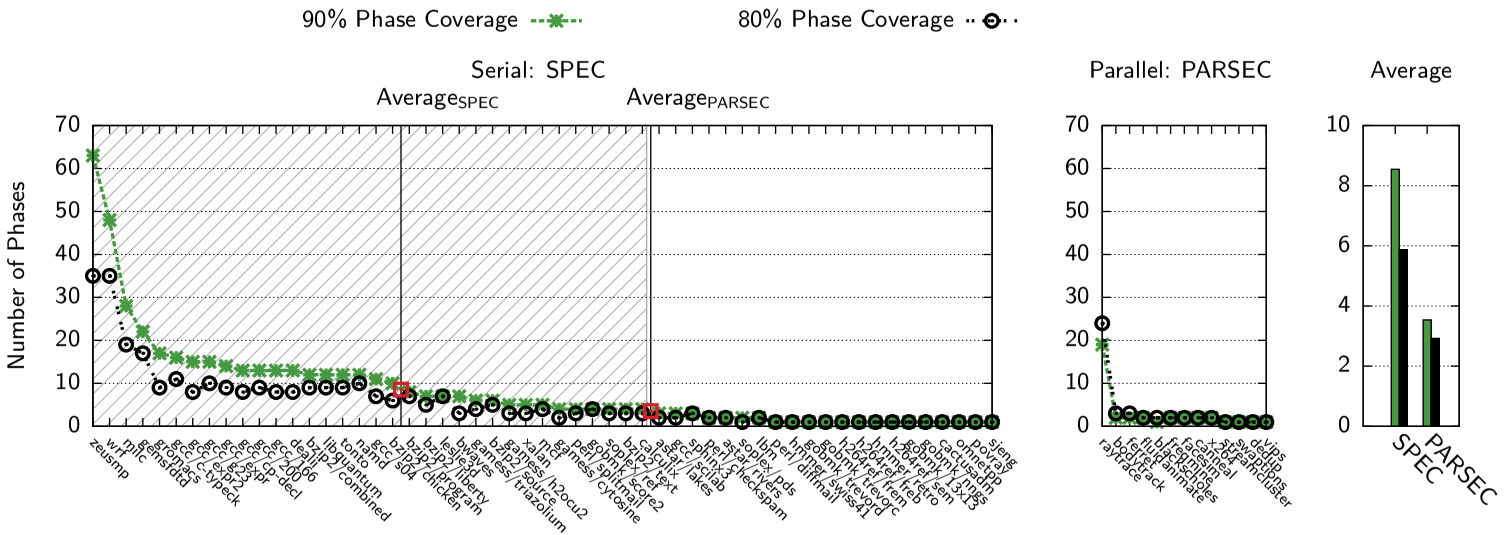


Figure II.4: The number of phases that are needed to cover 80% and 90% of the program execution. The gray area highlights the difference between the benchmarks suites. All SPEC programs in the gray region have more phases than all the PARSEC program except raytrace. *This shows that SPEC has more phases (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.*

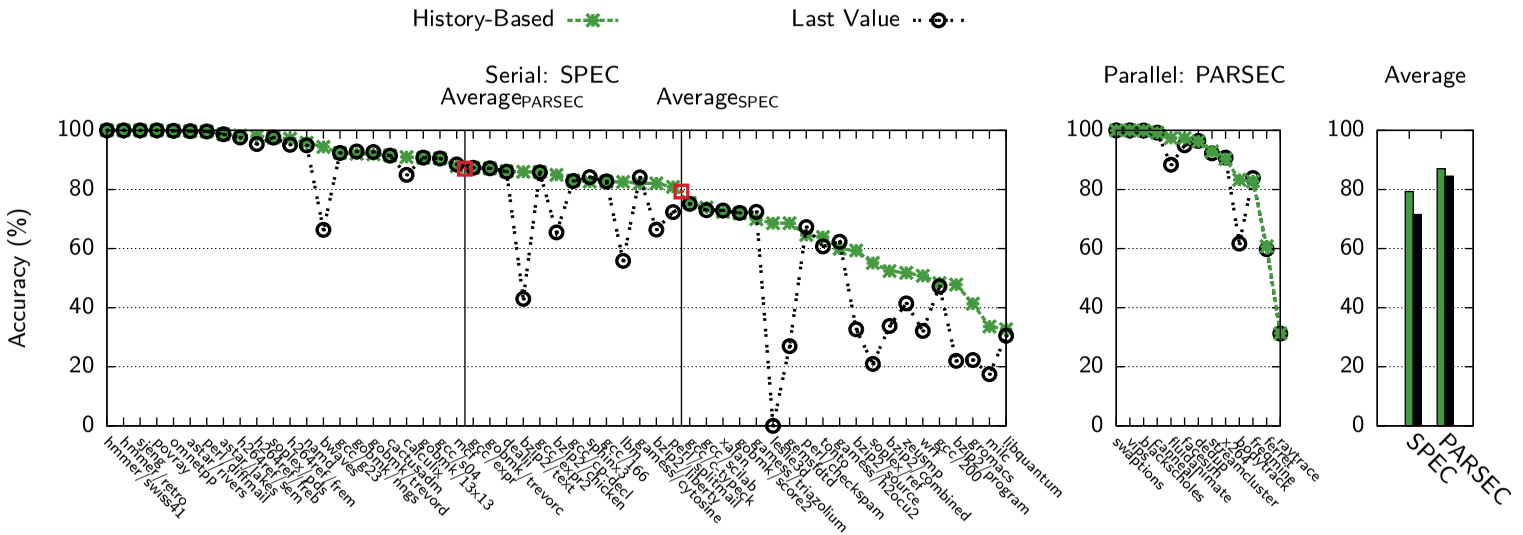


Figure II.5: The prediction accuracy of predicting the phase id of the next window, using last value prediction (LV) and history based prediction with run length encoding (RLE). *The two benchmarks suites show similar accuracy, with a slight advantage (i.e., easier to predict) to PARSEC.*

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 II.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 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.

II.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× 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.

II.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 ap-

[4]This can happen due to aliasing and when the window size does not match the underlying phase bahavior [23].

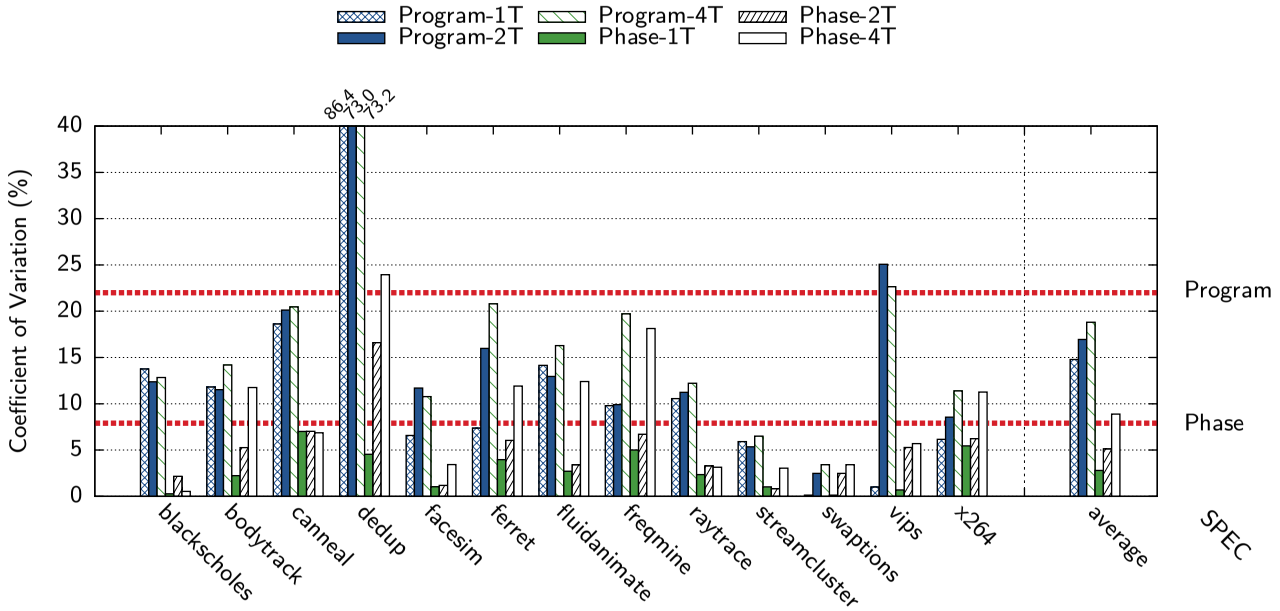


Figure II.6: The CPI CoV (standard deviation divided be the mean) for 1, 2 and 4 threads. The figure shows how the CPI varies over the whole execution (Whole Program CoV) and within a phase (Per-Phase CoV). *The CPI variations increases slightly with more threads for both the whole execution and within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

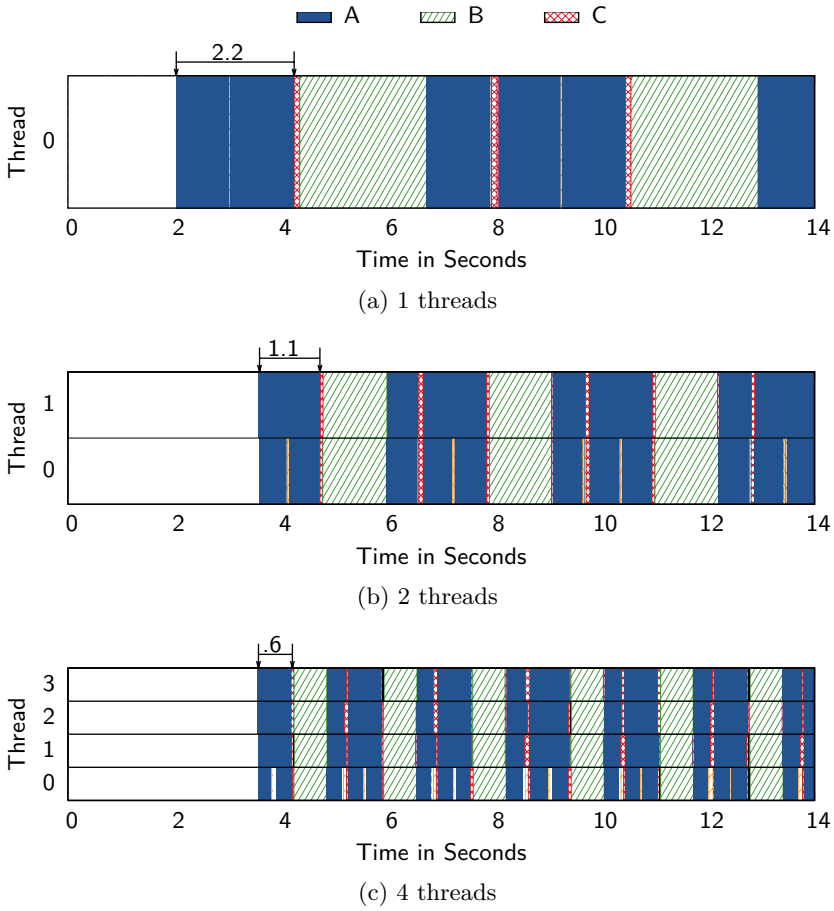
plications, we extend the ScarPhase library to monitor and detect phases at runtime in multiple threads. To do so, ScarPhase asynchronously detect phases in the application. Whenever a thread finishes executing a window, ScarPhase classifies what phase the window belongs to using the same method as in 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 II.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.

II.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 II.7, streamcluster in Figure II.8 and dedup in Figure II.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 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



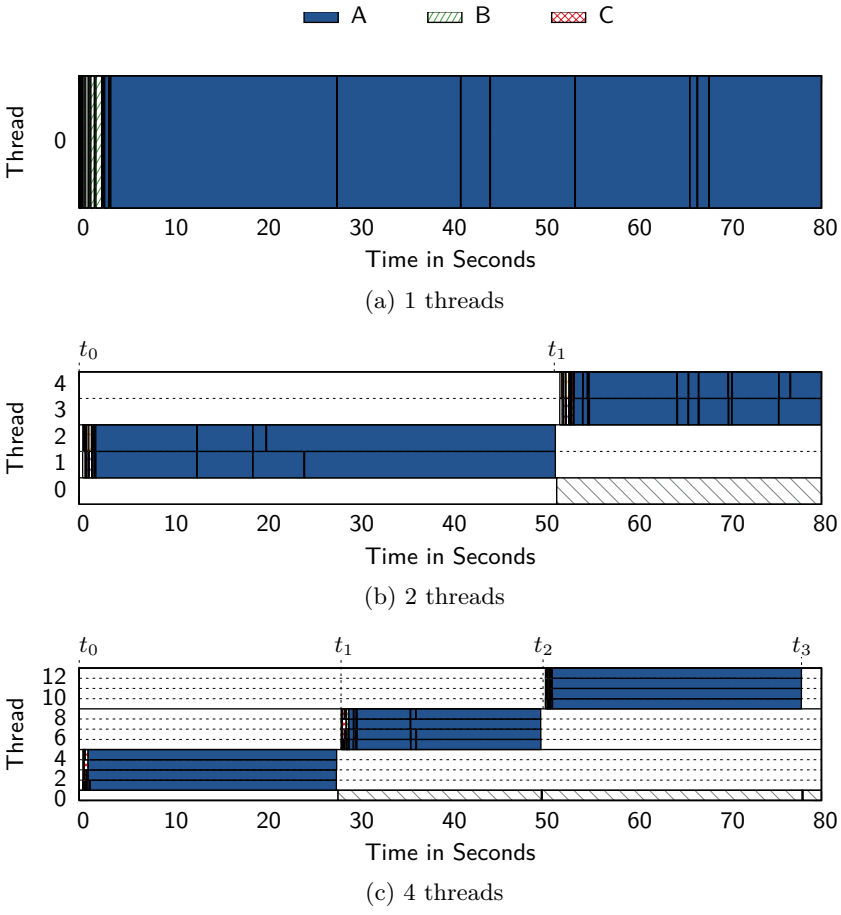
(a) 1 threads

(b) 2 threads

(c) 4 threads

Figure II.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).*

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 win-



(a) 1 threads

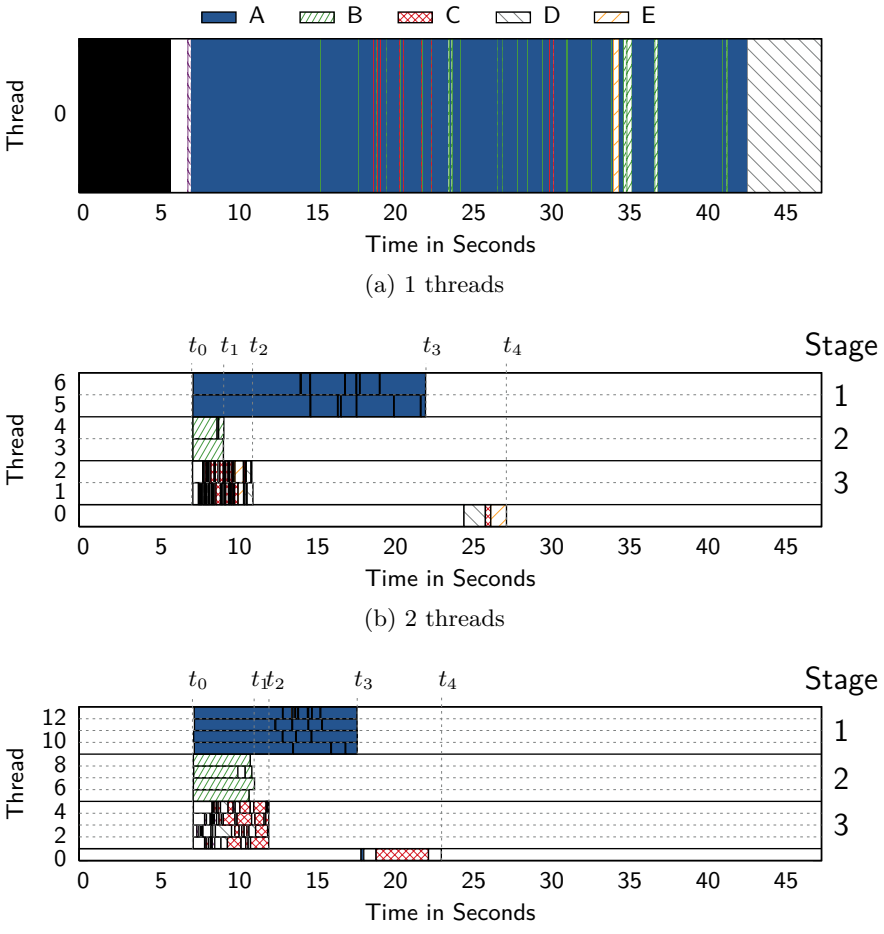
(b) 2 threads

(c) 4 threads

Figure II.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 II.8b and stops at t_1). Using thread private clusters for phase classification would therefore create a significant amount of duplicated phase ids.*

dows (colored in white) but the thread will take a long time to complete.

Data-parallel. Figure II.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



(a) 1 threads

(b) 2 threads

(c) 4 threads

Figure II.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).*

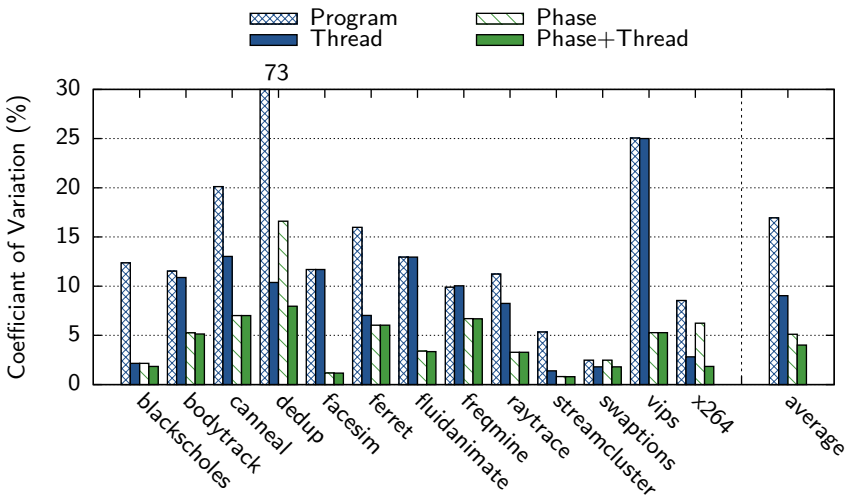


Figure II.10: The CPI CoV for the whole program (Program), per thread (Thread), per phase (Phase) and per phase within each thread (Phase+Thread) using 2 threads. *Dividing the execution into threads provides accurate results for blackscholes, streamcluster and for the two pipeline-parallel benchmarks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

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 II.8 shows how streamcluster creates new work-threads in each iteration. For example in Figure II.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 II.9 shows the detected program phases for dedup's whole execution. It executes different stages (phases) in different threads. For example, in Figure II.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 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 II.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 II.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 II.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.

II.5.2 Summary

The overall CPI variations and number of detected phases are lower for PARSEC compared to SPEC as seen in section II.4. However, the PARSEC benchmarks show a diverse set of phase behaviors which are important to understand when developing new runtime optimizations for parallel applications.

II.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,

[5]If the results from the NAS [3] benchmarks were to be included, the overall behavior would be more similar to that of facesim.

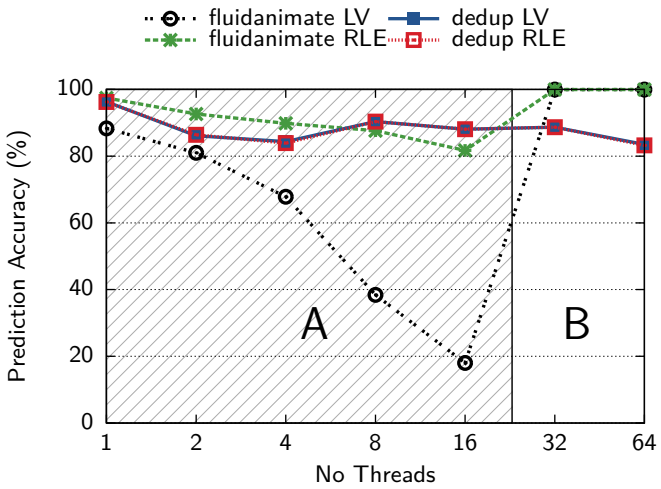


Figure II.11: Next window prediction accuracy for different number of threads for dedup (pipeline-parallel) and fluidanimate (data-parallel), using last value prediction (LV) and history based prediction with run length encoding (RLE). *The length of the phases shrinks (lower prediction accuracy for 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.*

but fluidanimate has shorter phases which makes it easier to analyze. Figure II.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 II.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 II.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.

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

[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).

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.

II.7 Related Work

Perelman et al. [35] extended SimPoint [41] to detect phases in parallel applications, and they used it to find architecture simulation points in the OpenMP version of the NAS [3] 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, Sherwood, and Calder [9, 8] 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. [19] extended hardware-based phase tracking [40, 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, 1] 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 [16, 21, 26], and divide the execution when the call stack changes depth (i.e., phase).

Bienia, Kumar, and Li [6] compared PARSEC with SPLASH-2 [43]. However, they only looked at aggregated values and focused on metrics related to multi-processors. Bhadauria, Weaver, and McKee [4] examined PARSEC using a range of different performance metrics on several multi-processors and Bhattacharjee et al. [5] characterized the TLB behavior.

II.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×) and larger variations in CPI (1.5×). 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.

II.9 References for Paper II

[1] Kartik K. Agaram et al. "Decomposing memory performance: data structures and phases". In: *Int. Symposium on Memory management*. 2006.

[2] Murali Annavaram et al. "The Fuzzy Correlation between Code and Performance Predictability". In: *Int. Symposium on Microarchitecture*. 2004.

[3] D. H. Bailey et al. "The NAS Parallel Benchmarks". In: *Int. Journal of Supercomputer Applications* (1991).

[4] Major Bhadauria, Vincent M. Weaver, and Sally A. McKee. "Understanding PARSEC Performance on Contemporary CMPs". In: *Int. Symposium on Workload Characterization*. 2009.

[5] Abhishek Bhattacharjee and Margaret Martonosi. "Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors". In: *Int. Conf. on Parallel Architectures and Compilation Techniques*. 2009.

[6] Christian Bienia, Sanjeev Kumar, and Kai Li. "PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors". In: *Int. Symposium on Workload Characterization*. 2008.

[7] Christian Bienia et al. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *Int. Conf. on Parallel Architectures and Compilation Techniques*. 2008.

[8] Michael Van Biesbrouck, Lieven Eeckhout, and Brad Calder. "Considering All Starting Points for Simultaneous Multithreading Simulation". In: *Int. Symposium on Performance Analysis of Systems and Software*. 2006.

[9] Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. "A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation". In: *Int. Symposium on Performance Analysis of Systems and Software*. 2004.

[10] B. Davies et al. *iPART : An Automated Phase Analysis and Recognition Tool*. Tech. rep. IR-TR-2004-1-iPART. Intel Corporation, 2004.

[11] Ashutosh S. Dhodapkar and James E. Smith. “Comparing Program Phase Detection Techniques”. In: *Int. Symposium on Microarchitecture*. 2003.

[12] Ashutosh S. Dhodapkar and James E. Smith. “Managing multi-configuration hardware via dynamic working set analysis”. In: *Int. Symposium on Computer Architecture*. 2002.

[13] Richard O. Duda, Peter E. Hart, and David G. Stork. “Pattern Classification”. In: 2nd ed. Wiley-Interscience, 2001. Chap. 10.11. On-line Clustering, pp. 559–565. ISBN: 0-471-05669-3.

[14] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. “Characterizing and Predicting Program Behavior and its Variability”. In: *Int. Conf. on Parallel Architecture and Compilation Techniques*. 2003.

[15] Lieven Eeckhout, John Sampson, and Brad Calder. “Exploiting Program Microarchitecture Independent Characteristics and Phase Behavior for Reduced Benchmark Suite Simulation”. In: *Int. Symposium on Workload Characterization*. 2005.

[16] Andy Georges et al. “Method-level phase behavior in java workloads”. In: *Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*. 2004.

[17] John L. Henning. “SPEC CPU2006 benchmark descriptions”. In: *SIGARCH Comput. Archit. News* (2006).

[18] *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Volume 3B: System Programming Guide. 30.4.4 Precise Event Based Sampling (PEBS). Intel Corporation. 2010.

[19] E. Ipek et al. “Dynamic program phase detection in distributed shared-memory multiprocessors”. In: *Int. Symposium on Parallel and Distributed Processing*. 2006.

[20] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. “Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management”. In: *Int. Symposium on Microarchitecture*. 2006.

[21] Jinpyo Kim et al. “Dynamic Code Region (DCR)-based Program Phase Tracking and Prediction for Dynamic Optimizations”. In: *Int. Conf. on High Performance Embedded Architectures and Compilers*. 2005.

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

[23] J. Lau et al. “Motivation for Variable Length Intervals and Hierarchical Phase Behavior”. In: *Int. Symposium on Performance Analysis of Systems and Software*. 2005.

[24] J. Lau et al. “Motivation for Variable Length Intervals and Hierarchical Phase Behavior”. In: *Int. Symposium on Performance Analysis of Systems and Software*. 2005.

[25] J. Lau et al. “The Strong correlation Between Code Signatures and Performance”. In: *Int. Symposium on Performance Analysis of Systems and Software*. 2005.

[26] Jeremy Lau, Erez Perelman, and Brad Calder. “Selecting Software Phase Markers with Code Structure Analysis”. In: *Int. Symposium on Code Generation and Optimization*. 2006.

[27] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. “Transition Phase Classification and Prediction”. In: *Int. Symposium on High-Performance Computer Architecture*. 2005.

[28] David Levinthal. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*. Tech. rep. Version 1.0. Intel Corporation, 2009.

[29] *Linux perf_events*. URL: `Linux/include/linux/perf\_event.h`.

[30] Ke Meng et al. “Multi-optimization power management for chip multiprocessors”. In: *Int. Conf. on Parallel Architectures and Compilation Techniques*. 2008.

[31] Priya Nagpurkar, Chandra Krintz, and Timothy Sherwood. “Phase-Aware Remote Profiling”. In: *Int. Symposium on Code Generation and Optimization*. 2005.

[32] Priya Nagpurkar et al. “Online Phase Detection Algorithms”. In: *Int. Symposium on Code Generation and Optimization*. 2006.

[33] Nitzan Peleg and Bilha Mendelson. “Detecting Change in Program Behavior for Adaptive Optimization”. In: *Int. Conf. on Parallel Architecture and Compilation Techniques*. 2007.

[34] Erez Perelman, Greg Hamerly, and Brad Calder. “Picking Statistically Valid and Early Simulation Points”. In: *Int. Conf. on Parallel Architecture and Compilation Technique*. 2003.

[35] Erez Perelman et al. “Detecting phases in parallel applications on shared memory architectures”. In: *Int. Symposium on Parallel and Distributed Processing*. 2006.

[36] Andreas Sembrant, David Black-Schaffer, and Erik Hagersten. “Phase Guided Profiling for Fast Cache Modeling”. In: *Int. Symposium on Code Generation and Optimization*. 2012.

[37] Andreas Sembrant, David Eklov, and Erik Hagersten. “Efficient Software-based Online Phase Classification”. In: *Int. Symposium on Workload Characterization*. 2011.

[38] Xipeng Shen, Yutao Zhong, and Chen 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] Timothy Sherwood, Suleyman Sair, and Brad Calder. “Phase tracking and prediction”. In: *Int. Symposium on Computer Architecture*. 2003.

[41] Timothy Sherwood et al. “Automatically characterizing large scale program behavior”. In: *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. 2002.

[42] Tyler Sondag and Hridesh Rajan. “Phase-based tuning for better utilization of performance-asymmetric multicore processors.” In: *Int. Symposium on Code Generation and Optimization*. 2011.

[43] Steven Cameron Woo et al. “The SPLASH-2 programs: characterization and methodological considerations”. In: *Int. Symposium on Computer Architecture*. 1995.

Paper III

Paper III

Phase Guided Profiling for Fast Cache Modeling

Andreas Sembrant, David Black-Schaffer, and Erik Hagersten

In Proceeding of the
International Symposium on Code Generation and Optimization
San Jose, California, USA, March 31 - April 4, 2012

Phase Guided Profiling for Fast Cache Modeling

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

Statistical cache models are powerful tools for understanding application behavior as a function of cache allocation. However, previous techniques have modeled only the average application behavior, which hides the effect of program variations over time. Without detailed time-based information, transient behavior, such as exceeding bandwidth or cache capacity, may be missed. Yet these events, while short, often play a disproportionate role and are critical to understanding program behavior.

In this work we extend earlier techniques to incorporate program phase information when collecting runtime profiling data. This allows us to model an application's cache miss ratio as a function of its cache allocation over time. To reduce overhead and improve accuracy we use online phase detection and phase-guided profiling. The phase-guided profiling reduces overhead by more intelligently selecting portions of the application to sample, while accuracy is improved by combining samples from different instances of the same phase.

The result is a new technique that accurately models the time-varying behavior of an application's miss ratio as a function of its cache allocation on modern hardware. By leveraging phase-guided profiling, this work both improves on the accuracy of previous techniques and reduces the overhead.

III.1 Introduction

The goal of this work is to develop and explore methods for understanding a program's cache behavior *over time* and as a function of its *cache allocation*. Such information is important for understanding performance [23], resource sharing [13, 7], and scheduling [14]. In particular, the ability to analyze a program's behavior as a function of its cache allocation is essential for modern systems with shared caches where the

cache allocation can change dynamically. This requirement makes it difficult to use data from hardware performance counters, as they only provide information for one particular cache allocation.

The low overhead statistical cache model, StatCache, developed by Berg and Hagersten [3, 5], can estimate the miss ratio for caches of arbitrary size. It answers the question: what is an application's miss ratio if it receives x amount of cache? StatCache has been used to estimate shared miss ratios for multi-threaded applications [4] and co-scheduled applications [13], and forms the basis of a commercial code optimization product [37].

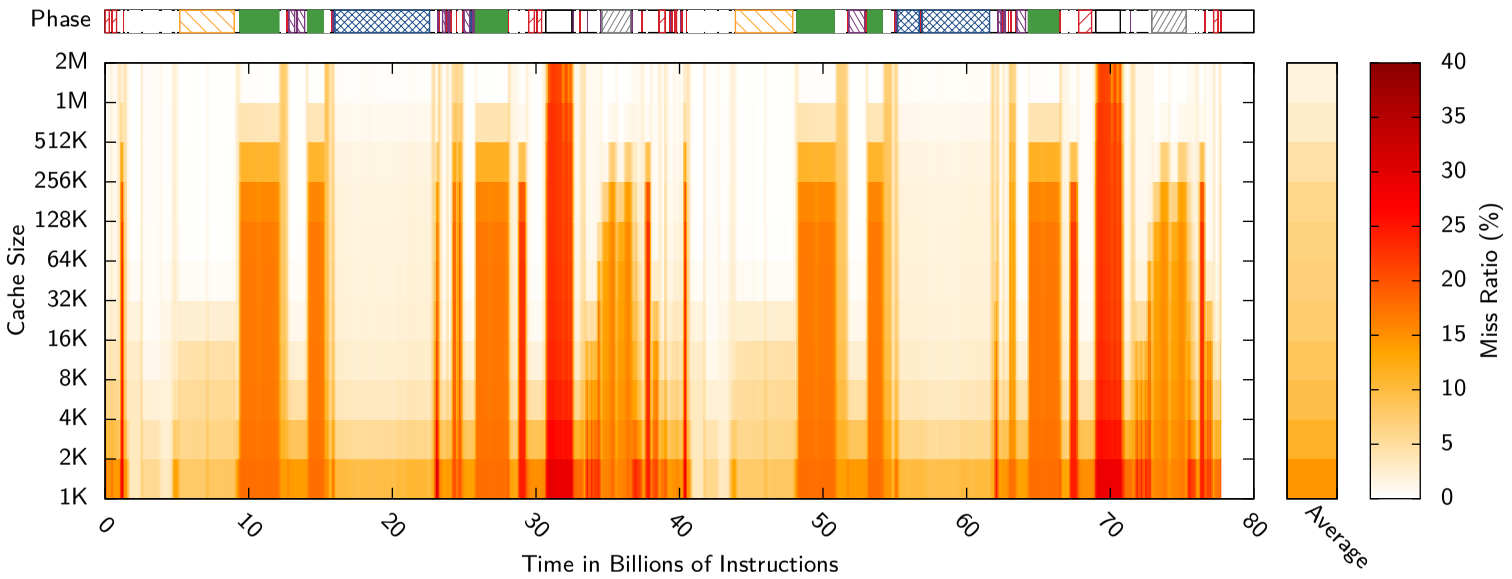
However, these existing models only report the application's average miss ratio, which can be misleading. Consider, for example, an application whose miss ratio is high enough to exceed the system bandwidth for a short portion of its execution. In such an application, the average miss ratio would fail to indicate that the application is at all bandwidth bound.

The simplest way to extend these methods to handle program phases is to divide the program execution into many *windows* and profile each window. This approach has the downside of a significant increase in overhead from having to sample all portions of the application's execution. To combat this, *periodic* profiling can be used, wherein only a randomly selected subset of all windows are profiled. Unfortunately, the number of profiled windows must still be high to capture short application phases.

A more intelligent solution is to use *phase-guided* profiling [30, 26]. In this approach, a phase-detection algorithm is used to select only a small part of each phase to profile, and this data is then used for subsequent instances of the same phase. This minimizes the number of profiled windows by avoiding redundantly sampling windows from the same phase.

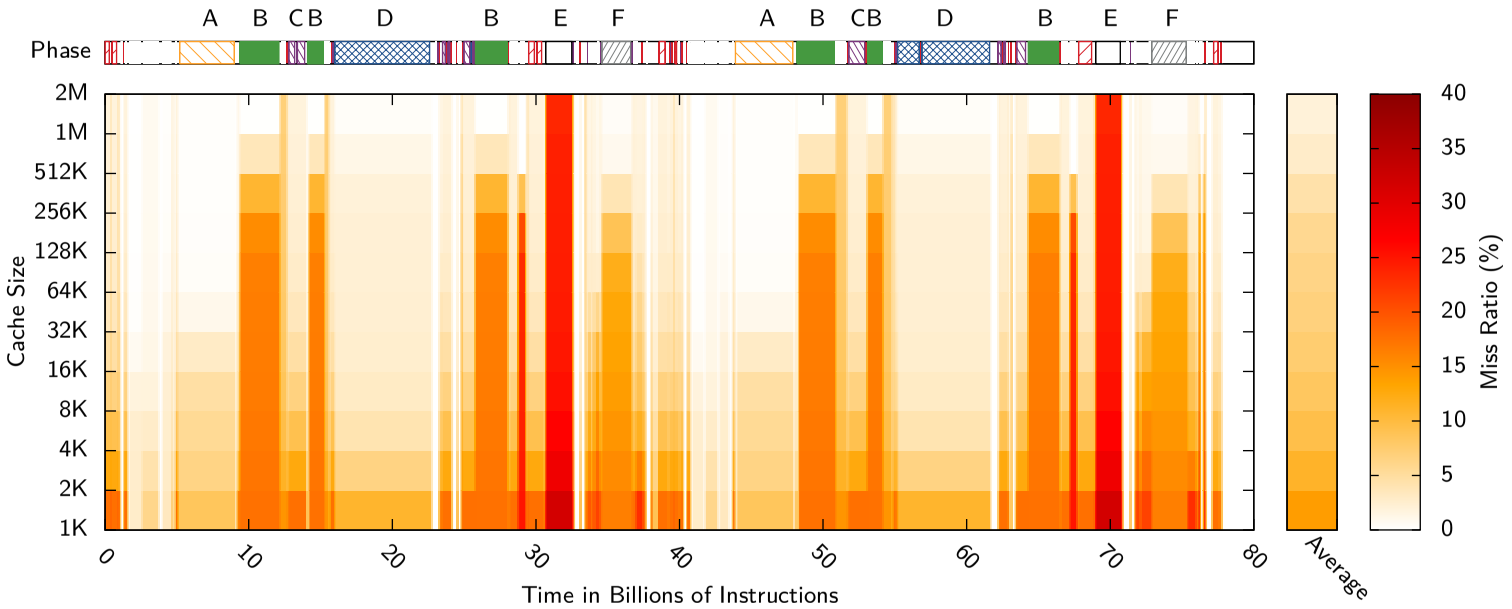
For such an approach to be generally applicable, it must have the following properties: 1) It should not require custom hardware support; 2) It should have minimal runtime overhead without loss of accuracy and fidelity; 3) It should be transparent and non-intrusive (e.g., require no recompilation of the analyzed program and work with dynamically generated code); 4) It should be architecturally independent (e.g., not affected by system load and able to model different cache sizes), and, finally; 5) It should be fully automatic (e.g., users should not have to adjust settings for each application).

To accomplish this, we leverage the ScarPhase (Sample-based Classification and Analysis for Runtime Phases) library developed during our previous work with phase classification [30]. This provides us with low-overhead (2%) runtime detection of program phases. We then combine



(a) gcc/166 miss ratio from reference simulation.

Figure III.1: Miss ratio (intensity) as a function of time (x-axis) and cache allocation (y-axis) for the whole execution of gcc/166 on a Nehalem machine. The average miss ratio for the whole execution is shown on the right. The detected execution phases are shown above, with shorter phases shown in white for clarity. The top figure (III.1a) shows results from a reference simulation and the bottom (III.1b) from online profiling.



(b) gcc/166 miss ratio from online phase-guided profiling with statistical cache modeling.
(32.5% overhead on a Nehalem system.)

Figure III.1: *The transient periods of very high miss ratio across all cache sizes (phase E) are of particular interest as they are not visible in the average data. This emphasizes the importance of time-based information for understanding application behavior.*

this phase information with the StatCache [3] statistical cache model to accurately model application cache behavior as a function of time and allocated cache.

The main contributions of this paper are:

- A method for accurately modeling cache behavior (miss ratio as a function of cache allocation) over time.
- An efficient method for capturing program cache behavior on modern hardware by integrating the StatCache cache model and the ScarPhase phase detection library.
- A comparison with previous statistical cache modeling methods demonstrating improved accuracy (39%) and efficiency (6×).
- An analysis of the impact of different types of intra-phase variations on phase-guided memory profiling.

III.2 Cache Behavior over Time

An application's cache behavior, in this case its miss ratio, varies due to both program phases and changes in cache allocation. To illustrate this, Figures III.1 and III.2 plot the miss ratio (intensity) over time (x-axis) as a function of cache size (y-axis), for the complete execution of the gcc/166 and bzip2/chicken benchmarks, respectively. The darker the points, the higher the miss ratio. The y-axis (cache size) indicates how the application's miss ratio is affected by its cache allocation. The x-axis (time) shows the intrinsic phase behavior of the application. The vertical bar marked "Average" on the right shows the application's overall average miss ratio as a function of cache size. And, finally, the bars above the graph indicate the phases detected by ScarPhase, with smaller phases grouped together in white for clarity.

The top figures (III.1a and III.2a) show reference results from a simulation using the Pin [6] instrumentation toolkit and the Dinero [11] cache simulator. The lower figures (III.1b and III.2b) show the results of this work. Data in these last two figures was generated by the StatCache model applied to phase-guided profiling data captured at runtime on our Nehalem machine.

The benefits of time-based information are clearly visible in Figure III.1. While the graph shows that there are two periods in gcc/166's execution with a very high miss ratio at 2MB of allocated cache (phase

E at 32 and 70 billion instructions), the overall average miss ratio appears far less severe. With the more fine-grained phase information, the correct portion of the application can be targeted for optimization.

From this information we can also see the limitations of defining phases based on hardware-specific information, such as performance counters. For example, if miss ratio was used to define phases, a machine with 2MB or more of cache would group the first 80 billion instructions of bzip2/chicken into one phase as the miss ratio is constant. (See the red box in Figure III.2b.) However, if the application's cache allocation were decreased, due to resource sharing, for example, its behavior would change, thereby revealing different phases. This demonstrates the importance of finding phases that are architecturally independent [29] properties of the application.

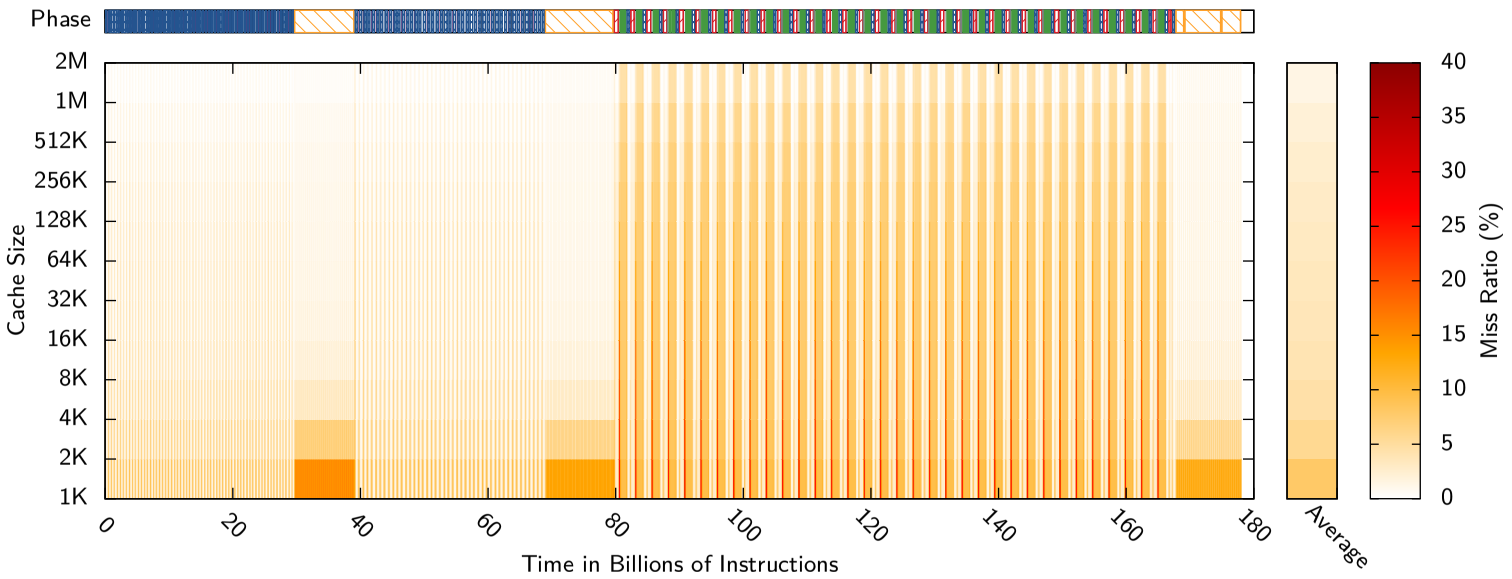
These examples show how important it is to consider both the intrinsic program phase behavior as well as the impact of program cache allocation when examining application behavior. With this more detailed information we can analyze how various runtime optimizations [9, 18] and scheduling decisions [14, 35] will affect the cache performance. For example, migrating gcc/166 to a core with a smaller cache for phase D could potentially save energy without sacrificing performance. However, trying to migrate bzip2/chicken between a large-cache core for phase B and a small-cache core for phases C and D would entail many more relocations and might not be beneficial.

III.3 Statistical Cache Modeling

StatCache [3, 5] is a low overhead statistical cache model. It can estimate the miss ratio of random replacement caches of arbitrary sizes. In this section we give an overview of the model and discuss how program phase behavior affects its accuracy.

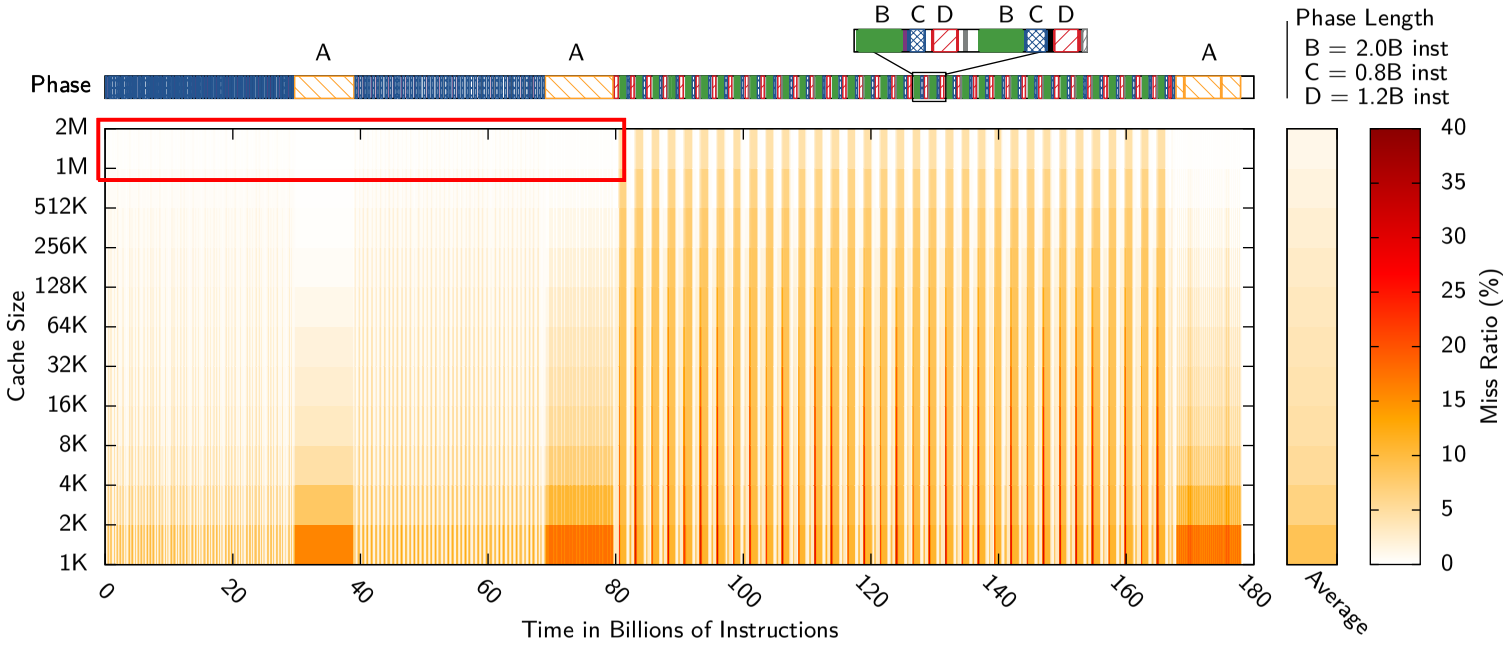
III.3.1 Reuse Distance

The input to the StatCache model is cache line reuse distance data. A reuse distance is defined to be the number of memory accesses between two accesses to the same cache line. For example, if the processor first accesses cache line A, then B and C, and finally A again, the reuse distance of the second access to A would be two. It is important to note that reuse distance counts *all* memory accesses. This is different from stack distance [25] where only the number of *unique* memory accesses are counted. As a result, measuring reuse distance requires far less bookkeeping.



(a) bzip2/chicken miss ratio from reference simulation.

Figure III.2: Miss ratio (intensity) as a function of time (x-axis) and cache allocation (y-axis) for the whole execution of bzip2/chicken on a Nehalem machine. The average miss ratio for the whole execution is shown on the right. The detected execution phases are shown above, with shorter phases shown in white for clarity. The top figure (III.2a) shows results from a reference simulation and the bottom (III.2b) from online profiling.



(b) bzip2/chicken miss ratio from online phase-guided profiling with statistical cache modeling. (27.4% overhead on a Nehalem system.)

Figure III.2: *The boxed area from 0 to 80B instructions highlights the importance of using hardware-independent information for determining phases: when run with 1MB or more of cache allocation, bzip2/chicken appears to have a single phase up to 80B instructions based on cache miss ratio. However, at lower allocations, or using hardware-independent metrics, distinct phases can be clearly seen.*

III.3.2 The StatCache Cache Model

The reuse distance distribution can be transformed into a miss ratio using the StatCache [3, 5] cache model. StatCache first sorts the reuse distances of the memory accesses into buckets, h_i, where h_i is the number of memory accesses with a reuse distance of i. Then, the following equation is solved for the miss ratio R:

$$R \cdot N = h_1 f(R) + h_2 f(2R) + h_3 f(3R) + \cdots \tag{III.1}$$

N is the number of reuse distance samples, i.e., $N = h_1 + h_2 + h_3 + \cdots$, and $f(n)$ is a function that gives the probability that a cache line has been evicted from the cache if we know that it was in the cache n cache misses ago. With random replacement the function $f(n)$ is:

$$f(n) = 1 - (1 - 1/L)^n \tag{III.2}$$

where L is the number of cache lines in the cache. The cache size is L times the cache line size. We can then model caches of arbitrary sizes by changing the L. The StatCache model can be readily extended to model LRU caches (StatStack [12]) without changing the input data.

III.3.3 Program Phases

The StatCache model works very well for single phase applications and its accuracy improves with the number of samples. Indeed, Equation III.1 assumes a constant miss ratio across the reuse distance samples. If the behavior is constant for the application, a *phase oblivious* overall miss ratio can be determined by simply applying the model to all samples at once.

However, as we observed in the previous section, the miss ratio can vary significantly over time. As Berg and Hagersten [3, 5] had no means to detect phases, they instead gathered samples in short *bursts*, where each burst was short enough for the miss ratio to remain approximately constant. They then applied the model to each burst separately and averaged the model output to determine the overall miss ratio. This method improves accuracy by ensuring that the miss ratio is approximately constant across the samples given to the model.

The work presented here is *phase aware*, and groups samples within the same phase together. The model is then applied to all samples from each phase separately. The application's overall miss ratio is then the weighted average of the phases. This approach improves accuracy as the miss ratio is far more constant within phases than across them, and by combining samples across phases, the model has more samples to work with at each time.

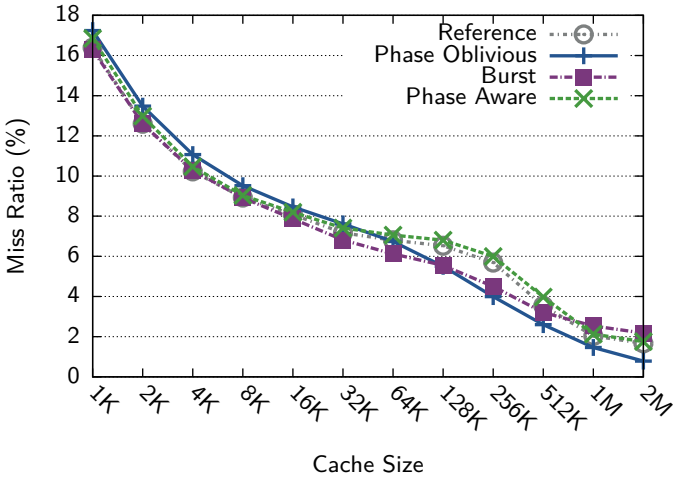


Figure III.3: The average miss ratio for gcc/166. All curves uses the same reuse distance data. *Phase Oblivious* applies the model to all reuse distances from the execution together. *Burst* applies the model to groups of reuse distances together, and then averages the model results of all the bursts. *Phase Aware* groups reuse distances from the same phase together and applies the model for each phase. The result is then the weighted average of all the phases. The figure shows that grouping reuse distances from the same phase together has the best accuracy.

Figure III.3 compares these three methods for gcc/166. Each method uses the same reuse distance samples. The *phase oblivious* approach applies the model to the most samples (all of them together), but incorrectly assumes that the underlying miss ratio is constant across all samples. As a result it has the worst accuracy. Both *burst* and *phase aware* apply the model to groups of samples taken from periods with a reasonably constant miss ratio, but the *phase aware* approach is able to group more samples together for the model, and thereby produce more accurate results.

III.3.4 Sampler Implementation and Overhead

We have implemented a reuse distance sampler on an Intel Xeon E5620 (Nehalem) machine to provide data to the StatCache model. To minimize the overhead, we use hardware performance counters [16] and page protection to sample and monitor reuses.

For the StatCache model to work, it is important that all memory accesses have the same probability of being sampled. We therefore use the executed loads and stores counters to interrupt the program exe-

cution at random (exponentially distributed) intervals. However, when these interrupts occur, a context-dependent number of extra instructions are executed. To avoid biasing the results with this "skid" [17], the counters are set up to interrupt before the target access. After the interrupt, the execution is single-stepped to the desired access. At this point the loads and stores counters are recorded for the access's pending reuse, and page protection is turned on for the access's page.

Execution then continues until a page fault occurs. If the memory access that caused the page fault belongs to a pending reuse, the loads and stores counters are read and the resulting reuse distance recorded. Otherwise, it was a false positive, i.e., the page protection was turned on because of another cache line that resides on the same page. In the latter case, the page protection is temporarily turned off and the execution is single-stepped past the access, before turning the page protection on again.

In this reuse sampler there are two parts to the overhead. First, the application must be single-stepped to the target access. Depending on the length of the skid, this can entail many context switches. Second, it can be equally time consuming to handle page faults, especially when the number of false positives are high. As both of these overheads are directly related to the number of samples required, it is clearly important to intelligently choose when to sample.

III.4 Phase Guided Profiling

Phase-guided profiling is a method to reduce the overhead of profiling without sacrificing accuracy, by taking advantage of the (nominally) uniform behavior of each program phase. The idea is to only profile a small part of each phase, and then use that profile for all instances of the same phase. There are two benefits to this approach. First, it removes redundant profiling as only a minimum part of each phase is profiled. Second, it automatically adapts to the application's characteristics, thereby eliminating the need to adjust profiling parameters for each application and data set.

III.4.1 Detecting Program Phases

We use the ScarPhase [30] library to detect and classify phases. ScarPhase is an execution history based, low overhead (less than 2%), online phase detection library. Because it is based on the application's execution history, it detects hardware independent phases [34, 28]. Such phases can be readily missed by performance counter based phase detection, as shown in Figure III.2b.

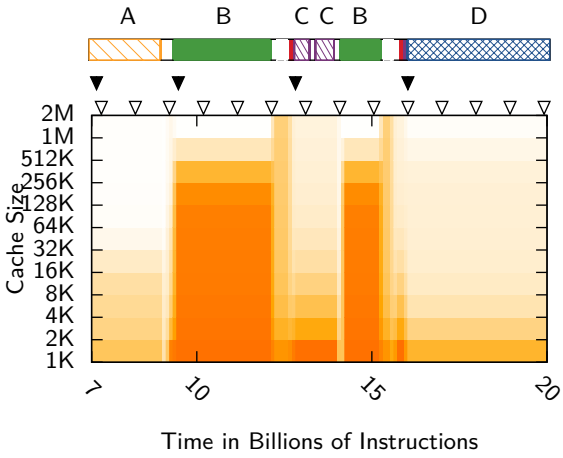


Figure III.4: Profiling overhead for gcc/166 with phase-guided profiling (black triangles) and periodic profiling (white triangles). The figure shows that periodic profiling requires more samples to achieve the same coverage.

To detect phases, ScarPhase monitors executed code, based on the observation that changes in executed code reflect changes in many different metrics [32, 34, 8, 33, 19]. To accomplish this, execution is divided into non-overlapping windows. During each window, hardware performance counters are used to sample conditional branches using Intel PEBS [24, 16]. 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) [32] 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, such as leader-follower [10]. Windows with similar vectors are then grouped into the same cluster and considered to belong to the same phase.

III.4.2 Phase Guided Profiling

The simplest approach to phase-guided profiling is to only profile one window from each phase, and to use that profile for all other instances of the same phase. This way, only a small part of each phase is profiled, thereby lowering overhead, and, if the behavior within the phase is uniform, the accuracy will not suffer. As a result, the overhead will

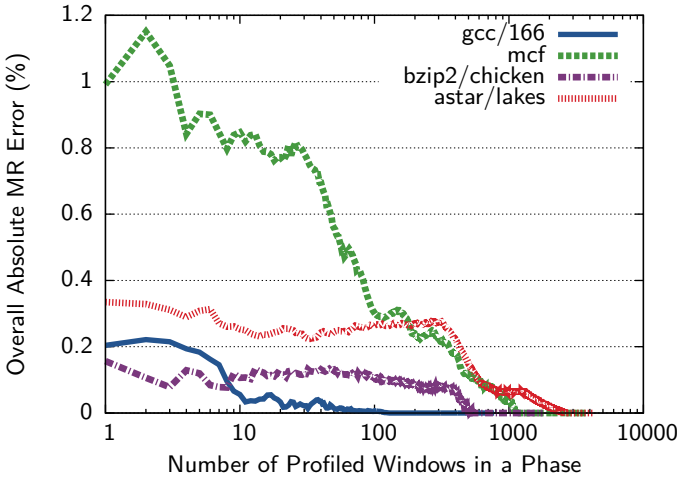


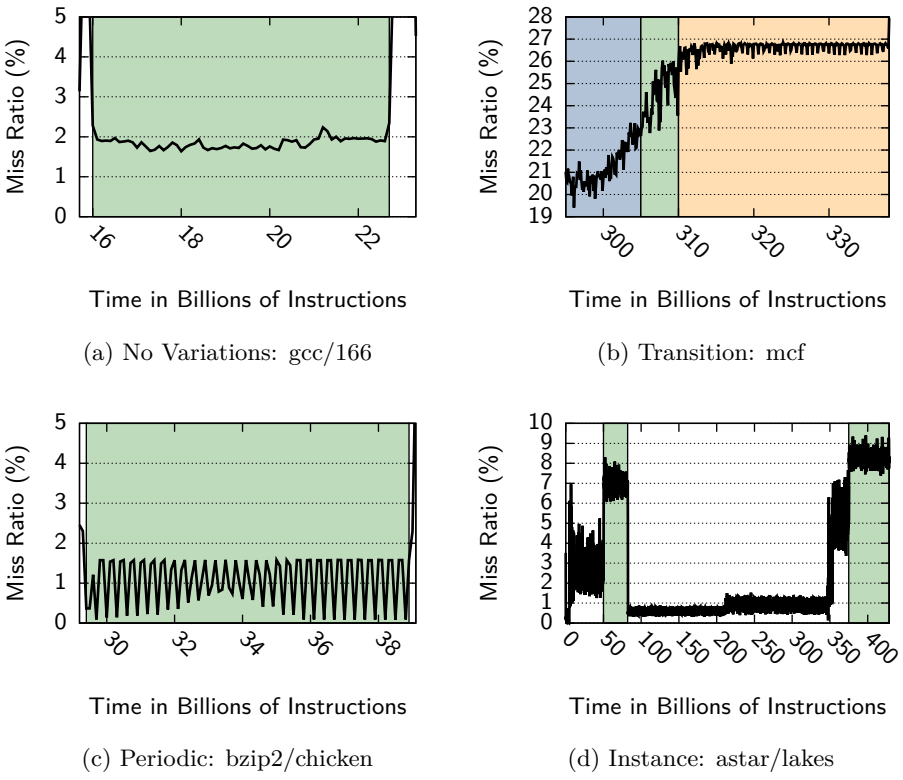
Figure III.5: *Choosing windows in order.* Overall Absolute Miss Ratio Error as a function of number of profiled windows in each phase. The windows are selected in the order they appear. The error graphs may show non-monotonic behavior due to the way in which the error metric is calculated: averaging across the phases may cause errors from one phase's miss ratio to cancel out with another's.

be proportional to the number of phases in the application, and the profiling will automatically adapt to the application's requirements.

To illustrate how phase-guided profiling works, we have zoomed in on a short part of gcc/166's execution in Figure III.4. The black triangles show where phase-guided profiling decides to profile, and the white triangles show the same for periodic profiling. Phase-guided profiling places the samples at the beginning of each phase. Periodic profiling, on the other hand, may profile the same phase more than once. Furthermore, the period between the profiles must be short enough to catch all phases. If an application has a mix of short and long phases, the profiling period must be set for the shortest phase to accurately capture the application's behavior. This results in a high overhead and requires the user to adjust the profiler to the application.

ScarPhase returns the phase ID of the just-executed window and a prediction for the next window [33, 22]. Since the phase ID is only known after the window has been executed, we need to rely on the prediction[1]. If the predicted phase has not been profiled, we start to sample reuse

[1]Most phases span several windows and we only need to profile when we are sure we are in the correct phase. A mis-prediction is thus very uncommon. Furthermore, a mis-prediction is unlikely to affect the accuracy since the phase will be profiled later.



(a) No Variations: gcc/166

(b) Transition: mcf

(c) Periodic: bzip2/chicken

(d) Instance: astar/lakes

Figure III.6: Intra-phase variation in miss ratio for Spec2006. Phases shown with shading. While most phases show little intra-phase variation, as in gcc/166 (III.6a), the three intra-phase variations identified above explain the improved error characteristics (Figure III.7) of randomly selecting windows to profile. Both Transitions (III.6b) and Periodic (III.6c) are artifacts of the tradeoff between phase size and the number of phases. The Instance Variations (III.6d), however, represent data-dependent changes in behavior for the same code path.

distances, otherwise, we turn off the profiler and do not sample.

III.4.3 Intra-Phase Variations

While the phases detected by ScarPhase have reasonably constant behavior within each phase, some applications exhibit intra-phase variation [32, 31]. To illustrate this, Figure III.5 plots the *Overall Absolute Miss Ratio Error*[2] for four applications as a function of the number

[2]The Overall Absolute Miss Ratio Error for Figures III.5 and III.7 is defined as the absolute error from the reference simulation across all cache sizes, as shown in

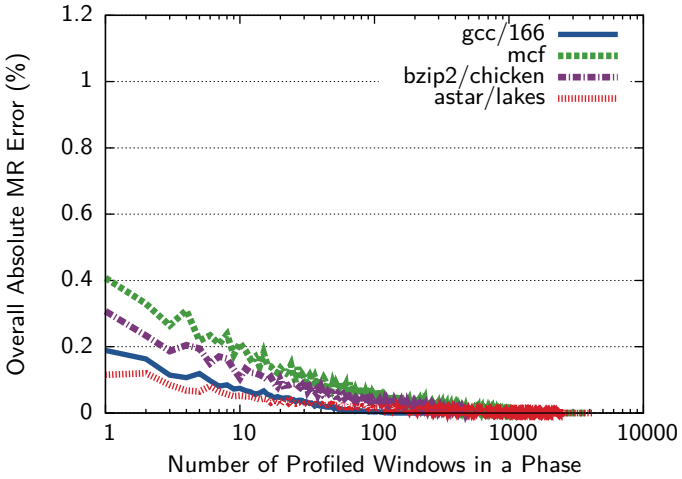


Figure III.7: *Choosing windows randomly.* Overall Absolute Miss Ratio Error as a function of number of profiled windows in each phase. The windows are randomly selected within each phase.

of windows profiled in each phase. For example, at 10 on the x-axis, the y-axis shows the error if only the first ten windows of each phase are profiled. If there were no intra-phase variation, the error would be constant. However, this is clearly not the case. Indeed, we observed three different types of intra-phase variation: transition, periodic, and instance. These are illustrated in Figure III.6 and discussed below.

No variations. Figure III.6a shows the miss ratio over time for gcc/166's phase D in Figure III.1. This is a typical phase for gcc/166 with very little intra-phase variation. Only a small part of each phase needs to be profiled, and we see that the error drops rapidly after seven windows in Figure III.5.

Transition variations. In Figure III.6b, mcf can be seen to have a long transition phase where the behavior slowly changes from one phase to another. The Figure shows how the miss ratio slowly increases over time between the two phases. During such a transition the execution will be divided into several phases with behavior from both the start and the end phase. The more windows we profile, the closer we will get to the average by including more behavior from the end phase. We therefore see a steady decrease in error for mcf in Figure III.5 when the number of profiled windows increases.

Periodic variations. Figure III.6c shows the miss ratio over time for bzip2/chicken's phase A in Figure III.2. The behavior is highly peri-

Figure III.3.

odic and changes rapidly. It is actually caused by two sub-phases whose vectors are not different enough to create two separate clusters. This tradeoff between uniformity and the number of phases is a limitation of clustering algorithms. If the settings are too sensitive we will identify more phases that necessary, and if they are too insensitive we combine the wrong sub-phases.

Instance variations. In Figure III.6d we can see that astar/lakes suffers from a different problem. The two separate instances of the same phase have different average miss ratios. The vectors for both instances are nearly identical but the cache behavior is different. This can happen when two instances of the same phase operate on different input data. This is the reason why so many windows must be profiled for the error to start to decrease in Figure III.5: all windows in the first instance of the phase must be profiled before the second instance can be included in the results. There has been a significant amount of research [31, 2, 21] discussing how changes in the code path are correlated to changes in other metrics.

The intra-phase variations identified above have a significant effect on the accuracy of this method. To make an accurate estimate of the behavior of a phase, it is therefore important to consider all instances of the phase. Figure III.7 shows the same metric as in Figure III.5, but instead of selecting only the first windows, the windows are randomly selected from the whole phase. For example, when x is ten, the average is calculated from ten randomly selected windows from the phase. If a phase is less than ten windows, the whole phase is profiled. The error starts to decrease rapidly for all applications compared to taking the windows in order. It is therefore important to spread samples throughout a phase.

III.4.4 Phase Sampling Implementation

To handle intra-phase variations, we try to profile several windows spread throughout the phase. However, we do not know the length of the phase in advance. We therefore start with a short period to catch the shorter phases, and increase the period with the number of profiled windows until an upper limit is reached. Specifically, we start sampling windows with an exponential distribution, and increase the period by a factor of two after each window until we reach an upper limit. In this way, we can reliably profile both short and long phases while maintaining a good distribution of samples.

It is worth noting that the runtime overhead is proportional to the number of sampled reuse distances. This is different from traditional profiling and simulation where the overhead comes from number of ex-

ecuted instructions. This has two implications for this work. First, spreading out profiling windows across an application's execution time does not increase overhead. This is because the overhead is per sample, regardless of when they are taken, and instructions executed between samples run at native speed. Second, because we capture many profile windows, we are less sensitive to selecting optimal windows [34, 26].

For reference, the data collection and modeling for Figures III.1b and III.2b took minutes to execute, while the simulation to produce the reference results in Figures III.1a and III.2a took days.

III.5 Evaluation

In this section we evaluate and compare the accuracy and performance of StatCache with periodic profiling and phase-guided profiling. We implemented periodic profiling by periodically[3] selecting windows to profile. The model was then independently applied to each window. The behavior over time was then approximated by observing how the behavior changes between the profiled windows. Phase-guided profiling used the ScarPhase library as discussed in Section III.4. The memory reuse data was captured online using the memory reuse sampler described in Section III.3.4. All benchmarks were run from start to completion with their reference input on a Intel Xeon E5620 (Nehalem) system. Because the random nature of the sampling, we average the data from 5 runs.

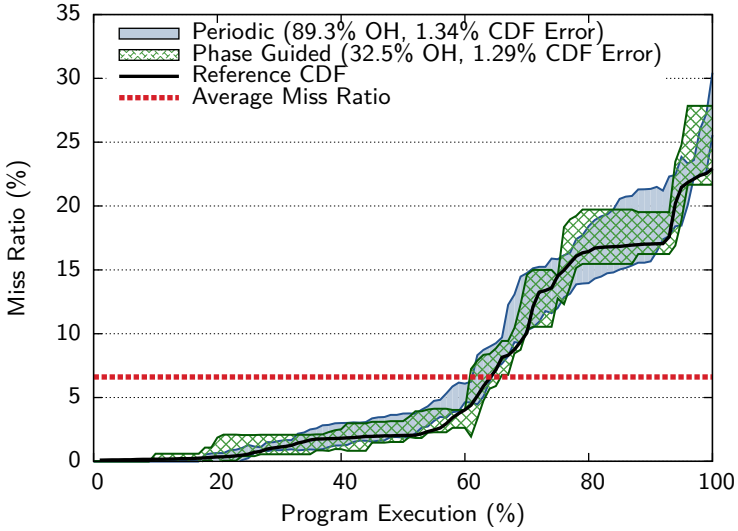
To create the reference data, we simulated the cache behavior for the whole execution using the Pin [6] instrumentation toolkit and the Dinero [11] cache simulator. Pin was used to divide the execution into windows and extract a memory reference trace that was sent to Dinero. After each window, the miss ratio for the window was extracted from the Dinero simulation.

We chose the eight applications from SPEC 2006 [15] with the most interesting phase behavior (astar/lakes, bzip2/chicken, bwaves, dealii, gcc/166, mcf, perl/splitmail and xalan) and simulated and modeled each for twelve cache sizes from 1KB to 2MB. The cache sizes were chosen to cover the most interesting changes in cache behavior for the benchmarks.

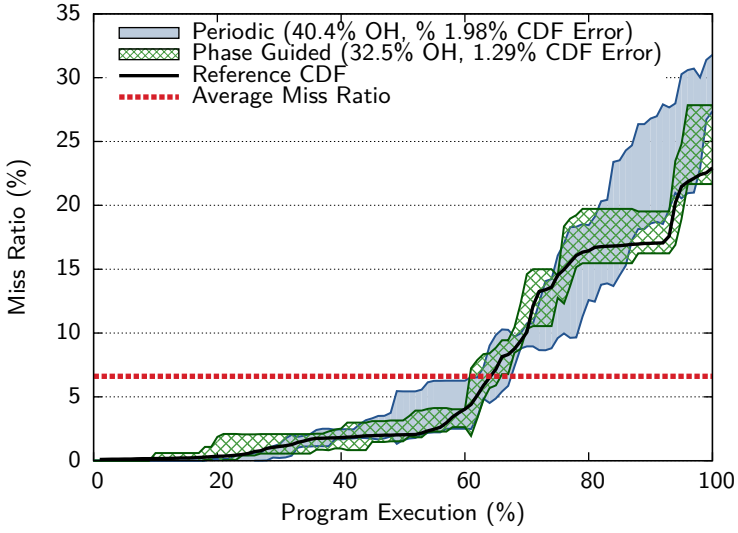
III.5.1 Measuring Errors Over Time: CDF Error

To evaluate error as a function of time, we define the *CDF Error*. This metric uses the cumulative distribution function (CDF) of the application's miss ratio, and is defined as the absolute difference between

[3]To avoid periodic behavior, the sampled windows are selected at random from an exponential distribution with a fixed period.



(a) *Similar accuracy, different overhead.* With similar accuracy, the phase-guided method has significantly lower overhead.



(b) *Different accuracy, similar overhead.* With similar overhead, the phase-guided method has significantly better accuracy.

Figure III.8: The CDF function for gcc/166's miss ratio for a single cache size of 64kB. The shaded areas indicate the standard deviation of the periodic and phase-guided methods. Their difference from the reference indicates the error. The overall average is shown as a straight line.

the reference CDF and the estimated CDF. The final CDF Error is the average of the individual CDF Errors across the target cache sizes.

The reference CDF for gcc/166's miss ratio is shown in Figure III.8a for a 64kB cache allocation. The x-axis shows the time in percent of the execution and the y-axis shows the miss ratio. We interpret the Figure as follows: x percent of the execution has a miss ratio below or equal to y. For example, 60% of the execution has a miss ratio below 5%.

The CDF can also give valuable insight into application behavior. For example, if gcc/166 hits the bandwidth limit when it has a miss ratio above 20%, the average would indicate that gcc never hits the limit, while the CDF shows that 7% of the execution would be bandwidth bound.

III.5.2 Sampling Parameters

We chose parameters for the window size and sample rate for periodic and phase-guided profiling to produce similar accuracy on gcc/166. (The exact settings and details on the selection process are found in the appendix.) This benchmark was chosen as the baseline because it has the highest number of phases (most difficult to accurately model) and a short execution (least chance to make up for missed phases).

The results of choosing settings to produce similar accuracy for gcc/166 can be seen in Figure III.8a. The shaded areas are the average miss ratio CDF +/- one standard deviation. The data shows the CDF for both the periodic and phase-guided methods, as well as the reference simulation. The smaller the shaded area and the closer it is to the reference the better the accuracy. In this graph both the periodic and the phase-guided methods show similar accuracy (1.34% and 1.29%, respectively), as expected. However, to obtain this degree of accuracy, the periodic method imposes an overhead of 89.3% compared to 32.5% for the phase-guided approach.

To compare the accuracy with similar overhead, we changed the parameters such that the periodic and phase-guided methods would have similar overhead and re-ran the experiment. The results of choosing settings to produce similar overhead for gcc/166 can be seen in Figure III.8b. As expected, overheads are similar (40.4% for periodic and 32.5% for phase-guided), but the accuracy (1.98% and 1.29%, respectively) and variance are significantly worse for the periodic sampling. For the subsequent evaluation we use the parameters that produce similar accuracy on gcc/166, as described in the appendix.

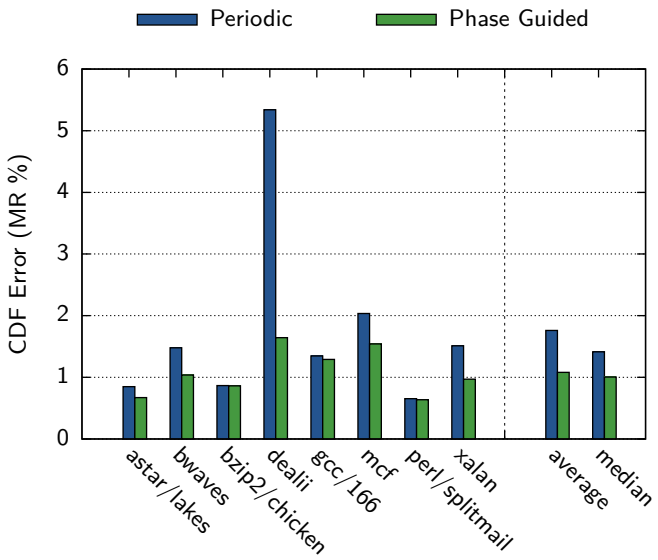


Figure III.9: The CDF Error for periodic profiling (average 1.76%) and phase-guided profiling (average 1.08%).

III.5.3 Accuracy: Error

Figure III.9 presents the CDF error for the benchmarks. The CDF error for the periodic and phase-guided profiling are very similar for gcc/166 since we chose the settings for this application. On average, the error is 1.08% and 1.76% for phase-guided profiling and periodic profiling, respectively, meaning that the distance between the reference and the estimated value in Figure III.8a is on average off by one percent for phase-guided profiling.

Despite using fewer samples, phase-guided profiling has a better accuracy. There are two reasons for this. First, phase-guided profiling can combine reuse distances from several windows belonging to the same phase which reduces the modeling error. Second, phase-guided profiling is better at distributing the samples over the execution: it forces shorter phases to be profiled which would otherwise have been missed. The profile thus represents a larger portion of the execution.

III.5.4 Performance: Overhead

Figure III.10 presents the overhead for the benchmarks. Phase-guided profiling demonstrates significantly better performance than periodic profiling. The overhead is on average six times lower (21% compared to

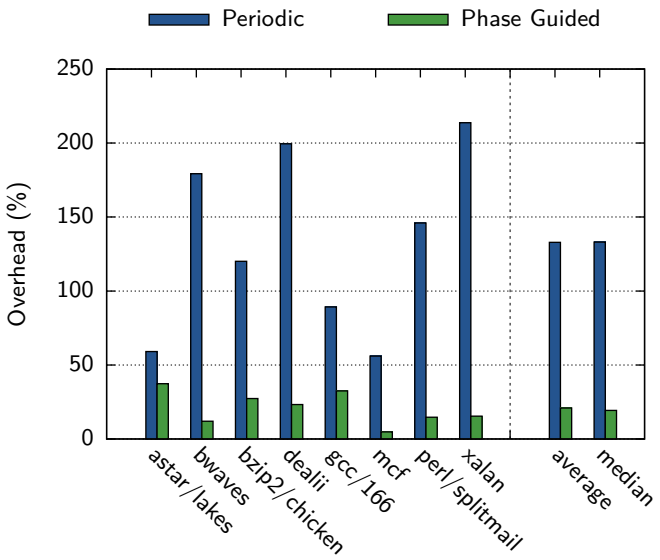


Figure III.10: The runtime overhead percent for periodic profiling (average 133%) and phase-guided profiling (average 21%).

133%) than periodic profiling[4].

This significant improvement is achieved by intelligently deciding when and where to profile. Only the most necessary parts of the execution are chosen, resulting in fewer samples required for similar accuracy. The longer the execution and the fewer phases an application has, the better the phase-guided profiling performance will be compared to periodic.

III.5.5 Summary

The accuracy and overhead results show that the StatCache cache model can be efficiently combined with phase-guided profiling to estimate the miss ratio over time for different cache sizes. The result is both more accurate and has a better performance than periodic profiling.

[4]The overhead for periodic profiling would be constant if the cost of a reuse distance was the same for all applications, and if the number of samples in each window was fixed. This is, however, not the case. First, the cost of a sample depends on the memory behavior, i.e., the number of false positives (page faults). Second, the sample rate is per memory access. Applications with more memory instructions will therefore collect more samples.

III.6 Related Work

In this section we discuss work related to cache behavior over time and phase detection.

Agaram et al. [1] looked at memory behavior over time by analyzing performance by data structure. They observed that a stable overall miss ratio can hide important changes. For example, the overall miss ratio can appear stable while the miss ratios for individual data structures changes. However, they did not map this behavior to application phases. The ScarPhase phase detection would detect such behavior as separate phases if it was caused by changes in the code path.

Both Agaram et al. and others [32, 20] 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 intra-phase variations will be, and vise versa. This is not a significant issue for this work since we profile several windows from the same phase. The profile for the phase will therefore be much closer to the true behavior than if only a single window was selected.

One important feature of this work not found in these others is that we model arbitrary cache sizes. Focusing on just one cache size can ignore important phase distinctions at other cache sizes, as seen in Figure III.2b.

ThreadSpotter [37] is a commercial tool that can detect memory bottlenecks. It leverages the work with reuse distances and statical cache models from [3, 5] to find memory bottlenecks and provide developers with information on how to improve performance. ThreadSpotter uses exponential back-off to reduce overhead by increasing the time between samples for long-running applications. This allows profiling of both short- and long-running applications. Unfortunately, the method is best suited for average miss ratios. Consider gcc in Figure III.1. Exponential back-off might detect phase B, but it would start to merge it with A, C or D in later instances.

Nagpurkar et al. [27] used phase-guided profiling for distributed profiling in embedded devices, where each phase was profiled separately on a different device. The results showed that phase-guided profiling can reduce communication, computation and energy costs. Their implementation used custom hardware [33] to collect basic block vectors in order to detect phases, and assumed perfect prediction.

In this work we use code-based phases to guide reuse distance sampling. Shen, Zhong, and Ding [31] turned this approach around, and instead used stack distances [25] to define phases. They argue that their phases are better at predicting memory behavior. While they do not

report any overhead numbers, it is clear that the cost of using hardware performance counters to sample code execution paths is much cheaper than sampling reuse distance or stack distances.

III.7 Conclusions

In this paper we have shown the importance of considering both program phases and the effects of cache allocation in understanding application behavior. Phase-aware analysis is required to identify important transient behavior in applications (see Figure III.1b), which is obscured by average metrics. The effects of different cache allocations can also have a significant impact on program behavior (see Figure III.2b), and ignoring them can lead to incorrect phase classification.

We have also shown the benefits of integrating online phase detection and statistical cache modeling to produce a phase-guided statistical cache analysis tool. By doing so we have improved both performance and accuracy over previous techniques, while also providing more valuable data in the form of time-dependent cache behavior. To further improve the accuracy we investigated different sources of intra-phase variation and described a sampling technique to overcome them.

The resulting method has better accuracy than previous statistical cache modeling methods, requires no custom hardware or application modifications, and has an overhead six times lower than previous methods.

Appendix: Selecting Sampling Parameters

One of the goals of this work has been to find methods that are automatic and do not require custom settings for every benchmark or data set. This is important since it allows the user to seamlessly work with different input data and applications without having to adjust the tools for each change. Phase-guided profiling makes most parts of the profiling automatic by adapting to the number and length of phases in the application. Configuring periodic profiling, however, is more difficult as the sampler does not adapt to the application's behavior. In general, a good sampler setting should be able to collect information from all phases of an application regardless of the input. In this appendix we show how we selected the settings used in the evaluation to achieve this.

We chose to base our setting on gcc/166 as it is short and has many phases. This makes it a particularly tricky application to profile accurately. Therefore, settings with good accuracy for gcc should produce good results for other applications, but may do so at the cost of higher

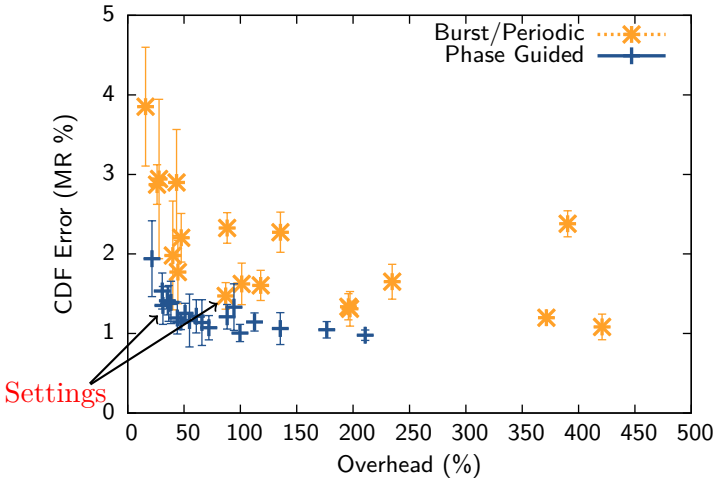


Figure III.11: CDF Error (accuracy) vs. overhead for different sampler settings. The chosen settings for equal accuracy are indicated with arrows. For all overheads the phase-guided approach has better accuracy.

overhead than necessary. This tradeoff between overhead and accuracy is a problem with fixed sampling strategies in general. To evaluate the impact of changing the sampling settings on the periodic and phase-guided sampling, we ran the samplers five times and varied the number of samples in each window and the period between the windows. The more samples and the shorter period, the higher the overhead.

Figure III.11 shows the tradeoffs in accuracy and overhead for periodic and phase-guided profiling. The error bars indicate the standard deviation in error across the different runs. As expected, the accuracy tends to improve with the overhead. However, the phase-guided profiling is both more accurate and has a lower overhead across the full range. We can also see that the variance is lower for phase-guided profiling, since it is less sensitive to different settings.

For the evaluation we chose the two settings indicated in Figure III.11 with roughly the same accuracy on gcc/166. In both cases the application is divided up into windows of 100M instructions. For periodic profiling, every eighth window is profiled with one sample for every 400K memory access in the window. For phase-guided profiling, the number of samples in each window is reduced to one per 1M memory accesses. However, the number of samples in each phase is still higher since the phase-guided method is able to combine samples from several windows belonging to the same phase before processing them.

III.8 References for Paper III

[1] Kartik K. Agaram et al. "Decomposing memory performance: data structures and phases". In: *Int. Symposium on Memory management*. 2006.

[2] Murali Annavaram et al. "The Fuzzy Correlation between Code and Performance Predictability". In: *Int. Symposium on Microarchitecture*. 2004.

[3] E. Berg and E. Hagersten. "StatCache: a probabilistic approach to efficient and accurate data locality analysis". In: *Int. Symposium on Performance Analysis of Systems and Software*. 2004.

[4] E. Berg, H. Zeffer, and E. Hagersten. "A statistical multiprocessor cache model". In: *Int. Symposium on Performance Analysis of Systems and Software*. 2006.

[5] Erik Berg and Erik Hagersten. "Fast data-locality profiling of native execution". In: *Int. Conf. on Measurement and Modeling of Computer Systems*. 2005.

[6] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Conf. on Programming Language Design and Implementation*. 2005.

[7] Dhruba Chandra et al. "Predicting inter-thread cache contention on a chip multi-processor architecture". In: *Int. Symposium on High-Performance Computer Architecture*. 2005.

[8] Ashutosh S. Dhodapkar and James E. Smith. "Comparing Program Phase Detection Techniques". In: *Int. Symposium on Microarchitecture*. 2003.

[9] Ashutosh S. Dhodapkar and James E. Smith. "Managing multi-configuration hardware via dynamic working set analysis". In: *Int. Symposium on Computer Architecture*. 2002.

[10] Richard O. Duda, Peter E. Hart, and David G. Stork. "Pattern Classification". In: 2nd ed. Wiley-Interscience, 2001. Chap. 10.11. On-line Clustering, pp. 559–565. ISBN: 0-471-05669-3.

[11] Jan Edler and Mark D. Hill. *Dinero IV Trace-Driven Uniprocessor Cache Simulator*. 1998. URL: `http://www.cs.wisc.edu/~markhill/DineroIV`.

[12] D. Eklov and E. Hagersten. "StatStack: Efficient modeling of LRU caches". In: *Int. Symposium on Performance Analysis of Systems Software*. 2010.

[13] David Eklov, David Black-Schaffer, and Erik Hagersten. “Fast modeling of shared caches in multicore systems”. In: *Int. Conf. on High Performance and Embedded Architectures and Compilers*. 2011.

[14] Alexandra Fedorova et al. “Performance of multithreaded chip multiprocessors and implications for operating system design”. In: *Proceedings on USENIX Annual Technical Conference*. 2005.

[15] John L. Henning. “SPEC CPU2006 benchmark descriptions”. In: *SIGARCH Comput. Archit. News* (2006).

[16] *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Volume 3B: System Programming Guide. 30.4.4 Precise Event Based Sampling (PEBS). Intel Corporation. 2010.

[17] *Intel VTune Amplifier XE 2011 Getting Started Tutorials for Linux* OS*. Section Key Concept: Event Skid. 2010. Chap. Key Concept: Event Skid, p. 15.

[18] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. “Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management”. In: *Int. Symposium on Microarchitecture*. 2006.

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

[20] J. Lau et al. “Motivation for Variable Length Intervals and Hierarchical Phase Behavior”. In: *Int. Symposium on Performance Analysis of Systems and Software*. 2005.

[21] J. Lau et al. “The Strong correlation Between Code Signatures and Performance”. In: *Int. Symposium on Performance Analysis of Systems and Software*. 2005.

[22] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. “Transition Phase Classification and Prediction”. In: *Int. Symposium on High-Performance Computer Architecture*. 2005.

[23] Alvin R. Lebeck and David A. Wood. “Cache Profiling and the SPEC Benchmarks: A Case Study”. In: *Computer* (1994).

[24] David Levinthal. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*. Tech. rep. Version 1.0. Intel Corporation, 2009.

[25] R. L. Mattson et al. “Evaluation techniques for storage hierarchies”. In: *IBM Systems Journal* (1970).

[26] Priya Nagpurkar, Chandra Krintz, and Timothy Sherwood. “Phase-Aware Remote Profiling”. In: *Int. Symposium on Code Generation and Optimization*. 2005.

[27] Priya Nagpurkar et al. “Online Phase Detection Algorithms”. In: *Int. Symposium on Code Generation and Optimization*. 2006.

[28] Nitzan Peleg and Bilha Mendelson. “Detecting Change in Program Behavior for Adaptive Optimization”. In: *Int. Conf. on Parallel Architecture and Compilation Techniques*. 2007.

[29] Erez Perelman et al. “Detecting phases in parallel applications on shared memory architectures”. In: *Int. Symposium on Parallel and Distributed Processing*. 2006.

[30] Andreas Sembrant, David Eklov, and Erik Hagersten. “Efficient Software-based Online Phase Classification”. In: *Int. Symposium on Workload Characterization*. 2011.

[31] Xipeng Shen, Yutao Zhong, and Chen Ding. “Locality phase prediction”. In: *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. 2004.

[32] 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.

[33] Timothy Sherwood, Suleyman Sair, and Brad Calder. “Phase tracking and prediction”. In: *Int. Symposium on Computer Architecture*. 2003.

[34] Timothy Sherwood et al. “Automatically characterizing large scale program behavior”. In: *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. 2002.

[35] Tyler Sondag and Hridesh Rajan. “Phase-based tuning for better utilization of performance-asymmetric multicore processors.” In: *Int. Symposium on Code Generation and Optimization*. 2011.

[36] David K. Tam et al. “RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations”. In: *Int. Conf. on Architectural support for Programming Languages and Operating Systems*. 2009.

[37] *ThreadSpotter*. 2010. URL: `http://www.roguewave.com/products/threadspotter.aspx`.

Paper IV

Paper IV

Modeling Performance Variation Due to Cache Sharing

Andreas Sandberg, Andreas Sembrant,
Erik Hagersten and David Black-Schaffer

In Proceeding of the
International Symposium on
High-Performance Computer Architecture
Shenzhen, China, February 2013

HPCA February 2013, Shenzhen, China

Modeling Performance Variation Due to Cache Sharing

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

Abstract

Shared cache contention can cause significant variability in the performance of co-running applications from run to run. This variability arises from different overlappings of the applications' phases, which can be the result of offsets in application start times or other delays in the system. Understanding this variability is important for generating an accurate view of the expected impact of cache contention. However, variability effects are typically ignored due to the high overhead of modeling or simulating the many executions needed to expose them.

This paper introduces a method for efficiently investigating the performance variability due to cache contention. Our method relies on input data captured from native execution of applications running in isolation and a fast, phase-aware, cache sharing performance model. This allows us to assess the performance interactions and bandwidth demands of co-running applications by quickly evaluating hundreds of overlappings.

We evaluate our method on a contemporary multicore machine and show that performance and bandwidth demands can vary significantly across runs of the same set of co-running applications. We show that our method can predict application slowdown with an average relative error of 0.41% **(maximum** 1.8%**) as well as bandwidth consumption. Our method is an average of** $213\times$ **faster than native execution of the applications for performance measurements.**

IV.1 Introduction

Shared caches in contemporary multicores have repeatedly been shown to be critical resources for performance [15, 23, 28, 8, 17]. A significant

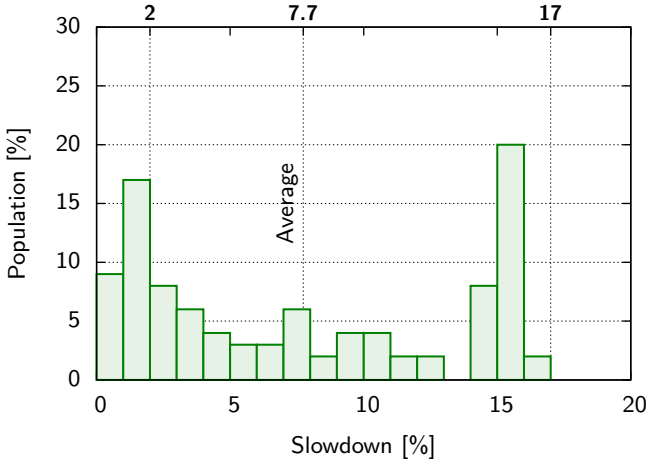


Figure IV.1: Performance distributions for astar co-running together with bwaves on an Intel Xeon E5620 based system. *Ignoring performance variablity can be misleading, since the average (7.7%), hides the fact that the performance can vary between 1% and 17% depending on how the two applications' phases overlap.*

amount of research has investigated the impact of cache sharing on application performance [23, 30, 12, 11]. However, most previous research provides a single value for the slowdown of an application pair due to cache sharing and ignores the variability that occurs across multiple runs. This variability occurs due to different overlappings of application phases that occur when they are offset in time. As the different phases have varying sensitivities to contention for the shared cache, the result is a wide range of slowdowns for the same application pair.

In multicore systems, there can be large performance variations due to cache contention, since an application's performance depends on how its memory accesses are interleaved with other applications' memory accesses. For example, when running astar/lakes and bwaves from SPEC CPU2006, we observe an average slowdown of 8% for astar compared to running it in isolation. However, the slowdown can vary between 1% and 17% depending on how the two applications' phases overlap. Figure IV.1 shows astars slowdown distribution of 100 runs with different offsets in starting times. A developer assessing the performance of these applications could draw the wrong conclusions from a single run, or even a few runs, since the probability of measuring a slowdown smaller than 2% is more than 25%, while the average slowdown is almost 8% and the maximum slowdown is 17%.

In order to accurately estimate the performance of a mixed workload, we need to run it multiple times and estimate its performance distribution. This is a both time- and resource-consuming process. The distribution in Figure IV.1 took almost seven hours to generate; our method reproduces the same performance distribution in less than 40 s.

To do this, we combine the cache sharing model proposed by Sandberg et al. [16], the phase detection framework developed by Sembrant et al. [19], with the co-execution phase optimizations proposed by Van-Biesbrouck et al. [25]. This allows us to efficiently predict the performance and bandwidth requirements of mixed workloads. In addition, the input data of the cache model is captured using low-overhead profiling [7] of each application running in isolation. This means that only a small number of profiling runs need to be done on the target machine. The modeling can then be performed quickly for a large number of mixed workloads and runs.

The main contributions of this paper are:

- An extension to a statistical cache-sharing model [16] to handle time-dependent execution phases.
- A fast and efficient method to predict the performance variations due to shared cache contention on modern hardware by combining a cache sharing model [16] with phase optimizations [19, 25].
- A comparison with previous cache-sharing methods [16] demonstrating a 2.78× improvement in accuracy from a relative error of 1.14% to 0.41% and a 3.5× smaller maximum error from 6.3% to 1.8%.
- A analysis of how different types of phase behavior impact the performance variations in mixed workloads.

IV.2 Putting it Together

Our method combines and extends three existing pieces of infrastructure: a cache sharing model [16], a low-overhead cache analysis tool [7], and a phase detection framework [19]. In this section, we describe the different pieces and how we extend them.

IV.2.1 Cache Sharing

We use the cache sharing model proposed by Sandberg et al. [16] for cache modeling. It accurately predicts amount of cache used, CPI,

and bandwidth demand for an application in a mixed workload of co-executing single-threaded applications. The input to the model is a set of independent *application profiles*. These profiles contain information about how the *miss rate* (misses per cycle) and *hit rate* (hits per cycle) vary for an application as a function of cache size. We use the Cache Pirating [7] technique (discussed below) to capture the model's input data.

The model conceptually partitions the cache into two parts with different reuse behavior. The model keeps frequently reused data safe from replacements, while less frequently reused data shares the remaining cache space proportionally to its application's miss rate. The partitioning between frequently reused data and infrequently reused data is an application property that is cache size dependent (i.e., the partitioning depends on how much cache an application receives). The model uses an iterative solver that first solves cache sharing for the infrequently reused data and then updates partitioning between frequently reused data and infrequently reused data.

The model however only works on phase-less applications where the average behavior is representative of the entire application. In practice, most applications have phases. To handle this, we extend the model by slicing applications into multiple small time windows. As long as the windows are short enough, the model's assumption of constant behavior holds within the window. We then apply the model to a set of co-executing windows instead of data averaged across the entire execution.

IV.2.2 Cache Pirating

The input to the cache sharing model is an application profile with information about cache miss rates and hit rates *as a function of cache size*. Traditionally, such profiles have been generated through simulation, but such an approach is slow and it is difficult to build accurate simulators for modern processor pipelines and memory systems. Instead, we use Cache Pirating [7] to collect the data. Cache Pirating solves both problems by measuring how an application behaves as a function of cache size on the target machine with very low overhead.

Cache Pirating uses hardware performance monitoring facilities to measure target application properties at runtime, such as cache misses, hits, and execution cycles. To measure this information for varying cache sizes, Cache Pirating co-runs a small cache intensive stress application with the target application. The amount of cache available to the target application is then varied by changing the cache footprint of the stress application. This allows Cache Pirating to measure any performance

metric exposed by the target machine as a function of available cache size.

The cache pirate method produces average measurements for an entire application run. This is illustrated in IV.2a. It shows CPI as a function of cache size for astar. The solid black line (Average) is the output produced with Cache Pirating.

Just examining the average behavior can however be misleading since most applications have time-dependent behavior. IV.2b instead shows astar's CPI as a function of both time and cache size. As seen in the figure, the application displays three different *phases* of behavior: some parts of the application execute with a very high CPI (phase A & phase B), while other parts execute with a very low CPI (phase C). This information is lost unless time is taken into account.

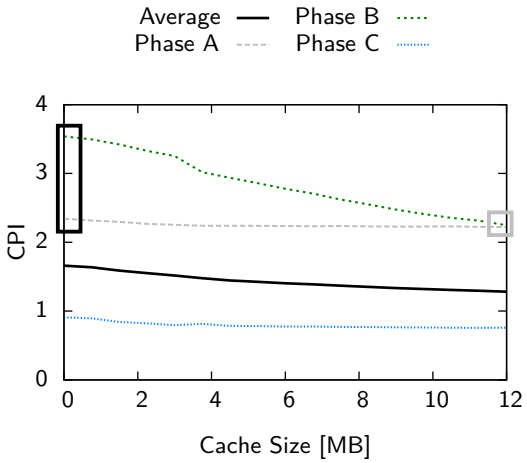
In this paper, we extend the cache pirate method to produce time-dependent data by dividing the execution into *sample windows* by sampling the performance counters at regular intervals.

IV.2.3 Phase Detection

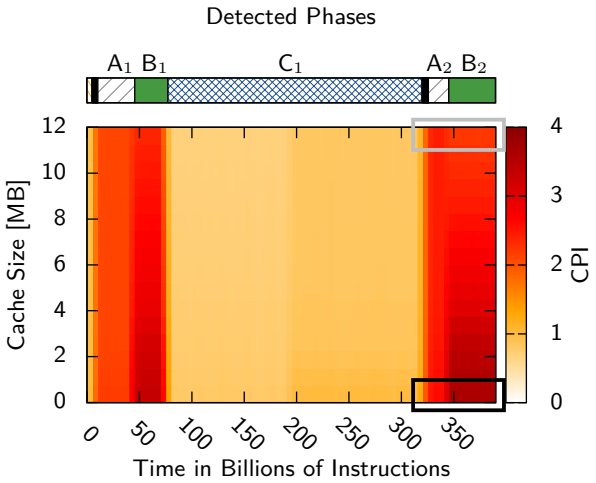
A naive approach to phase-aware cache modeling would be to model the effect of every pair of measured input sample windows. However, to make the analysis more efficient, we incorporate application phase information. This enables us to analyze multiple sample windows with similar behavior at the same time, which reduces the number of times we need to invoke the cache sharing model.

We use the ScarPhase [19] library to detect and classify phases. ScarPhase is an execution-history based, low-overhead (2%), online phase-detection library. It examines the application's execution path to detect hardware independent phases [22, 14]. Such phases can be readily missed by performance counter based phase detection, while changes in executed code reflect changes in many different metrics [21, 22, 5, 20, 9, 18]. To leverage this, ScarPhase monitors what code is executed by dividing the application into windows and using hardware performance counters to sample which branches executed in a window. The address of each branch is hashed into a vector of counters called a basic block vector (BBV) [21]. Each entry in the vector shows how many times its corresponding branches were sampled during the window. The vectors are then used to determine phases by clustering them together using an online clustering algorithm [6]. Windows with similar vectors are then grouped into the same cluster and considered to belong to the same phase.

The phases detected by ScarPhase can be seen in the top bar in Figure IV.2b for astar, with the longest phases labeled. This benchmark



(a) Time oblivious



(b) Time aware

Figure IV.2: Performance (CPI) as a function of cache size as produced by Cache Pirating. Figure (a) shows the time-oblivious application average as a solid line. Figure (b) shows the time-dependent variability of the cache sensitivity and the phases identified by ScarPhase above. The behavior of the three largest phases vary significantly from the average as can be seen by the dashed lines in Figure (a).

has three major phases; A, B and C, all with different cache behaviors. To highlight the differences in CPI, we have plotted the average CPI of each phase in IV.2a. For example, phase A runs slower than C, since it has a higher CPI. Phase B is more sensitive to cache-size changes than phase A since phase B's CPI decreases with more cache.

The same phase can occur several times during execution. For example, phase A reoccurs two times, once in the beginning and once at the end of the execution. We refer to multiple repetitions of the same phase as as *instances* of the same phase, e.g., A_1 and A_2 in IV.2b.

In addition, IV.2b also demonstrates the limitation of defining phases based on changes in hardware-specific metrics. For example, the CPI is very similar from 325 to 390 billion instructions when using 12 MB of cache (the gray rectangle), but clearly different when using less than 4 MB (the black rectangle). This difference is even more noticeable in IV.2a when comparing phase A and B. A phase detection method looking at only the CPI would draw the conclusion that phase A and B are the same phase when the application receives 12 MB, while in reality they are two very different phases. It is therefore important to find phases that are independent of the execution environment (e.g., co-scheduling).

IV.3 Time Dependent Cache Sharing

The key difficulty in modeling time-dependent cache sharing is to determine which parts of the application (i.e., sample windows or phases) will co-execute. Since applications typically execute at different speeds depending on phase, we can not simply use the ith sample windows for each application since they may not overlap. For example, consider two applications with different executions rates (e.g., CPIs of 2 and 4), executing sample windows of 100 million instructions. The slower application with a CPI of 4 will take twice as long to finish executing its sample windows as the one with a CPI of 2. Furthermore, when they share a cache together they impact each others execution rates. Instead, we advance time as follows:

1. Determine the cache sharing using the model for the current windows and the resulting CPI for each application due to its shared cache allocation.

2. Advance the fastest application (i.e., the one with lowest CPI) to its next sample window. The slower applications will not have had time to completely execute their windows. To handle this, their windows are first split into two smaller windows so that the

first window ends at the same time the fastest applications sample window. Finally, time is advanced to the beginning of the latter windows.

This means that the cache model is applied several times per sample window, since each window is usually split at least twice. For example, when modeling the slowdown of astar co-executing together with bwaves, we solve the cache sharing model roughly 13 000 times while astar only has 4 000 sample windows by itself.

We refer to the method described so far as the *window-based method* (Window) in the rest of paper. In the rest of this section, we will add two more methods, the *dynamic-window-based method* (Dynamic Window) and the *phase-based method* (Phase), which uses phase information to improve the performance by reducing number of times the cache sharing model needs to be applied[1].

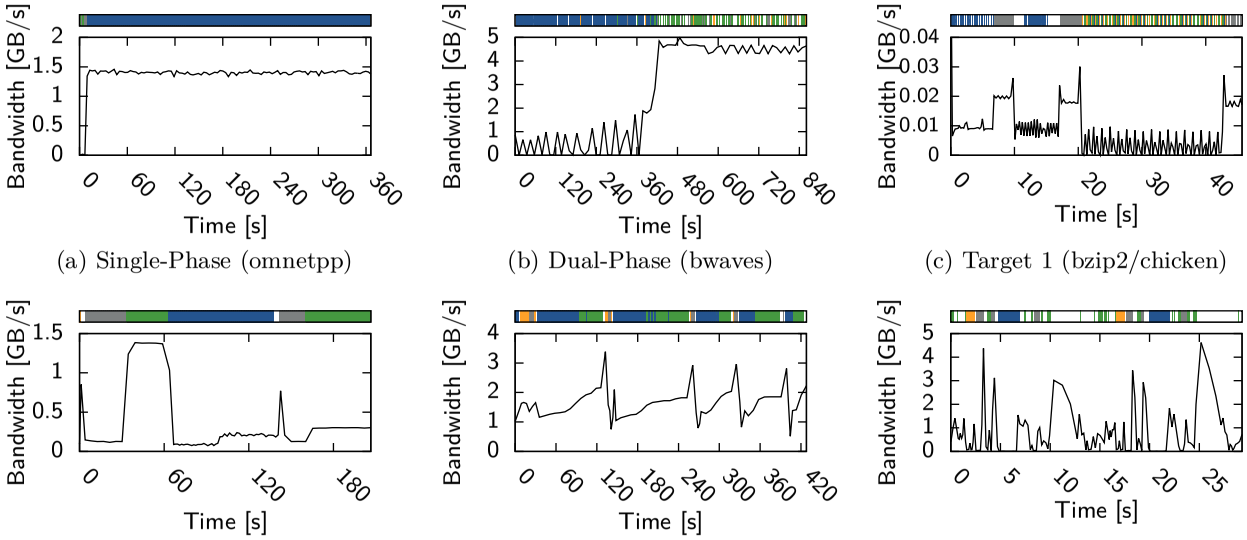
IV.3.1 Dynamic-Windows: Merging Sample-Windows

To improve performance we need to reduce the number of times the cache sharing model is invoked. To do this, we merge multiple adjacent sample windows belonging to the same phase together into larger windows, a dynamic window. For example, in astar (Figure IV.2), we consider all sample windows in A_1 as one unit (i.e., the average of the sample windows) instead of looking at every individual sample window within the phase. Merging consecutive windows within a phase assumes that the behavior is stable within a that instance (i.e., all windows have similar behavior). This is usually true and does not significantly effect the accuracy of the method. However, compared to the window-based method, it is dramatically faster. For example, modeling astar running together with bwaves we reduce the number of times the cache sharing model is used from 13 000 to 520, which leads to 25x speedup over the window-based method.

IV.3.2 Phase: Reusing Cache-Sharing Results

The performance can be further improved by merging the data for all instances of a phase. For example, when considering astar (Figure IV.2), we consider all phase instances of A (i.e., $\mathsf{A}_1 + \mathsf{A}_2$) as one unit. This makes the assumption that all instances of the same phase have similar behavior in an execution. This is not necessarily true for all applica-

[1]The cache sharing model is implemented in Python and takes approximately 88ms per invocation on our reference system (see Section IV.4.1).



(a) Single-Phase (omnetpp)

(b) Dual-Phase (bwaves)

(c) Target 1 (bzip2/chicken)

(d) Few-Phase (astar/lakes)

(e) Multi-Phase (mcf)

(f) Target 2 (gcc/166)

Figure IV.3: Bandwidth usage over the whole execution of our six benchmark applications, including the four interference applications. Detected phases are shown above. The Single-Phase, Dual-Phase, Few-Phase, and Multi-Phase behavior is clearly visible for the interference applications.

tions (e.g., same function but different input data), but works well in practice.

Looking at whole phases does not change the number of times we need to determine an applications cache sharing. It does however enables us to reuse cache sharing results for co-executing phases that reappear later [25]. For example, when astars phase A_1 co-executes with bwaves phase B, we can save the cache sharing result, and later reuse the result if the second instance (A_2) co-executes with bwaves B.

In the example with astar and bwaves, we can reuse the results from previous cache sharing solutions 380 times. We therefore only need to run the cache sharing model 140 times. The performance of the phase-based method is highly dependent on an applications phase behavior, but it normally leads to a speed-up of 2–10x over the dynamic-window method.

The main benefit with the phase-based method is when determining performance variability of a mix. In this case, the same mix is run several times with slightly different offsets in starting times. The same co-executing phases will usually reappear in different runs. For example, when modeling 100 different runs of astar and bwaves, we need to evaluate 1 400 000 co-executing windows, but with the phase-based method we only need to run the model 939 times.

In addition to reducing the number of model invocations, using phases reduces the amount of data needed to run the model. Instead of storing a profile per sample window, all sample windows in one phase can be merged. This typically leads to a 100–1000x size reduction in input data. For example, bwaves, which is a long running benchmark with a large profile, reduces its profile size from 57 MB to 82 kB.

IV.4 Evaluation

To evaluate our method we compare the overhead and the accuracy against results measured on real hardware. We ran each *target* application together with an *interference* application and measured the behavior of the target application. In order to measure the performance variability, we started the applications with an offset by first starting the interference application and then waiting for it to execute a predefined number of instructions before starting the target. We then restarted the interference application if it terminated before the target.

In order to get an accurate representation of the performance, we ran each experiment (target-interference pair) 100 times with random start offsets for the target. We used the same starting time offsets for both the hardware reference runs and for the modeled runs.

	REF	Single-Phase (omnetpp)				Dual-Phase (bwaves)				Few-Phase (astar)				Multi-Phase (mcf)			
	Time	# Model Invocations			Speedup	# Model Invocations			Speedup	# Model Invocations			Speedup	# Model Invocations			Speedup
	ISO	W	D	P	P	W	D	P	P	W	D	P	P	W	D	P	P
astar	5.9h	723K	302	84.0	2.1K×	1.4M	123K	938	88.7×	797K	1.8K	460	506×	575K	6.9K	465	435×
bwaves	24.3h	4.9M	62.0K	174	113×	7.3M	2.4M	2.0K	54.9×	5.2M	249K	1.0K	105×	4.3M	973K	1.1K	98.5×
bzip2	1.3h	242K	3.7K	63.0	103×	383K	475K	711	31.6×	272K	17.0K	373	59.7×	213K	64.0K	414	58.3×
gcc	0.8h	119K	870	140	133×	209K	203K	1.5K	18.9×	139K	4.5K	759	38.4×	101K	16.0K	786	36.6×
mcf	12.3h	1.0M	1.2K	97.0	1.9K×	2.3M	578K	1.1K	86.0×	1.2M	8.6K	518	798×	702K	30.9K	589	594×
omnetpp	10.3h	1.1M	33.0	14.0	18.6K×	2.2M	52.8K	172	110×	1.2M	358	85.0	2.5K×	856K	1.4K	99.0	1.8K×
average	9.2h	1.3M	11.4K	95.3	**695×**	2.3M	634K	1.1K	**54.9×**	1.5M	46.9K	540	**250×**	1.1M	182K	582	**215×**
												global average		1.6M	219K	572	**213×**

Table IV.1: Performance statistics for 100 runs with different starting time offsets. The number of model invocations for the three methods (W:Window, D:Dynamic-Window, and P:Phase) is shown along with the speedup for running the phase-based model vs. reference executions on the hardware. The model-based approach is on average 213× faster than hardware execution. (The highlighted results are discussed in the text.)

IV.4.1 Experimental Setup

We ran the experiments on a 2.4 GHz Intel Xeon E5620 system (Westmere) with 4 cores and 3×2 GB memory distributed across 3 DDR3 channels. Each core has a private 32 kB L1 data cache and a private 256 kB L2 cache. All four cores share a 12 MB 16-way L3 cache with a pseudo-LRU replacement policy.

The cache sharing model requires information about application fetch rate, access rate and hit rate as a function of cache size and time. We measured cache-size dependent data using cache pirating in 16 steps of 768 kB (the equivalent of one way) up to 12 MB, and used a sample window size of 100 million instructions.

IV.4.2 Benchmark Selection

In order to see how time-dependent phase behavior affects cache sharing and performance, we selected benchmarks from SPEC 2006 with interesting phase behavior. In addition to interesting phase behavior, we wanted also to select applications that make significant use of the shared L3 cache. For our evaluation, we selected four interference benchmarks that represents four different phase behaviors: Single-Phase (omnetpp), Dual-Phase (bwaves), Few-Phase (astar/lakes) and Multi-Phase (mcf).

Figure IV.3 shows the interference applications' bandwidth usage (high bandwidth indicates significant use of the shared L3 cache), and the detected phases. In addition to the interference benchmarks, we selected two more benchmarks, gcc/166 and bzip2/chicken, that we only use as targets. These benchmarks have a lower average bandwidth usage than the interference benchmarks, but they are still sensitive to cache contention. For the evaluation, we ran all combinations of the six applications as targets vs. each of the four interference applications.

IV.4.3 Performance: Speedup

Table IV.1 presents the performance of the three methods, Windows (W), Dynamic-Windows (D) and Phase (P) per interference application. The Model Invocation column shows number of times the cache sharing model was invoked. For example, when astar co-execute with bwaves, the cache model is invoked 1 400 000, 123 000 and 938 times for window, dynamic-window and the phase-based method respectively.

The reference column (REF) shows the execution time to run each target in isolation 100 times. For example, on our system, it takes 5.9 hours to run astar 100 times. The speedup column shows the speedup to model 100 co-executed runs with the phase-based method compared to

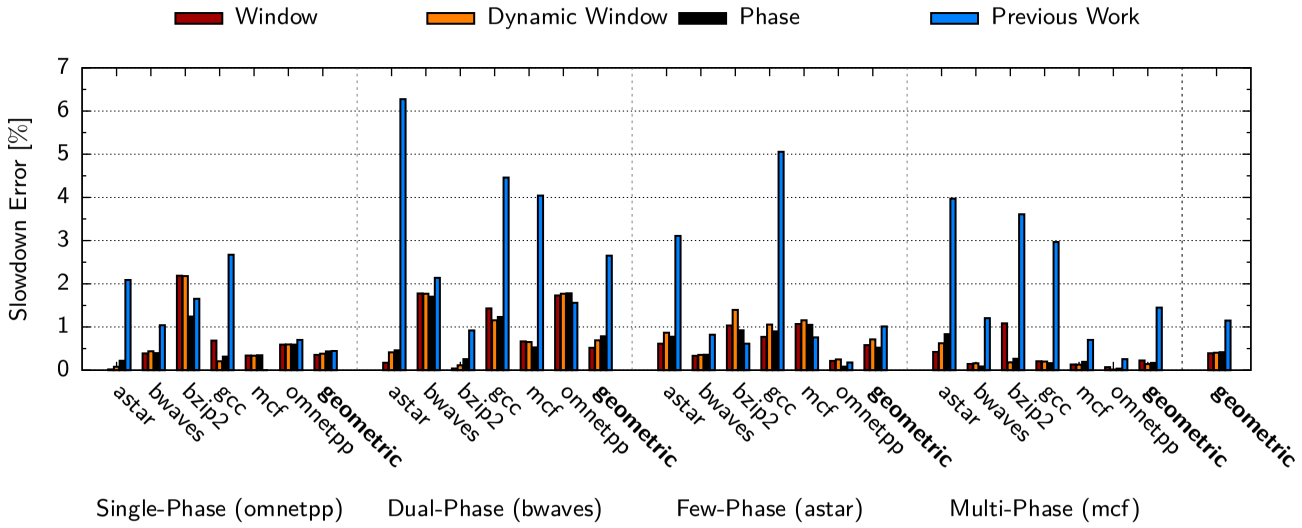


Figure IV.4: Relative error in predicted slowdown for the three methods and the previous phase-oblivious cache sharing model [16]. *This shows that the phase-based method (the fastest) can be used without lowering the accuracy. In addition, ignoring applications phase behavior will result in noticeable larger prediction errors (e.g., a 6.3% error for the phase-oblivous method when astar co-executes with bwaves).*

running the target 100 times[2]. For example, it is 88.7× faster to model 100 co-executions of astar with bwaves than to run astar 100 times in isolation.

Single-Phase. As expected, the speedup is greatest for omnetpp since it consists of just one phase. The dynamic-window method can therefore use a single large window for the whole execution. The phase-based method can then easily reuse cache sharing results whenever the target executes more instances of a phase. The geometric mean of the speedup is 695×, the highest of the four interference benchmarks.

Dual-Phase. In a similar sense, we should expect a high speedup for bwaves as well. However, bwaves executes much longer than the other interference benchmarks. So, even though the phase-based method reduces the number of times the cache sharing model is used, it has a high overhead from reading through all application profile data. On average the speedup is only 54.9×.

Few-Phase and Multi-Phase. The three methods have roughly the same performance for astar and mcf, and fall in between Single-Phase and Dual-Phase in performance. On average the speedup is 250× and 215× for astar and mcf respectively.

It is clear from the table that the phase-based methods provides the best performance for all benchmarks, with an average speedup of 213× for all interference benchmarks[3]. Next, we will evaluate the accuracy of three methods to determine if there are any tradeoffs associated with the phase-based method.

IV.4.4 Accuracy: Average Slowdown Error

Figure IV.4 presents the relative error when predicting the *average* slowdown for the three methods. On average, the windows-based method has an error of 0.39% and a maximum error of 2.2% (bzip2 + omnetpp), while the phase-based method has an average error of 0.41% and a maximum of 1.8% (omnetpp + bwaves). We can therefore safely use the much faster phase-based method without sacrificing accuracy. In the rest of this paper, we will therefore only look at the phase-based method.

In addition to the three methods, the figure also includes the error of using the previous phase-oblivious cache sharing model [16] that does not take time-varying phase behavior into consideration. The phase oblivious method has a reasonably good accuracy for omnetpp since it only

[2]The speedup exclude the time to collect the required data with cache pirating. That data is collected only once, and is then used in all the application mixes, and hence not included.

[3]Note that the speedup numbers are based on our python implementation. A C/C++ implementation would most likely result in greater speedups.

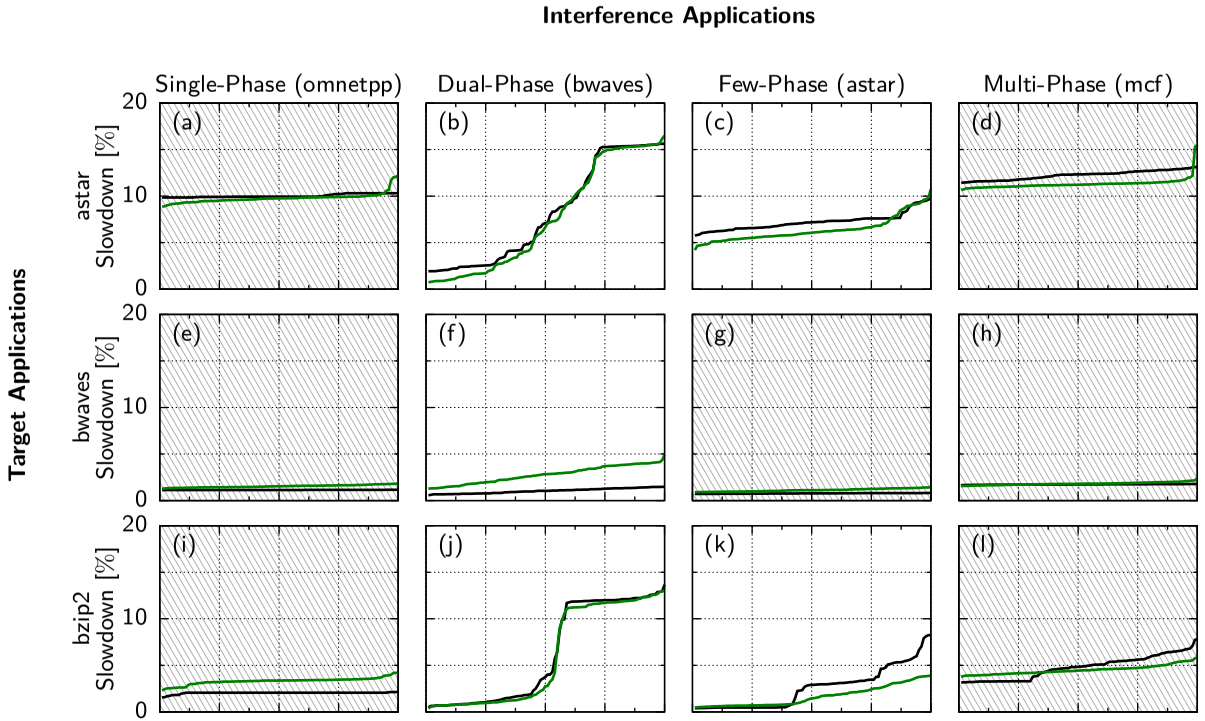


Figure IV.5: Cumulative distributions of target-slowdowns for 100 runs of each pair of applications with random start time offsets. The 100 application runs were sorted by slowdown, with the largest slowdown on the right. A flat line indicates no performance variation between the slowest and fastest run, and hence no variation. A steep curve indicates significant performance variation across the 100 runs.

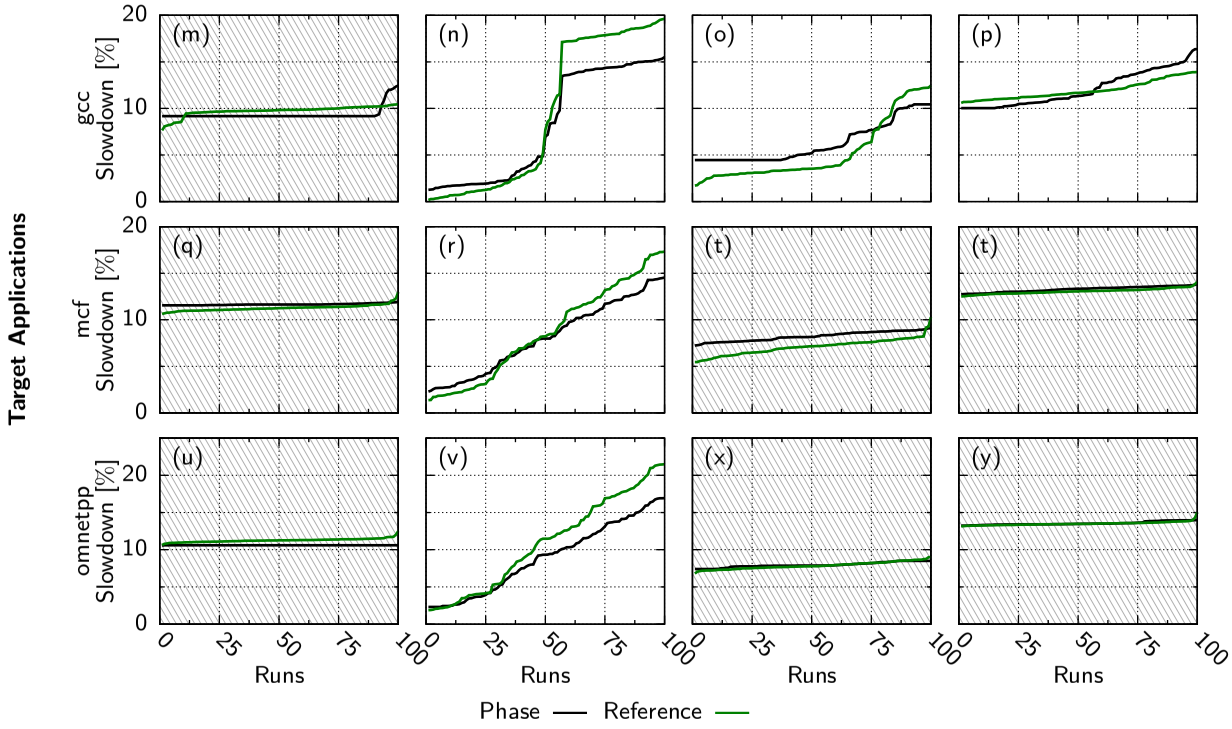


Figure IV.5: *This shows for example, that applications co-running with omnetpp have little variations (i.e., flat curves). This is to be expected since omnetpp only has one phase. Application co-executing with bwaves on the other hand have large performance variations (i.e., steep curves) since bwaves has two very different phases. The performance will thus vary depending on which of the two phases the targets co-runs with.*

has one phase. However, the error is noticeably larger for applications with more phase behavior. For example, 6.3% when astar co-executes with bwaves. This indicates that even when considering average slowdowns (e.g., ignoring variability), it is still important to consider the time-varying behavior and how the two applications' phases overlap.

On average, our phase-based method is 2.78× more accurate than previous work, with an average error of 0.41% instead of 1.14% and a 3.5× lower maximum error of 1.8% instead of 6.3% (astar + bwaves).

IV.4.5 Performance Variability

The average slowdown is a good metric for evaluating the overall accuracy of the different methods. However, it does not take performance variation into consideration. We therefore use another more descriptive metric, the *cumulative slowdown distributions* (CDF), to display the performance variations. Figure IV.5 presents the CDF for the phase-based method along with results from the reference hardware runs. The graphs with white backgrounds highlight the benchmark pairs with interesting performance variations.

The cumulative slowdown distributions can be interpreted as showing the probability for a certain maximum slowdown. For example, in Figure IV.5b, when astar is co-running together with bwaves, it has a 50% probability of having a slowdown less than 6.5%. At the same time, there is a 25% probability that the slowdown is larger than 15%.

Single-Phase. The CDF curves are mostly flat when omnetpp is used as a interference application. For example, Figure IV.5e, when bwaves is co-running with omnetpp, the curve is basically flat at 1% slowdown. This means that there are no performance variations for bwaves co-running with omnetpp, which is to be expected since omnetpp does not have any time-varying behavior.

Dual-Phase. In contrast to omnetpp, bwaves has two-phases with very different behavior. The higher bandwidth usage in the second phase indicate that it uses a larger part of the L3 cache, and will thus impose a larger slowdown on the target application. The effect on the target applications will therefore depend on the starting offset. Since the two phases have roughly the same length, we expect targets behavior to depend on how it is aligned with the phase change. For example, short targets (e.g., bzip2 in Figure IV.5j and gcc in Figure IV.5n), have a sharp turn in the CDF because their execution is not likely to overlap with the phase change. Longer targets (e.g., mcf in Figure IV.5r), have smoother distribution since they are more likely to overlap with the phase change, causing the part of the application running before the phase change to have a small slowdown, while the parts after the phase-change have a

larger slowdown. Since the position of the phase change relative to the target application will change, the CDFs will tend to become smooth.

Few-Phase. There are both flat and curved CDFs for astar as interference application. This is due to differences in the execution lengths (see Figure IV.3). The CDF in Figure IV.5g (bwaves) is flat because astar is much shorter than bwaves. Whenever the interference application terminates, it is restarted. This means that astar will be restarted over and over until bwaves terminates. The phase behavior will therefore appear homogeneous from a distance, and it results in a flat CDF. However, shorter targets (e.g., gcc in Figure IV.5o) will overlap with different phases in astar. We therefore see different target performance between runs and we find a curved CDF.

Multi-Phase. The CDFs for mcf are similar in shape to astar's for mostly the same reasons. However, mcf has a slightly different phase behavior. The same set of phases reappear several times in mcf (see Figure IV.3e). Since astar takes about half the time to execute, its execution will overlap with several of mcf's phases. Changing offsets in starting time will therfore not change astars performance, since astar will just co-execute with the same set of phases but with different instances of the same phases. We therefore see a flatter curve for astar co-running with mcf (Figure IV.5d) than with astar (Figure IV.5c).

IV.4.6 Error: Performance Variability

The CDFs produced with the phase-based method have an overall good accuracy, but do not always overlap completely with the reference curves. There are two main sources of error: cache pirating data and bandwidth limitations. We will discuss these two problems in the following sections.

Pirate Data

To measure cache-size dependent data, cache pirating co-executes a cache intensive stress application that tries to steals parts of the cache. There are two main limitations with this approach: first, if the target is also cache intensive, the pirate will have trouble keeping its working set in the cache. Second, when stealing a large portion of the cache, the pirate will have trouble touching all of its the data before it is evicted.

Figure IV.5k shows the CDF for bzip2 when co-executing with astar. The problem here is that bzip2 uses a smaller part of the L3 cache compared to the others (see Figure IV.3c), but it is also rather cache intensive for the parts it uses. Therefore Cache Pirating has difficulty stealing the require cache space, and we incorrectly estimated the cache-

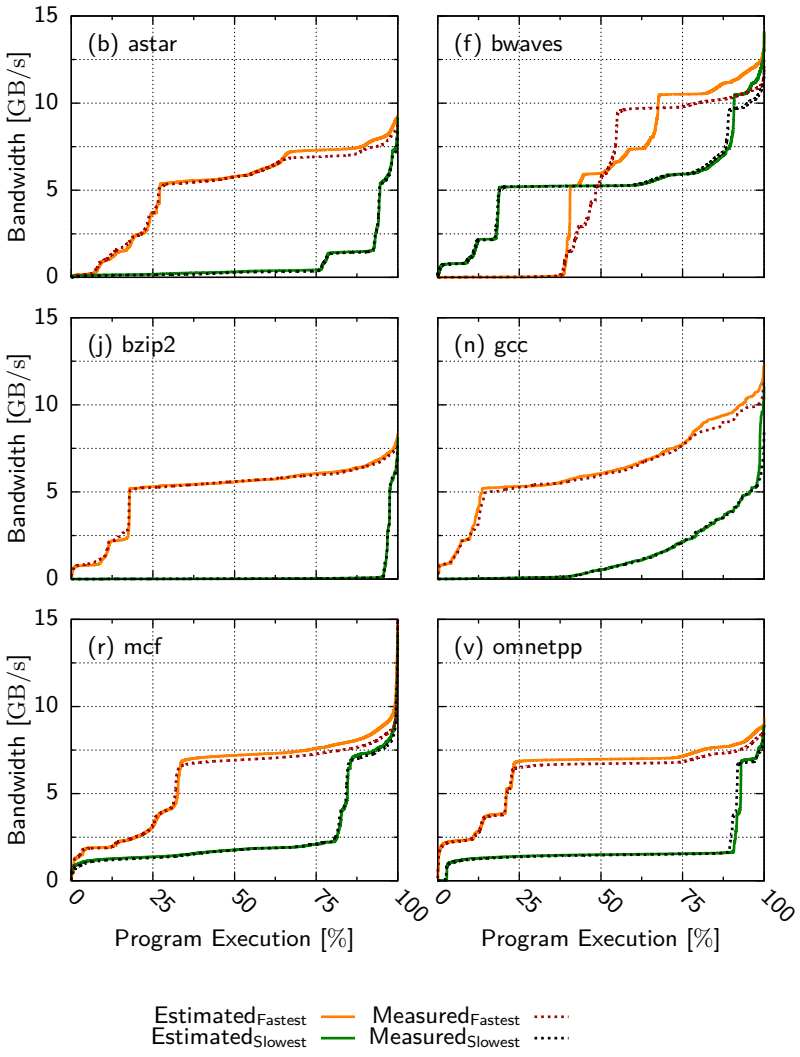


Figure IV.6: Predicted cumulative bandwidth demand (Estimated) and measured cumulative bandwidth usage (Measured) for the fastest and the slowest run when co-executing with bwaves. *Larger slowdowns when the bandwidth utilization is high.*

size dependent effects. This results in our overestimating the slowdown in the CDF.

One solution would be to instead use more cumbersome and expensive methods to acquire the data. For example, page coloring [10] could be used to limit the amount of cache the target application is allocated.

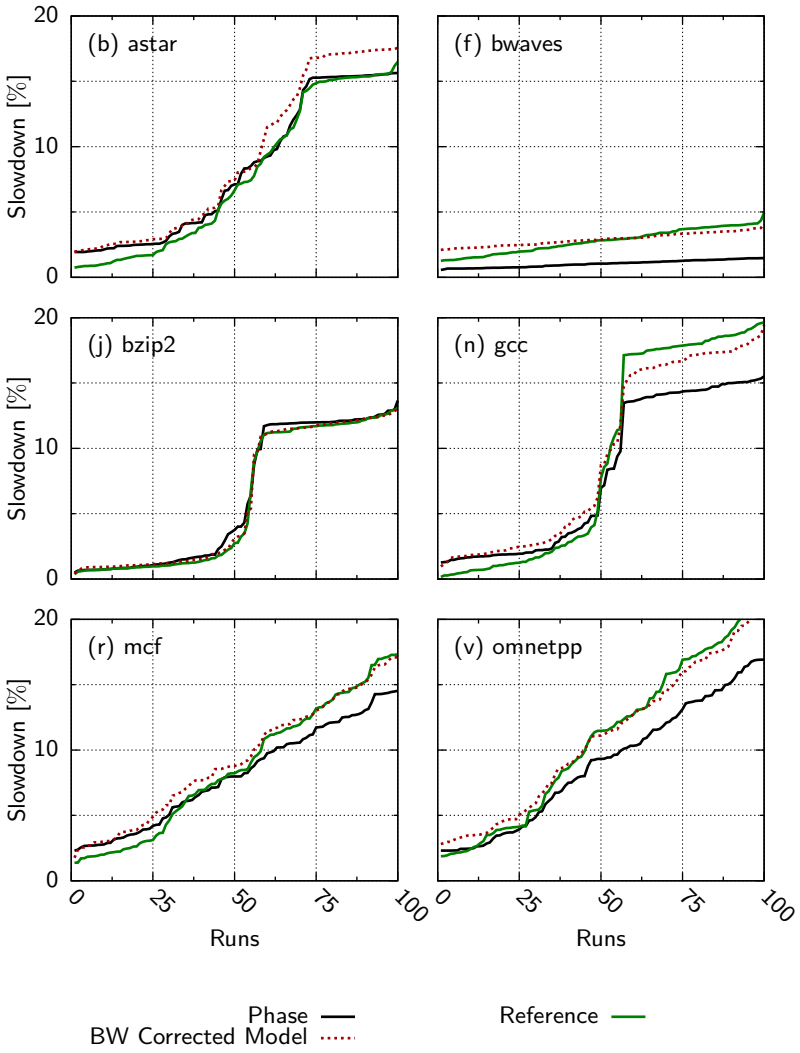


Figure IV.7: Cumulative slowdown distributions for 100 runs (as in Figure 5) with the bandwidth corrected model. *This shows that the accuracy can be improved be combining a bandwidth model with our cache sharing model to handle both cache sharing and bandwidth.*

Bandwidth

The cache sharing model assumes that the system has infinite bandwidth. This is obviously not the case, and as a result the model will underestimate the slowdown whenever the targets need more bandwidth than the system can provide. Figure IV.5 shows that we tend to un-

derestimate the slowdown for bwaves. The second phase in bwaves (see Figure IV.3b) consumes more bandwidth than the other applications[4]. If this is a problem, we should expect that we will find the largest errors when modeling bwaves, which is indeed the case.

One feature of the cache sharing model is that it can predict the required bandwidth demand the application mix would require to avoid being bandwidth limited. Figure IV.6 shows the estimated cumulative bandwidth demand[5] and the measured cumulative bandwidth the application mix received during the fastest (i.e., run 1 in Figure IV.5) and the slowest (i.e., run 100 in Figure IV.5) runs. We interpret the figures as follows: x percent of the execution has a bandwidth demand of more than y GB/s. For example, during the slowest run with mcf (Figure IV.6r), 50% of mcf's execution needs more than 7.5 GB/s to avoid slowing down due to bandwidth limitations.

The bandwidth demand is lowest for the fastest run since the target applications is co-running against the first phase in bwaves. Here, the estimated bandwidth demand and the measured bandwidth usage closely matches each other. This means that the system can provide the required bandwidth. But also, since we accurately estimate the slowdown, this also implies that the method can accurately estimate the bandwidth demand.

The slowest runs occurs when the targets are co-running with the second phase in bwaves. Here the bandwidth demand is much higher, and sometimes the estimated bandwidth demand is higher than the measured bandwidth received. This means that the target is slowing down due to bandwidth limitations. To see if we can correct the slowdown estimations by taking this into consideration, we use the measured bandwidth the application mix receives from the reference hardware runs. To do this, we update the estimated number of executed cycles (c_{est}) with the following formula:

$$c_{new\_est} = c_{est} + \frac{(BW_{est} - BW_m) * c_{est}}{BW_{MAX}}$$

where c_{new_est} is the new estimate, c_{est} is the old estimate, BW_{est} is the estimated bandwidth demand, BW_m is the measured bandwidth received and finally, BW_{MAX} is the maximum bandwidth our system

[4]Since memory accesses can come in bursts, the average bandwidth usage during a sample window can be much lower than the demand during the bursts. A low measured bandwidth usage can thus be more performance critical than expected.

[5]The model can produce bandwidth estimates by using the input profile data to estimate the application's bandwidth consumption for a given cache allocation.

can provide[6]. In other words, we extend the modeled execution time by the number of cycles it takes to transfer the amount of data that exceeds what actually was transferred.

Figure IV.7 shows the result of estimating the slowdown with the bandwidth corrected model. This correction reduces the slowdown error for bwaves, mcf, and omnetpp. However, we still underestimate the slowdown slightly for gcc, and now overestimate the slowdown for astar.

Unfortunately, such a bandwidth correction will not work in practice since it uses oracle information (i.e., BW_m), but it illustrates that a better slowdown estimate can be obtained by combining the cache sharing model with a bandwidth model to model both cache sharing and bandwidth limitations. This is a promising direction for future work.

IV.5 Case Study – Modeling Multi-Cores

In the previous section we investigated performance variations of applications pairs. However, modern processors have more than two cores. In this section, we perform a small case study to demonstrate that our method can be used to model larger application mixes, and to model system throughput.

Since all of the techniques we integrate in this method scale beyond two cores, we demonstrate that our method can scale as well by estimating the system throughput when co-running a mix of four applications on our four core reference system. To do this we compare the estimated behavior (IPC and bandwidth) to that of the actual behavior for a four application mix. Figure IV.8 shows the IPC and system throughput over time for the first ten seconds when co-running gcc, bzip2, astar and bwaves. The figure shows that the estimated IPCs matches the reference well.

The two sources of error, pirate data and bandwidth, will become more problematic when modeling larger application mixes. The amount of cache available to each application is reduced when adding more programs to the mix, which puts more pressure on the cache pirate to collect data for smaller cache allocations.

The bandwidth limitation will also become more noticeable for two reasons. First, more applications will contend for bandwidth, and thus lower the amount available to each application. Second, when an application receives less cache space, its bandwidth usage increase since it misses more in L3 and that data needs to be fetch from memory again.

[6]We estimated the real-world bandwidth limit of our reference system to approximately 12 GB/s using the STREAM benchmark [13].

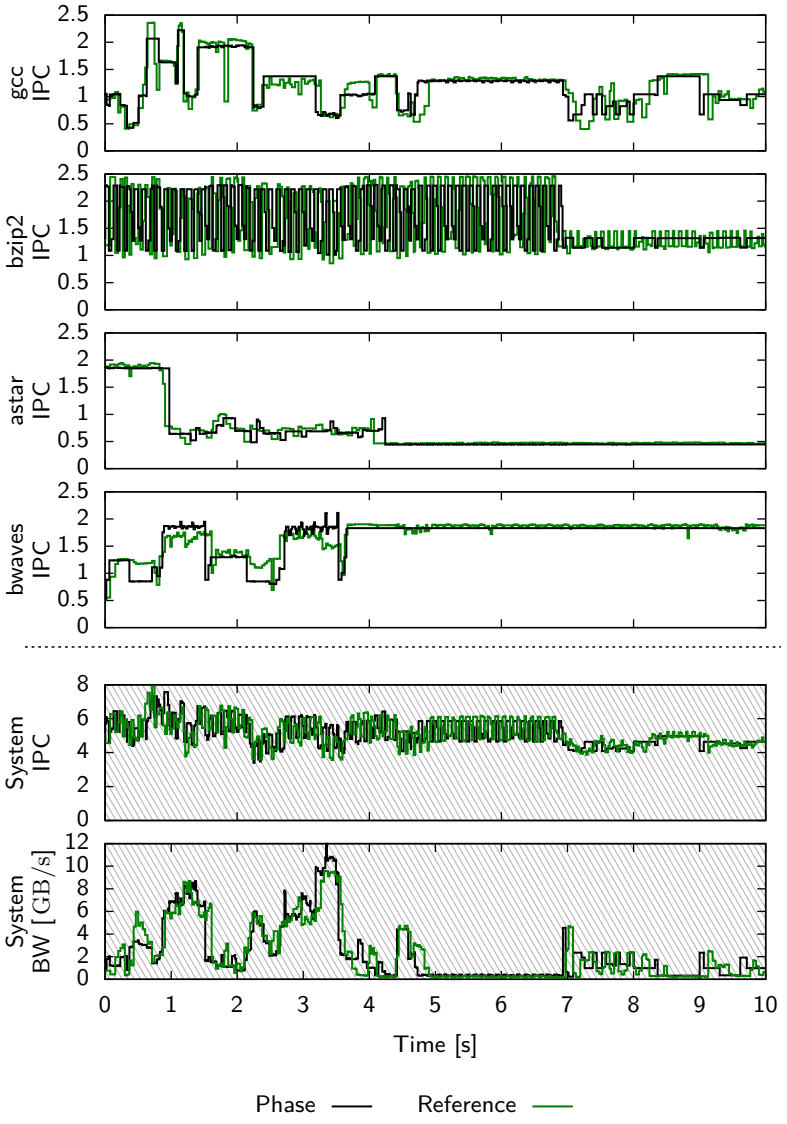


Figure IV.8: Predicted IPC (Phase) and measured IPC (Reference) for four co-running applications over time on a four-core system, as well as predicted and reference aggregate throughput (IPC) and bandwidth.

IV.6 Related Work

Techniques to explore and understand multicore performance can generally be divided into three different categories; full system simulation, partial simulation/modeling, and higher level modeling. The most expensive but also the most detail approach is full system simulation [25,

1, 24] where all cores and the entire memory system are simulated. A faster, but less detailed, approach is to only simulate/model parts of the system, and in particular the memory system. Such methods are either trace driven [2, 4, 3, 27] or use high-level data [29, 16] similar to the data we use. Finally, the least detailed approach simply aims to identify which applications are sensitive to resource contention [28, 17, 11].

Simulation normally requires combinations of applications to be simulated together, which leads to poor scaling. Van Craeynest and Eeckhout [26] combine simulation and memory system modeling to reduce the cost of simulating co-scheduled applications. Instead of simulating how applications contend for shared resources, they simulate applications running in isolation and use the output from the simulator to drive a cache sharing model. A major difference between our methods is that they depend on a single high-fidelity simulation to generate the application profiles used by their model, whereas we measure our input data with a relatively low overhead on the target system. Also, accurately simulating commodity hardware is often hard, or even impossible, since manufacturers seldom release enough information to implement a cycle-accurate simulator. Additionally, their evaluation focuses on the performance variations of the underlying hardware due to different application mixes, whereas we focus on the performance variations of the applications.

The method most similar to ours is the phase guided simulation methods by Van Biesbrouck et al. [25, 24]. Similar to our phase-based method, they use phase information to reuse simulation results. However, since their method relies on simulation they need to find and simulate representative regions (i.e., sample windows) of co-running phases. We do not have this problem since we can use the average behavior for the entire phase in our profiles.

IV.7 Conclusions

In this paper, we have presented a new analytical method that predicts performance variability due to the cache sharing effects imposed by other co-running applications. The per-application profile data the method requires can be captured cheaply and accurately during native execution on real hardware for each application in isolation. Three alternative cache-sharing methods with different performance properties were compared. We showed that the fastest method provides excellent accuracy. We have analyzed the performance variations caused by bandwidth sharing and showed that even a simple bandwidth sharing model

could explain most of the deviations observed when the bandwidth contention is high. In future work, we plan on extending our analytical method to include such bandwidth-sharing effects.

Due to its speed, simple input data, and accuracy, this method can be used to build efficient tools for software developers or system designers, and is fast enough to be leveraged in scheduling and operating system designs.

IV.8 References for Paper IV

[1] N. L Binkert et al. "The M5 Simulator: Modeling Networked Systems," in: *Proc. Annual International Symposium on Microarchitecture (MICRO)*. 2006.

[2] Dhruba Chandra et al. "Predicting Inter-thread Cache Contention on a Chip Multi-Processor Architecture". In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. 2005. ISBN: 1530-0897.

[3] X. Chen and T. Aamodt. "Modeling Cache Contention and Throughput of Multiprogrammed Manycore Processors". In: *IEEE Transactions on Computers* PP (2011).

[4] Xi E. Chen and Tor M. Aamodt. "A First-Order Fine-Grained Multithreaded Throughput Model". In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. 2009.

[5] Ashutosh S. Dhodapkar and James E. Smith. "Comparing Program Phase Detection Techniques". In: *Proc. Annual International Symposium on Microarchitecture (MICRO)*. 2003.

[6] Richard O. Duda, Peter E. Hart, and David G. Stork. "Pattern Classification". In: 2nd ed. Wiley-Interscience, 2001. Chap. 10.11. On-line Clustering, pp. 559–565. ISBN: 0-471-05669-3.

[7] David Eklöv et al. "Cache Pirating: Measuring the Curse of the Shared Cache". In: *Proc. International Conference on Parallel Processing (ICPP)*. 2011.

[8] Aamer Jaleel et al. "Adaptive Insertion Policies for Managing Shared Caches". In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008.

[9] J. Lau, S. Schoemackers, and B. Calder. "Structures for Phase Classification". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2004.

[10] Jiang Lin et al. “Gaining Insights into Multicore Cache Partitioning: Bridging the Gap Between Simulation and Real Systems”. In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. 2008.

[11] Jason Mars, Lingjia Tang, and Mary Lou Soffa. “Directly Characterizing Cross Core Interference Through Contention Synthesis”. In: *Proc. International Conference on High Performance and Embedded Architecture and Compilation (HiPEAC)*. 2011.

[12] Jason Mars et al. “Contention Aware Execution”. In: *Proc. International Symposium on Code Generation and Optimization (CGO)*. 2010.

[13] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Tech. Rep. http://www.cs.virginia.edu/stream/. University of Virginia, 1991–2007.

[14] Nitzan Peleg and Bilha Mendelson. “Detecting Change in Program Behavior for Adaptive Optimization”. In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2007.

[15] Moinuddin K. Qureshi and Yale N. Patt. “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches”. In: *Proc. Annual International Symposium on Microarchitecture (MICRO)*. 2006.

[16] A. Sandberg, D. Black-Schaffer, and E. Hagersten. “Efficient Techniques for Predicting Cache Sharing and Throughput”. In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2012.

[17] A. Sandberg, D. Eklöv, and E. Hagersten. “Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses”. In: *Proc. High Performance Computing, Networking, Storage and Analysis (SC)*. 2010.

[18] A Sembrant, D Black-Schaffer, and E Hagersten. “Phase Behavior in Serial and Parallel Applications”. In: *Proc. International Symposium on Workload Characterization (IISWC)*. 2012.

[19] A. Sembrant, D. Eklov, and E. Hagersten. “Efficient Software-based Online Phase Classification”. In: *Proc. International Symposium on Workload Characterization (IISWC)*. 2011.

[20] T. Sherwood, S. Sair, and B. Calder. “Phase Tracking and Prediction”. In: *Proc. International Symposium on Computer Architecture (ISCA)*. 2003.

[21] Timothy Sherwood, Erez Perelman, and Brad Calder. "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications". In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2001.

[22] Timothy Sherwood et al. "Automatically Characterizing Large Scale Program Behavior". In: *Proc. Internationla Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2002.

[23] David Tam et al. "Managing Shared L2 Caches on Multicore Systems in Software". In: *Proc. Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*. 2007.

[24] M. Van Biesbrouck, L. Eeckhout, and B. Calder. "Considering All Starting Points for Simultaneous Multithreading Simulation". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2006.

[25] M. Van Biesbrouck, T. Sherwood, and B. Calder. "A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2004.

[26] K. Van Craeynest and L. Eeckhout. "The Multi-Program Performance Model: Debunking Current Practice in Multi-Core Simulation". In: *Proc. International Symposium on Workload Characterization (IISWC)*. 2011.

[27] Xiaoya Xiang et al. "All-Window Profiling and Composable Models of Cache Sharing". In: *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2011.

[28] Yuejian Xie and Gabriel H Loh. "Dynamic Classification of Program Memory Behaviors in CMPs". In: *Proc. Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*. 2008.

[29] Chi Xu et al. "Cache Contention and Application Performance Prediction for Multi-Core Systems". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2010.

[30] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. "Addressing Shared Resource Contention in Multicore Processors via Scheduling". In: *Proc. Internationla Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2010.

Paper V

Paper V

Power-Sleuth: A Tool for Investigating your Program's Power Behavior

Vasileios Spiliopoulos, Andreas Sembrant and Stefanos Kaxiras

In Proceeding of the
International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems
Washington, D.C., USA, August 2012

MASCOTS August 2012, Washington, D.C., USA 1526-7539/12/$26.00

Power-Sleuth: A Tool for Investigating your Program's Power Behavior

Vasileios Spiliopoulos, Andreas Sembrant and Stefanos Kaxiras
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
{vasileios.spiliopoulos, andreas.sembrant, stefanos.kaxiras}@it.uu.se

Abstract

Modern processors support aggressive power saving techniques to reduce energy consumption. However, traditional profiling techniques have mainly focused on performance, which does not accurately reflect the power behavior of applications. For example, the longest running function is not always the most energy-hungry function. Thus software developers cannot always take full advantage of these power-saving features.

We present Power-Sleuth, a power/performance estimation tool which is able to provide a full description of an application's behavior for any frequency from a single profiling run. The tool combines three techniques: a power and a performance estimation model with a program phase detection technique to deliver accurate, per-phase, per-frequency analysis.

Our evaluation (against real power measurements) shows that we can accurately predict power and performance across different frequencies with average errors of 3.5% **and** 3.9% **respectively.**

V.1 Introduction

In the past decades, design of computer systems has focused on delivering the highest possible performance. Optimizing for speed has been the main goal in all levels of building a system, from architecture-level decisions (e.g., complex cache hierarchies, out-of-order execution) and physical-layer design (faster transistors) to program development. Regarding the latter, profiling software [11] has proven to be a powerful tool in the hands of developers to optimize their code for speed. In the last few years, however, it is power consumption that is turning into the most critical constraint in system design. Although hardware design has taken large steps towards minimizing power consumption

through advanced power saving techniques (e.g., clock gating, power gating, voltage-frequency scaling), less effort has been spent on developing power-aware software. One of the reasons for this is the lack of advanced profiling tools for providing software developers with power-related information required to improve energy-efficiency of their code. This paper introduces Power-Sleuth, a tool for investigating your program's power behavior.

Power-Sleuth is unique in that it brings together three techniques: efficient run-time phase detection and identification (Section V.2.1), performance estimation based on analytical Dynamic Voltage and Frequency Scaling (DVFS) models (Sections V.2.2, V.4), and power estimation based on novel correlation models (Section V.5). All three components are important for a complete understanding of the power behavior of a program.

Figure V.1 motivates why both power and timing information are required for characterizing energy behavior. The figure illustrates two of the phases of gcc/166 (SPEC2006 [8]), detected with a phase detection tool, and the corresponding core power, execution time and energy consumption. Traditional profiling tools have focused on analyzing an application with respect to time. When it comes to energy, however, it is not always the case that phases with the longest execution time are the most energy-hungry phases. In our example, phase X executes for longer time in maximum frequency compared to phase Y, but it consumes less power, so the total energy of the two phases is roughly the same. This means that both time and power are required to classify phases regarding energy.

Modern processors support multiple clock-frequency steps. As we show in Figure V.1, ignoring this functionality can provide misleading information about the program behavior. Phase X is the most important of the two regarding execution time under maximum frequency, however phase Y runs for longer time at minimum frequency. This means that different phases are affected in a different way by frequency scaling, thus determining where a program spends most of its execution time is frequency dependent. Moreover, since time and power do not change uniformly with frequency, it is not valid to claim that phases X and Y are similar in terms of energy; although energy consumptions are roughly the same at maximum frequency, phase Y consumes about 39% more energy when the two phases are executed at minimum frequency.

In addition, analyzing a program in phases, as opposed to, say, execution intervals, is indispensable in two ways. First, phases allow us to relate the performance and power analysis back to the source code. A program phase typically comprises a small set of function calls, thus when we talk about phase A, we can actually reason about the power/per-

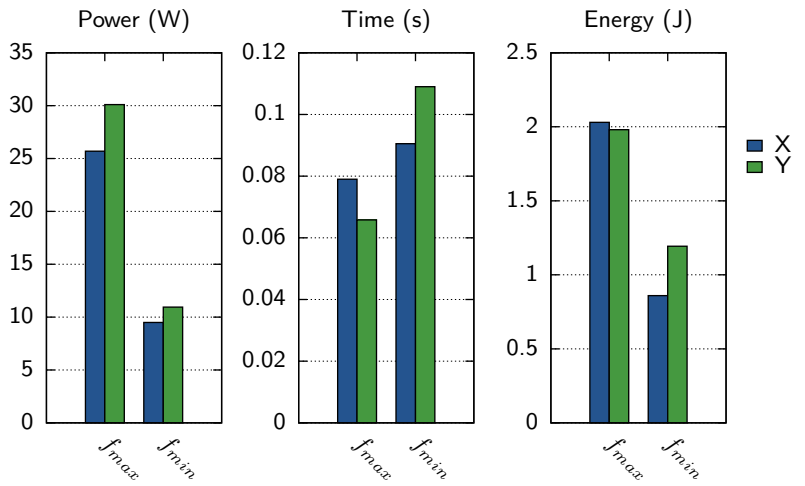


Figure V.1: The power, time and energy consumption of two phases, X and Y, at different frequencies, f_{min} and f_{max}, in gcc/166. The figure illustrates the importance of considering all frequencies. For example, phases X and Y have similar energy consumptions at f_{max}, but have distinctly different energy consumptions at f_{min}.

formance behavior of specific parts of the program. Second, if we want to understand the power/performance behavior of a program in sufficient detail so as to optimize it, by necessity we need to collect profiling information at a finer granularity than the whole program. Breaking up the execution of a program in intervals, and profiling each interval individually, allows us to do just that. However, adding phase detection on top of the intervals, brings significant leverage in how we can collect the profiling information. Since, with phase detection, each interval is assigned to a specific phase, we know that it has similar behavior to other intervals of the same phase. We can thus guide our profiling to track many more events than what the hardware allows us to sample at any time. The end result is that phase detection allows us to profile a single run of the application without restricting our ability to gather the necessary profiling information.

In brief, Power-Sleuth works as follows: it runs an application once, collects the data required, and then from this data it can provide power and performance information in any frequency of interest.

The main contributions of this paper are:

- We present a novel power correlation model that is independent of frequency.

- We combine three main components (power and performance estimation models and a phase detection and classification technique) to deliver per-phase and per-frequency power/performance analysis.

- We evaluate our approach against real, fine-grained power measurements.

- We demonstrate how accurate power/performance prediction can be utilized for understanding and improving the power efficiency of applications.

The evaluation of our approach shows that we can predict power and performance with average errors of 3.5% and 3.9% respectively.

V.2 Background

V.2.1 Detecting Program Phases

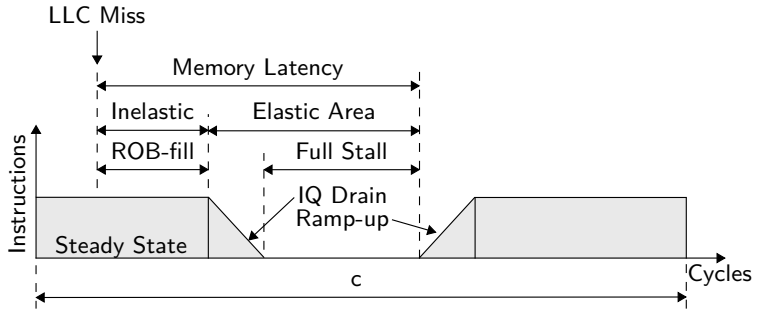
Power-Sleuth analyzes programs at the level of program-phases. We use the ScarPhase [24] library to detect and classify phases. ScarPhase is an execution history based, low overhead (2%), online phase detection library. Since it is based on the application's execution history, it detects hardware independent phases [27, 21]. Such phases can be readily missed by performance-counter based phase detection.

To detect phases, ScarPhase monitors executed code, based on the observation that changes in executed code reflect changes in many different metrics [25, 27, 2, 26, 19]. To accomplish this, execution is divided into non-overlapping intervals. During each interval, hardware performance counters are used to sample conditional branches using Intel Precise Event Based Sampling [20, 9]. 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) [25] but with only conditional branches. Each entry in the vector shows how many times its corresponding conditional branches were sampled during the interval.

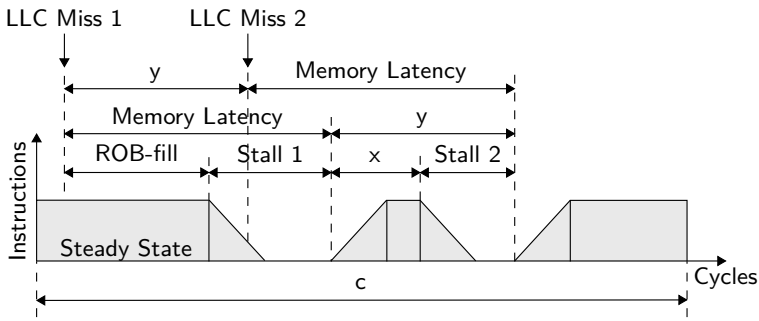
The vectors are then used to determine phases by clustering them together using an online clustering algorithm, such as leader-follower [3]. Intervals with similar vectors are then grouped into the same cluster and considered to belong to the same phase.

V.2.2 Analytical DVFS Models

In our previous work [18] we described two analytical interval-based models, named *stall-based* and *miss-based* models, to predict the impact



(a) Breakdown of LLC miss-interval into elastic-inelastic areas. The inelastic area does not scale with frequency scaling, whereas the elastic area changes with frequency scaling (but not proportionally). The total memory latency (sum of elastic and inelastic cycles) scales proportionally with frequency.



(b) Case of overlapping LLC misses handled by stall-based model. Due to the forwarding of the second miss to the head of the Reorder Buffer, an additional stall interval is introduced.

Figure V.2: Interval-based DVFS model

of frequency scaling in an application's execution time. Both models are derived from Karkhanis and Smith [16, 5] interval-based performance model. The two models extend the basic performance model by identifying: 1. the critical events that are affected by frequency scaling and 2. how these events are affected by frequency. In particular, the models suggest that execution time measured in cycles remains unaffected by frequency unless an off-chip request occurs. In this case, slowing down the processor results in a reduction of the latency of the main memory (measured in cycles). Two more groups, working independently, came up with models similar to ours [4, 23].

Stall-based model. The basic interval-based performance model breaks execution of program into intervals. During the *steady state intervals*, the processor executes instructions at a constant rate, lim-

Table V.1: Performance Counters

Phase Detection		
EVENT NAME	*EVENT CODE*	
INST_RETIRED.ANY	FIXED_CTR	
BR_INST_RETIRED	0x01C4	
Power Estimation		
EVENT NAME	*EVENT CODE*	*PARAM*
UOPS_EXECUTED.PORT_234_CORE	0x80B1	0.75
L2_RQSTS.MISS	0xAA24	-4.51
L2_RQSTS.REFERENCES.ANY	0xFF24	3.08
RESOURCE_STALLS.ANY	0x01A2	-1.38
FP_COMP_OPS_EXE.SSE_FP	0X0410	0.94
BR_MISP_EXEC.ANY	0x7F89	0.35
POWER MODEL CONSTANT	–	2.11
Performance Estimation		
EVENT NAME	*EVENT CODE*	
CPU_CLK_UNHALTED	FIXED_CTR	
UOPS_EXECUTED.CORE_STALL_CYCLES	0x3FB1	
LLC_MISSES	0x412E	

ited by the processor's width and the program's Instruction Level Parallelism. Steady state intervals are punctuated by *miss events* (e.g., cache misses, branch mispredictions), which introduce stall cycles. Figure V.2a shows how the basic interval-based performance model represents a miss-interval due to a Last Level Cache miss. When an LLC miss occurs, the processor continues to issue instructions for a few cycles until the *Reorder Buffer* (ROB) fills up. The processor then keeps executing instructions until the *Instruction Queue* (IQ) drains out of instructions independent to the pending miss. When the miss is serviced, new instructions enter the instruction window and the issue rate ramps up until it reaches the steady state. Off-chip requests are crucial events for core frequency scaling, thus it is important to understand how memory latency changes with frequency. Since $mem\_lat\_in\_core\_cycles = mem\_lat\_in\_nsec \times core\_freq$ and memory latency measured in nsec is not affected by core frequency scaling, memory latency measured in cycles scales proportionally with frequency. The stall-based model assumes that stall cycles due to off-chip requests scale proportionally with frequency. This is of course an approximation, since it disregards the ROB-fill area:

$$\begin{aligned} stall\_cycles &= mem\_lat - ROB\_fill \\ &\approx mem\_lat \end{aligned} \tag{V.1}$$

Figure V.2b shows the case of overlapping LLC misses. In addition

to the first stall interval, a second stall interval appears due to the forwarding of the second miss to the head of the ROB. In this case, the stall-based model can still be applied, since total stall cycles are approximately equal to memory latency:

$$\begin{aligned} stall_1 + stall_2 &= y + mem\_lat - ROB\_fill - x \\ &\approx mem\_lat \end{aligned} \tag{V.2}$$

In any case, stall cycles are approximately equal to memory latency, thus the total number of stall cycles (for a given part of the program) will scale proportionally to frequency. On the other hand, non-stall cycles remain intact when frequency is scaled. If c is the total execution cycles for a given amount of instructions and st is the total stall cycles under frequency f_0, then for frequency f_1 (with a scaling factor $k = f_0/f_1$) stall cycles, execution cycles and execution time can be approximated as

$$\begin{aligned} st_{new} &= \frac{st}{k} \\ c_{new} &= c - st + \frac{st}{k} \\ t_{new} &= \frac{c_{new}}{f_1} = \frac{c_{new} \times k}{f_0} = \frac{(c - st) \times k + st}{f_0} \end{aligned} \tag{V.3}$$

Miss-based model. The miss-based model on the other hand improves prediction accuracy by taking into account the existence of $ROB-fill$ area. As shown in Figure V.2a, the whole miss interval equals memory latency, and thus it is the whole miss interval that scales *proportionally* with frequency. In the case of overlapping misses (Figure V.2b), if $LLC\ Miss$ 2 occurs y cycles after $LLC\ Miss$ 1, it will also be serviced y cycles after $LLC\ Miss$ 1, regardless the frequency, since y belongs to the inelastic area of the first miss interval. Thus, the stalls generated by $LLC\ Miss$ 2 do not scale with frequency, meaning that *only the miss interval of the first miss in a cluster of overlapping misses scales with frequency*. By counting the number of these misses, our model predicts execution time over different frequencies with great accuracy (1% error on average). More details about the miss-based model can be found in [18].

V.3 Power-Sleuth Overview

Power-Sleuth is a power/performance estimation tool. To use Power-Sleuth, the user needs to run an application only once. During this run, the tool collects all the data required, and then it estimates the energy consumption of each program phase. Moreover, without any additional

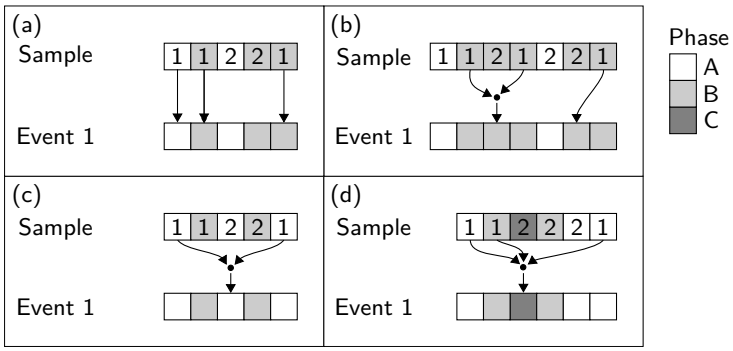


Figure V.3: Multiplexing/Interpolation methodology for event 1. (a) shows the case that an event is sampled in an interval. (b) shows how an event is approximated when it is not sampled in the same interval, but in intervals of the same instance of the phase. In (c), the event is not monitored in the current instance, so it is approximated using information from past and future instances. Finally, in (d) the event is never monitored for this phase and is approximated using information of the whole program execution. Note that for cases b, c and d, for the intervals that event 1 is actually sampled, the same approach as in case (a) is used.

runs of the application, Power-Sleuth is able to predict how power and performance of each phase will be affected if it is executed under any other frequency.

Power-Sleuth combines three basic components: an analytical DVFS performance model, an effective capacitance correlation model and a phase detection technique. Execution of the program is divided into intervals of 100M committed instructions each. Using ScarPhase, each interval is characterized by a phase ID. Performance and power models rely on information gathered by hardware performance counters: with our current setup, 11 different performance counter events have to be monitored per interval. Execution cycles and instructions retired can be monitored using two of the fixed counters, but the remaining 9 events have to be monitored using programmable performance counters. However, only 4 counters are available in our processor. One way of gathering all the required information is running the application multiple times and monitoring different events in every run. This would imply performing 3 complete runs of the same application before applying the power/performance models to give a picture of the application's behavior. Power-Sleuth manages to collect all the required information from a **single** run of the application by using a phase-guided multiplexing/in-

terpolation technique.

In the remainder of the paper we use the following terminology. We refer to a window of 100M retired instructions as an *interval*. Each interval belongs to a *phase* and is assigned a *phase ID* by the phase detection component of Power-Sleuth. Finally, we refer to consecutive intervals of the same phase as an *instance* of this phase.

V.3.1 Phase-Guided Counter Multiplexing

Since we need to concurrently monitor more events than the hardware supports, we have to multiplex multiple events in the available set of counters. Employing a simple time-based multiplexing methodology requires the program to have rather uniform behavior, otherwise approximating events that are not monitored with previous values of these events results in high errors. Instead of a simple time-based approach, we use a *phase-guided* time-based multiplexing methodology, shown in Figure V.3. In this example we assume that there is only one available counter and two different events to measure, named *event 1* and *event 2*. The same approach can be extended for different number of events to monitor. The performance counter is programmed to measure events in a per-phase, round-robin fashion. This means that Power-Sleuth remembers the type of event monitored in the last interval of each phase and programs the performance counter to measure the next event when an interval with the same phase ID is about to execute again. Since the counter has to be programmed at runtime, Power-Sleuth needs to predict the phase ID of the next interval. We use the same prediction method as in [24]. Figures V.3a-d show how events 1 and 2 are sampled for different patterns of phases A, B and C.

V.3.2 Phase-Guided Interpolation

As a consequence of multiplexing counters, not all of the events are sampled in every interval. To solve this issue we develop a phase-aware interpolation method. There is a total of 4 different cases for an event during an interval. The event is:

1. monitored in this interval,
2. not monitored in this interval but monitored in other intervals of the current instance of the phase,
3. not monitored in the current instance of the phase, but monitored in other (future or past) instances of the same phase,
4. never monitored for this phase.

Figures V.3a-d show the priority for approximating event 1 in various intervals, with (a) being the highest priority and (d) the lowest. Of course, the same idea applies for approximating event 2. Figure V.3a shows case 1, when event 1 was actually sampled in some interval. Then the value sampled is used for this interval. In Figure V.3b consecutive intervals of phase B are executed, and thus sampling is interleaved between events 1 and 2. For the intervals that event 2 was sampled, event 1 is approximated as the average of the values of event 1 sampled in the current instance of this phase. Figure V.3c depicts an example of case 3: there is an instance of phase A that event 1 is never sampled (third interval of this example). In this case, the average of the values of event 1 sampled in all past and future intervals of the same phase are used as an approximation. Finally, as shown in Figure V.3d, there is an extreme case that event 1 is never sampled for some phase (phase C in the example). In this case, the average of values of event 1 sampled in the whole execution of the program, regardless the phase, is used to approximate event 1.

The approach described above lets us have one value for each event of interest (either sampled or approximated) in every interval. This information is then fed to our models to predict power and execution time of each interval not only for the frequency the application was profiled in, but for any frequency the user wants, even if this frequency is not included in the processor's possible V-f configurations. Intervals of the same instance of a phase are grouped together to get a more smooth view of the average behavior per instance of the phases, and finally different instances of the phases are accumulated to get a per-phase performance/energy breakdown for the whole program. Note however that since we use a multiplexing and interpolating method, Power-Sleuth can still provide per interval information.

V.3.3 Experimental Setup

We run our experiments in an Intel Core i7 920 machine (Nehalem microarchitecture). The processor is a quad-core machine with 9 frequency steps (2.66GHz to 1.6GHz), 4 programmable performance counters and 3 more counters monitoring fixed events. We run benchmarks from SPEC2006 suite in a 2.6.38-12 Linux kernel. Power gating (turning off idle cores to save leakage power) is deactivated so that we can measure idle power in all frequencies. We collect real power samples using current sensors and a 16-channel A/D device. These samples are used as reference for our power estimation.

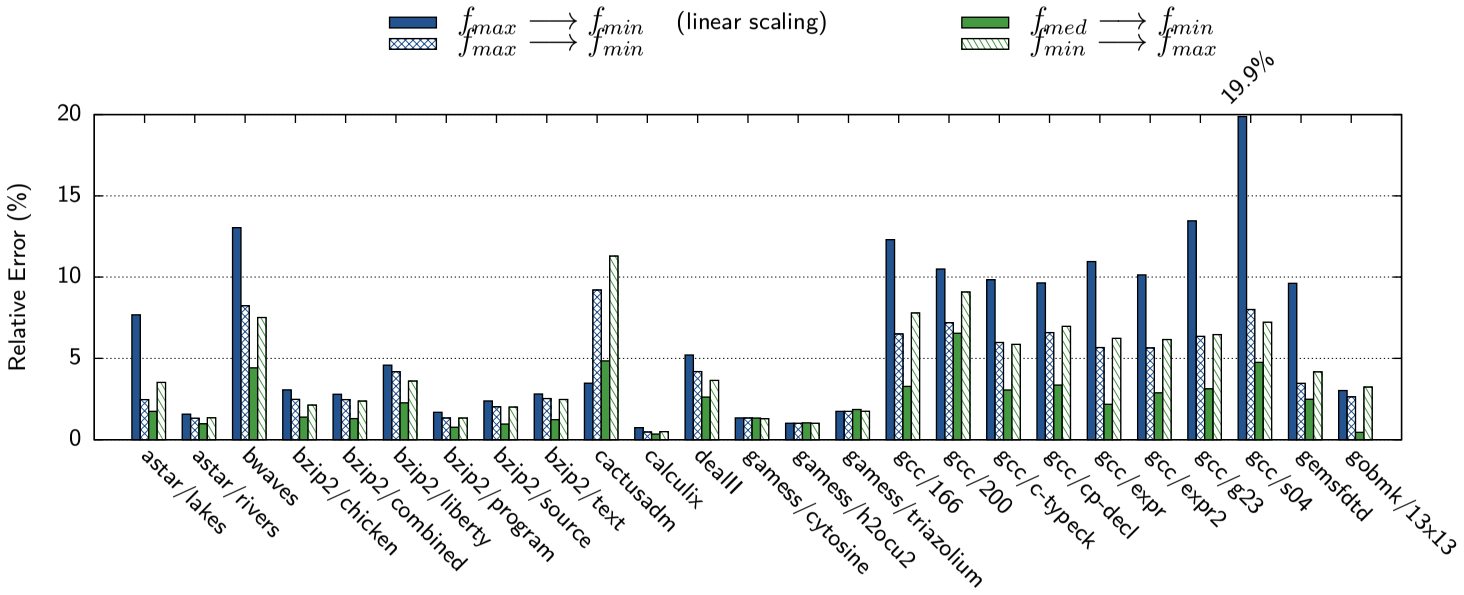


Figure V.4: Relative error in predicting execution time. The figure shows (from left to right) prediction: a) using data collected in maximum frequency and predicting for minimum frequency assuming execution time scales linearly with frequency, b) from maximum to minimum frequency using the stall-based model, c) from median to minimum frequency using the stall-based model and d) from minimum to maximum frequency using the stall-based model.

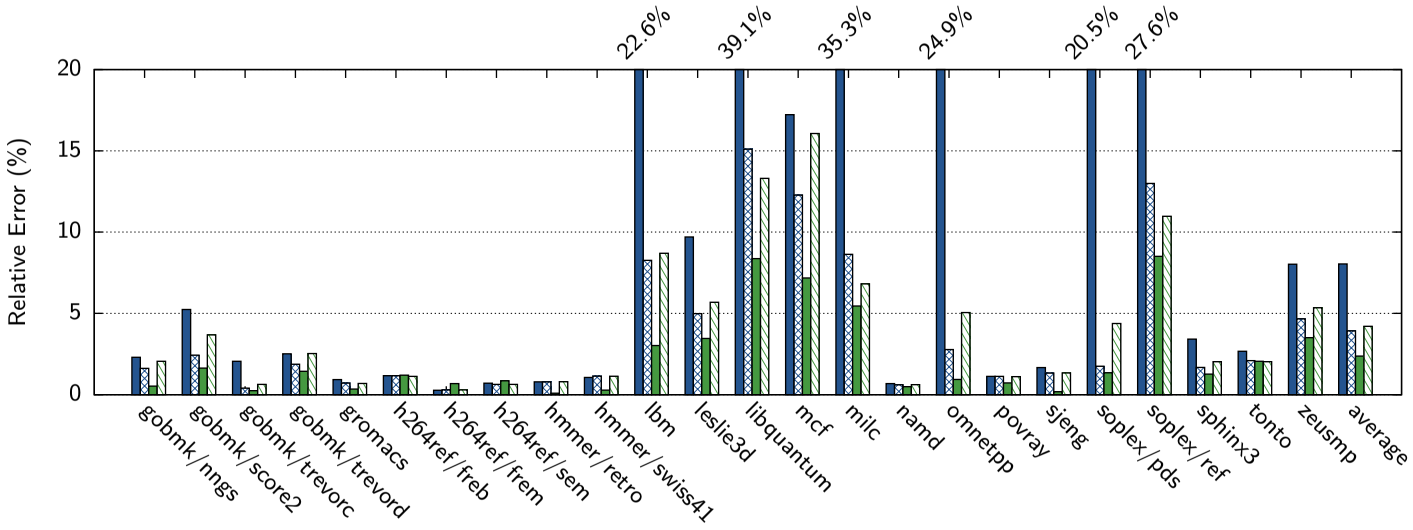


Figure V.4: Relative error in predicting execution time. The figure shows (from left to right) prediction: a) using data collected in maximum frequency and predicting for minimum frequency assuming execution time scales linearly with frequency, b) from maximum to minimum frequency using the stall-based model, c) from median to minimum frequency using the stall-based model and d) from minimum to maximum frequency using the stall-based model.

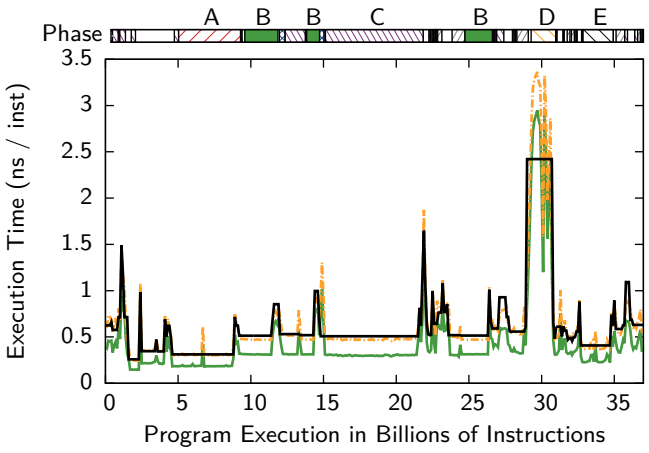
V.4 Performance Estimation

In [28] we employed the models described in Section V.2.2 to develop frequency scaling governors. These governors adapt to program behavior and pick the frequency that minimizes various energy efficiency metrics, such as *Energy Delay Product* (EDP) with/without performance constraints. In this work we use the same models, but from a different perspective: the goal now is to provide a per-phase estimation of how execution time is affected by frequency scaling. Moreover, performance prediction is crucial for our novel, cross-frequency power estimation method described in Section V.5.

After running an application once and collecting appropriate statistics, performance of each interval/phase can be estimated using the models described in Section V.2.2. The miss-based model, though more accurate, cannot be applied in our processor since there is no performance-counter event monitoring the number of clusters of misses. Neither the stall-based model can be applied as it is; there is no counter for measuring stalls due to off-chip requests. These stalls, however, can be approximated by the minimum between all the pipeline execution stalls and the worst case stalls due to off-chip misses. The latter ones are simply the number of last level cache misses multiplied by the memory latency. More details about applying the stall-based model in a Nehalem processor can be found in [28].

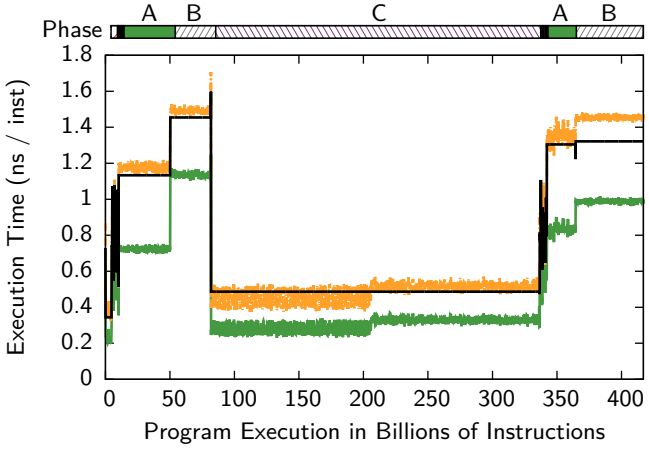
Figure V.4 illustrates the error of our prediction. The error shown per benchmark is not just the error in predicting the whole execution time. Consecutive intervals of the same phase are grouped together and the execution time of this instance of the phase is predicted. The prediction is compared to the actual execution time of the instance and the relative error is calculated. The figure shows the average of all of these errors along the execution of the program, weighted with the number of intervals of each phase. This way possible over/underestimations of different phases do not cancel each other and the error presented corresponds to the accuracy of predicting execution times of separate phases across different frequencies.

The leftmost bar shows the accuracy of a simple linear scaling model which assumes that execution time scales proportionally with frequency. Though this estimation is accurate for CPU bound programs (such as calculix and h264ref), the error is huge for the memory bound applications, reaching up to 39.1% for libquantum. This proves the necessity of using a model that takes into account the asynchronous nature of off-chip accesses. The next bar shows the prediction error when a profile run is performed in maximum frequency (2.66GHz) and execution time is predicted for minimum frequency (1.6GHz). As shown in the



f_{max} f_{min} $f_{max} \longrightarrow f_{min}$

Figure V.5: nsec per instruction for the first half of gcc/166. The second half is identical and thus omitted. The top of the figure shows the phases detected. f_{max} and f_{min} show the execution times for maximum and minimum frequency and $f_{max} \longrightarrow f_{min}$ is the predicted execution time for minimum frequency when profiling is performed in maximum frequency.



f_{max} f_{min} $f_{max} \longrightarrow f_{min}$

Figure V.6: nsec per instruction for astar/lakes. The top of the figure shows the phases detected. f_{max} and f_{min} show the execution times for maximum and minimum frequency and $f_{max} \longrightarrow f_{min}$ is the predicted execution time for minimum frequency when profiling is performed in maximum frequency.

figure, most of the benchmarks suffer low prediction errors, within 5%, whereas the worst case prediction is 15.1% (libquantum). Bare in mind that libquantum is the most memory bound application, so disregarding the ROB-fill area results in higher prediction error. The average error over all benchmarks is about 3.9%. The next bar shows how prediction accuracy can be improved by performing the profiling run in the median frequency between max and min (2.13GHz). Profiling time is now longer, but the closer the profiling frequency is to the target frequency, the better the prediction accuracy, thus worst case and average errors are reduced down to 8.4% and 2.4% respectively. Finally, the rightmost bar shows that prediction accuracy is not affected when Power-Sleuth collects profiling data in low frequency and predicts performance for a high frequency, since average error is 4.2%.

Power-Sleuth predicts execution time individually for each interval/phase. The metric we use is nsec per instruction: we divide execution time of each interval with the instructions executed in that interval (100M). Figures V.5, V.6 show the execution of the phases of gcc/166 and astar/lakes respectively. We run Power-Sleuth in maximum frequency and we measure execution time (f_{max}). At the same time, we predict execution time for minimum frequency ($f_{max} \longrightarrow f_{min}$). Finally, we run Power-Sleuth once more to actually measure execution time in minimum frequency and evaluate the accuracy of our prediction. At the top of each figure, we show the phases detected by Power-Sleuth. The figures show that Power-Sleuth accurately predicts the increase in execution time due to frequency scaling. The relative increase of execution time between maximum and minimum frequency is a metric of how memory bound an application is. The closer the ratio to 1, the more memory bound the phase is. For the predicted execution time, we average intervals of the same instance of a phase together, to get a more smooth view of the phase behavior. Thus, we can see the average performance of phase D in gcc, filtering out the peaks that appear in some intervals. However, since we perform per interval prediction, the user can skip this step.

V.5 Power Estimation

We model total power consumption as the sum of dynamic power consumption (dissipated by the switching activity of transistors) and static power (dissipated by leakage currents)[17]:

$$P = P_{static} + afCV^2 = P_{static} + fC_{eff}V^2 \tag{V.4}$$

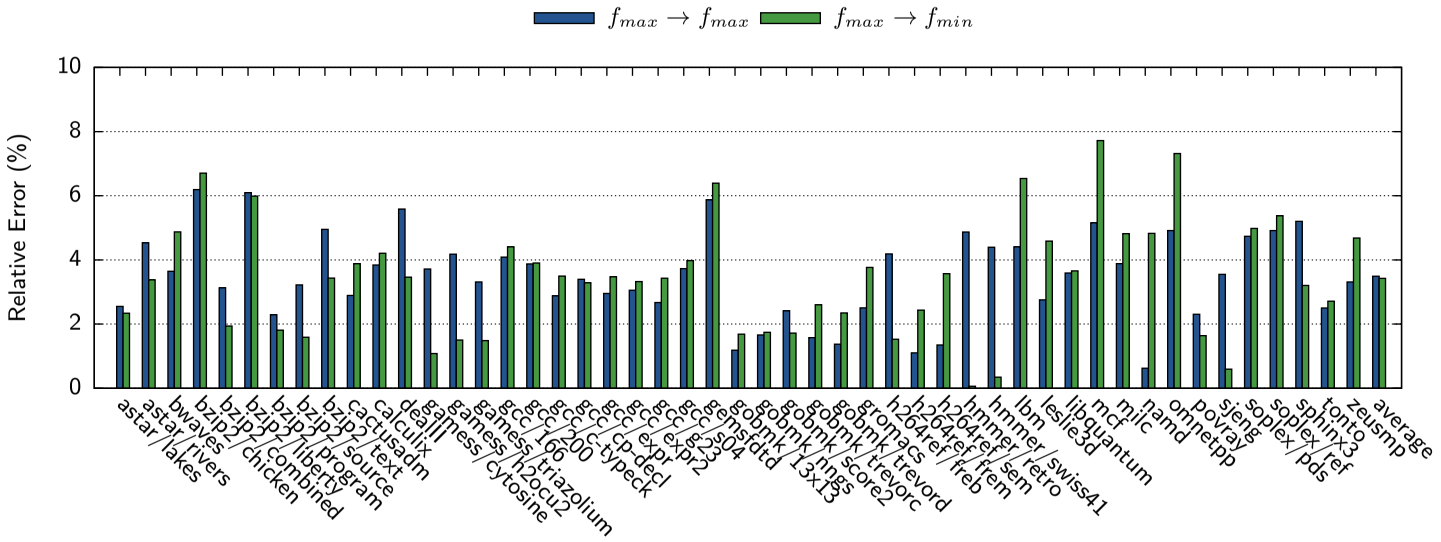


Figure V.7: Relative error in predicting power consumption. We run the application in maximum frequency and predict power for maximum (left bar) and minimum (right bar) frequency.

where P_{static} is the static power, f is the operating frequency, V is the supply voltage, C is the processor capacitance, a is the activity factor and $C_{eff} = a \times C$ is the *effective capacitance*.

Several researchers have correlated power with performance counter events [1, 7, 14, 15]. Their work is limited by voltage and frequency scaling in the sense that different models have to be formed to predict power in different frequencies. Moreover, these models, unless coupled with a performance prediction model, are not able to predict power when target frequency is not the same as the profiling frequency. Modern processors' power measuring capabilities [10] suffer from similar limitations: power monitoring can only provide power for current V-f configuration and for the whole package. Thus, it is impossible to estimate power per core in multiprocess workloads, as well as for different frequencies. To address these limitations, we develop a model that correlates core's effective capacitance with performance counters. As explained later in this section, the model is frequency independent and thus fulfills the requirements of Power-Sleuth.

V.5.1 Methodology

What is unique in our approach is that rather than correlating power with events, we develop a model that investigates directly the source of power consumption: charging and discharging of processor node-capacitances. Component activity makes capacitors switch state, and this switching activity results in power consumption which depends on the processor frequency and voltage supply. Component activity (i.e. effective capacitance) does not depend on voltage and frequency; it is only related to architecture level details and application behavior.

We develop a correlation model for estimating effective capacitance by training the model in maximum frequency, and then using the same parameters we can estimate power in any Voltage-frequency setting with Equation (V.4). Thus, the problem of estimating power is reduced to estimating effective capacitance. More importantly, we do not need to collect the data fed to the model from a profiling run in the same frequency as the target frequency: we monitor performance counter events in maximum frequency (minimizing profiling time) and then we estimate event rates by using the analytical DVFS performance model presented in Section V.2.2. This is possible because most events of interest (like micro-ops executed, cache misses etc.) are not affected by frequency scaling and the event rates (on a per cycle basis) are simply the event counts divided by the prediction of execution cycles under the target frequency.

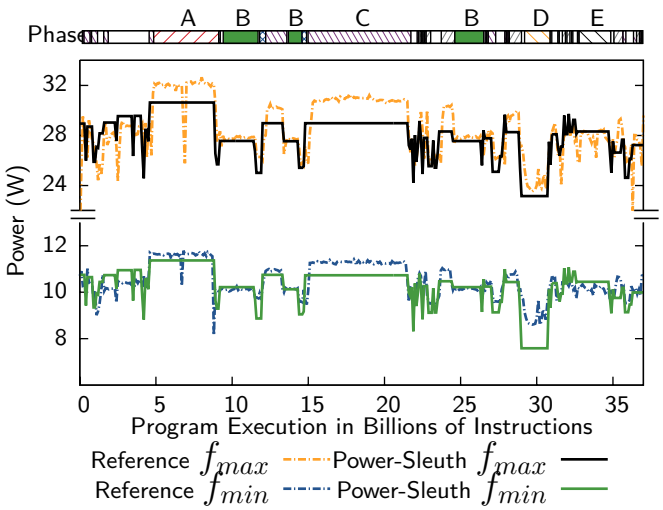


Figure V.8: Power consumption (measured and predicted) for the first half of gcc/166 at maximum (upper part) and minimum (bottom part) frequency. The second half is identical and thus omitted. Both predictions are based on the data collected from a profiling run in maximum frequency.

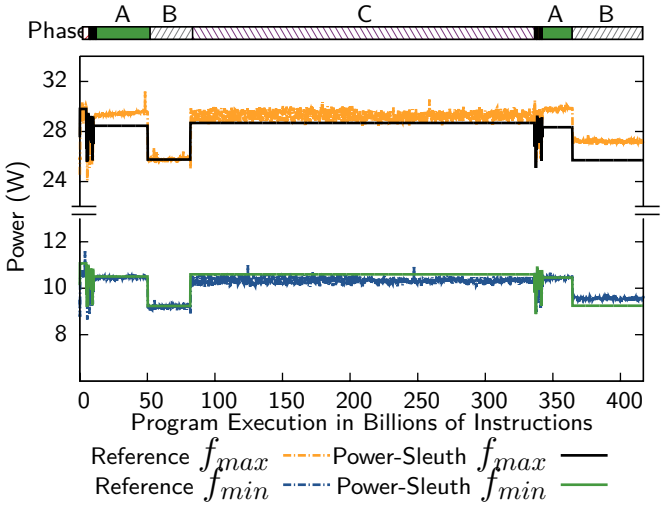


Figure V.9: Power consumption (measured and predicted) for astar/lakes at maximum (upper part) and minimum (bottom part) frequency. Both predictions are based on the data collected from a profiling run in maximum frequency.

We assume that effective capacitance is a linear function of various event rates that best describe the power behavior of the processor. We tried different events in order to track the activity of different processor components: execution units, cache hierarchy, branch predictor, and utilization of various other resources, such as load buffers and reservation station. We ended up with the events shown in Table V.1, which is a good compromise between good accuracy and low number of events monitored. One of the events with high correlation is the RESOURCE_STALLS.ANY event, which is a cycle count event and thus can be affected by frequency scaling: if resource stalls overlap with memory stalls they scale with frequency, otherwise they do not. Since it is impossible to monitor a union of the two events (cycles that are both resource and execution stalls), we use the heuristic that if resource stalls are more than execution stalls, a part of them overlaps with execution stalls and thus scales with frequency in the same way as execution stalls. This approximation, though based on a -educated- guess, provides good results: it improves the error compared to the case that no special care is taken for the scaling of resource stalls. To predict effective capacitance, we use the following equation:

$$C_{pred} = \sum_{k=0}^{5} \frac{param_k \times event_k}{cycles} + param_6 \tag{V.5}$$

where $event_k, k = 0, ..., 5$ are shown in Table V.1. To train our model for the specific hardware, we use real power measurements. We run a set of benchmarks in maximum frequency, measure processor total power consumption, subtract static power (measured for all frequencies when the processor is idle) and finally divide with $f \times V^2$. This way we compute the average effective capacitance C_i for benchmark i. We train our model by minimizing $\sum_{i \in specs} \left(C_i - C_{pred_i}\right)^2$, or

$$\sum_{i \in specs} \left(C_i - \sum_{k=0}^{5} \frac{param_k \times event_{k,i}}{cycles_i} - param_6\right)^2 \tag{V.6}$$

After obtaining the parameter values $param_k, k = 0, 1, ..., 6$ that minimize expression (V.6) (shown in Table V.1), we can use them for effective capacitance estimation of any application.

V.5.2 Evaluation

We run all the applications in maximum frequency and let Power-Sleuth collect the profiling data. Similarly to performance estimation, we estimate power consumption for each interval separately and then we av-

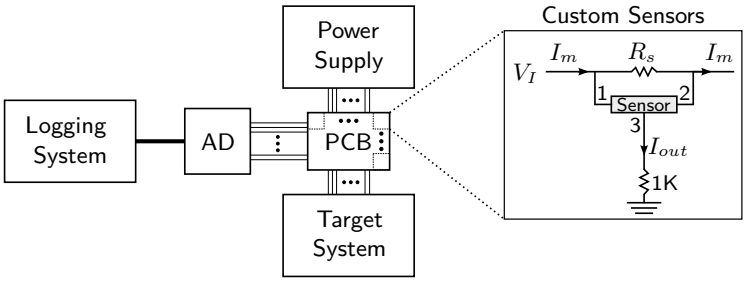


Figure V.10: Our power measurement setup. A PCB board with an array of current sensors is inserted between the power supply and the motherboard of the target system. The outputs of the sensors are tracked by an ADC, which sends the results through a USB connection to a separate logging PC.

erage intervals of the same instance of the phase. Unlike performance prediction, we have to predict power consumption even for the profiling frequency, since no real power measurements are used by the end tool. The left bar in Figure V.7 shows the error when profiling frequency is maximum and we predict for the same frequency. The worst case error is about 6%, while the average error is below 4%. The right bar of the same figure shows the error when from the same data (collected in the profiling run under maximum frequency) we predict power consumption under minimum frequency. Remember that to do so, we first need to use the performance model to get the event rates under minimum frequency, and then we can use the power correlation model to estimate power. The figure shows that we can accurately predict power even when we profile the application under a frequency different than the target frequency.

Finally, Figures **??** and V.9 show power predicted over time for gcc/166 and astar/lakes in maximum and minimum frequency. Each application was profiled in maximum frequency, and the data was used to predict for both frequencies. The figures show that Power-Sleuth successfully tracks power variation over time, with a worst case prediction of about 10% (phase D for gcc/166 in minimum frequency).

V.5.3 Power Measurement Infrastructure

In Sections V.5.1, V.5.2 we used real power measurements to train our model and to evaluate our method. To achieve the high level of accuracy and resolution we need for Power-Sleuth, we created the infrastructure depicted in Figure V.10. We use current sensors [29] to measure the current through each voltage rail supplying the motherboard: the main

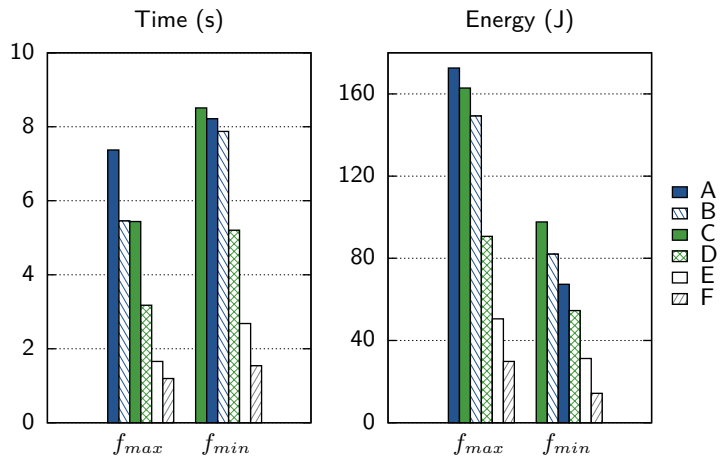


Figure V.11: Per-phase execution time and energy for gcc/166. The 6 most important phases are shown. The figure shows that ranking of phases is different for execution time and energy, and also depends on frequency.

ATX connector 3.3V, 5V and 12V, as well as the separate 12V rail supplying the processor. Though only the latter one is needed for this work, our future work includes extending Power-Sleuth for estimating uncore (L3 cache and memory controller in Intel architectures) and memory power, thus our measuring setup tracks them as well. The current sensors are 3-pin components. Pins 1 and 2 are connected across a sense resistor R_s. The current flows through the resistor and, depending on this current, an output voltage is produced in pin 3. To measure currents for the whole system, we design a PCB which is installed between the power supply and the motherboard, with a current sensor inserted between the two ends of the cables of interest. The outputs of the sensors are connected to a 16-channel Analog-to-Digital data acquisition device capable of sampling 200K samples per second. We use a sampling rate of 1KHz per channel, which is enough for measuring power at a phase granularity and evaluating the accuracy of phase-power prediction.

V.6 Using Power-Sleuth

Having evaluated the accuracy of Power-Sleuth in the previous sections, in this section we demonstrate how a user can employ the tool to characterize and possibly improve the energy efficiency of an application.

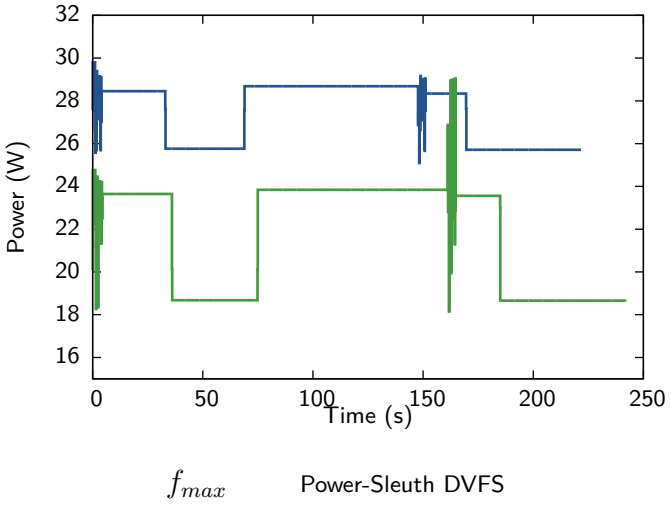


Figure V.12: Power over time predicted by Power-Sleuth for astar/lakes. The two curves show power for maximum frequency, as well as power when a DVFS schedule that aims to minimize EDP within 10% of performance penalty is applied. The schedule was calculated by Power-Sleuth, and the prediction for this schedule is performed without an extra run of the application.

V.6.1 Phase-Energy Characterization

Power-Sleuth can give the user a breakdown of execution time and energy of program phases, for any possible frequency. Figure V.11 shows the 6 most important phases of gcc/166. As the figure shows, ordering the phases by execution time depends on frequency: phase A is the longest running in maximum frequency, but in minimum frequency it is phase C that runs for the longest time. By comparing execution times of the phases for maximum and minimum frequency, it is obvious that phase A is memory bound (execution time is not affected much by frequency scaling), whereas phase C is CPU bound (scaling frequency down results in significant performance overhead). Using this information, the user can select to run phase A in low frequency, by injecting a DVFS command in the source code, at the beginning of phase A. The energy consumed in this case will be reduced from 170J to 70J, as shown in the right part of Figure V.11. Alternatively, since memory is the bottleneck in phase A, one can try to optimize the memory behavior of this phase using appropriate tools [22]. By doing so, power will increase (since the processor will execute instructions at a higher rate), but execution time will decrease, so energy could either increase or decrease: in this case, the user can evaluate both versions of the program using Power-Sleuth

and pick the most efficient code.

Phases B and C, on the other hand, are significantly affected by frequency scaling and thus are CPU-bound phases. Frequency scaling should be avoided in such phases to keep performance at a high level. High energy consumption in these phases comes from high processor utilization, and thus this piece of code is already efficient enough. However, the user can still apply various optimizations, such as improving hit rate in the lower levels of the cache hierarchy or reordering instructions to increase Instruction Level Parallelism, and see how these optimizations impact energy.

V.6.2 Optimal DVFS Schedule

After the profiling run, Power-Sleuth has all the required data to predict performance and energy under any frequency. It is thus capable of providing an optimal DVFS configuration for each phase, according to user specifications. The metric of interest for optimization can be any metric involving performance or power, such as minimum EDP (Energy Delay Product) or ED^2P, or even metrics under constraints (e.g., limited performance penalty, etc.). Figure V.12 shows astar/lakes power consumption over time predicted by Power-Sleuth for maximum frequency, as well as for the recommended DVFS schedule. In this example, minimizing EDP within a performance overhead of 10% for each phase is our optimization criterion. The schedule then can be applied in practice with DVFS commands at the beginning of each phase.

V.7 Related Work

In this section we discuss work related to power profiling and estimation.

Power measurement. Ge et al. [6] developed a tool, called PowerPack, to profile power dissipation of a computer system (processor, memory, disks etc.). The authors use extra hardware to measure power consumption, and they modify the target application by inserting library calls to communicate with the profiler so as to monitor specific code regions. Instead, we only use real power measurements once, to get the characteristics of the processor, and then we estimate power of applications at a program-phase granularity (10s of msec). Moreover, we do not modify the target application, since mapping of power estimation with code is achieved using the phase-detection component.

Power estimation. Joseph and Martonosi [15] correlate power with hardware performance counters to estimate power consumption. They evaluate their method in both a simulator and a real processor. The authors use circuit-level power information and approximate component

activities using heuristics (when this information is not readily available from the performance counters) to get a per-component breakdown of power consumption. Conrteras and Martonosi [1] use a total of 7 performance counter events to estimate core and main memory power consumption in Intel XScale processor. The power model formed can be parameterized in the sense that different regression parameters are used for the different processor Voltage-frequency configurations. More recently, Goel et al. [7] followed a similar approach to get power estimation for multithreaded and multiprocess workloads and employ it for implementing a power-capping scheduler.

Unlike the above mentioned approaches, we correlate *effective capacitance* (the processor's capacitance coupled with node activity factors) [28] with processor performance counters events and then estimate power using Equation (V.4). This approach allows us to have a unified model across all different V-f combinations.

Phase detection. Isci and Martonosi [13] compared control-flow-based (e.g., ScarPhase [24], SimPoint [27]) phase detection with power-event-counter-based [14, 12] phase detection. They used 15 power related performance counters to monitor the execution and to detect phases. They found that event-counter-based phase detection produced slightly more accurate results (i.e., more homogeneous power behavior within phases). However, the goal of their work was to use phase detection for runtime optimizations (i.e., apply new hardware settings at phase changes). In this work, we focus on profiling, and we want to map the power profile back to the code, which control-flow-based phase detection does by default.

V.8 Conclusions and Future Work

In this paper we introduced Power-Sleuth, a tool that estimates performance and power consumption of an application in different frequencies. The tool is capable of characterizing the behavior of an application in any frequency, from a set of data collected in a single frequency. To achieve this, we utilize an analytical DVFS performance model, a novel power-estimation model and a phase detection and classification technique. We show that we can predict power and performance with high accuracy, not only for the whole execution of an application, but also for each program phase individually. Finally, we show use-cases of how the information provided by Power-Sleuth can be used to improve the power-efficiency of an application.

In our future work we plan to extend our methodology to account for memory and uncore power, as well as port and evaluate Power-Sleuth

in more platforms.

V.9 References for Paper V

[1] Gilberto Conrteras and Margaret Martonosi. "Power prediction for intel XScale processors using performance monitoring unit events". In: *Int. Symposium on Low Power Electronics and Design*. 2005.

[2] Ashutosh S. Dhodapkar and James E. Smith. "Comparing Program Phase Detection Techniques". In: *Int. Symposium on Microarchitecture*. 2003.

[3] Richard O. Duda, Peter E. Hart, and David G. Stork. "Pattern Classification". In: 2nd ed. Wiley-Interscience, 2001. Chap. 10.11. On-line Clustering, pp. 559–565. ISBN: 0-471-05669-3.

[4] S. Eyerman and L. Eeckhout. "A Counter Architecture for Online DVFS Profitability Estimation". In: *Computers, IEEE Transactions on* (2010).

[5] Stijn Eyerman et al. "A mechanistic performance model for superscalar out-of-order processors". In: *ACM Trans. Comput. Syst.* (2009).

[6] Rong Ge et al. "PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications". In: *IEEE Transactions on Parallel and Distributed Systems* 21.5 (2010), pp. 658–671.

[7] B. Goel et al. "Portable, scalable, per-core power estimation for intelligent resource management". In: *Int. Green Computing Conference*. 2010.

[8] John L. Henning. "SPEC CPU2006 benchmark descriptions". In: *SIGARCH Comput. Archit. News* (2006).

[9] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Volume 3B: System Programming Guide. 30.4.4 Precise Event Based Sampling (PEBS). Intel Corporation. 2010.

[10] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Volume 3A: System Programming Guide. 14.7 Platform Specific Power Management Support. Intel Corporation. 2010.

[11] *Intel VTune*. URL: `http://www.intel.com/\\software/products/vtune/`.

[12] C. Isci and M. Martonosi. "Identifying program power phase behavior using power vectors". In: *Int. Workshop on Workload Characterization*. 2003.

[13] C. Isci and M. Martonosi. “Phase characterization for power: evaluating control-flow-based and event-counter-based techniques”. In: *Int. Symposium on High-Performance Computer Architecture*. 2006.

[14] Canturk Isci and Margaret Martonosi. “Runtime power monitoring in high-end processors: Methodology and empirical data”. In: *Int. Symposium on Microarchitecture*. 2003.

[15] Russ Joseph and Margaret Martonosi. “Run-time power estimation in high performance microprocessors”. In: *Int. Symposium on Low Power Electronics and Design*. 2001.

[16] Tejas S. Karkhanis and James E. Smith. “A First-Order Superscalar Processor Model”. In: *Proceedings of the 31st annual international symposium on Computer architecture*. 2004.

[17] Stefanos Kaxiras and Margaret Martonosi. “Computer Architecture Techniques for Power-Efficiency”. In: Morgan and Claypool Publishers, 2008. ISBN: 0-471-05669-3.

[18] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. “Interval Based Models for Run-Time DVFS Orchestration in SuperScalar Processors”. In: *Int. Conf. on Computing Frontiers*. 2010.

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

[20] David Levinthal. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*. Tech. rep. Version 1.0. Intel Corporation, 2009.

[21] Nitzan Peleg and Bilha Mendelson. “Detecting Change in Program Behavior for Adaptive Optimization”. In: *Int. Conf. on Parallel Architecture and Compilation Techniques*. 2007.

[22] *RogueWave ThreadSpotter*. URL: `http://www.rogue\\wave.com/products/threadspotter.aspx.`.

[23] B. Rountree. “Theory and practice of dynamic voltage/frequency scaling in the high-performance computing environment”. In: *Ph.D. dissertation, University of Arizona* (2010).

[24] Andreas Sembrant, David Eklov, and Erik Hagersten. “Efficient Software-based Online Phase Classification”. In: *Int. Symposium on Workload Characterization*. 2011.

[25] 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.

[26] Timothy Sherwood, Suleyman Sair, and Brad Calder. “Phase tracking and prediction”. In: *Int. Symposium on Computer Architecture*. 2003.

[27] Timothy Sherwood et al. “Automatically characterizing large scale program behavior”. In: *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. 2002.

[28] Vasileios Spiliopoulos, Stefanos Kaxiras, and Georgios Keramidas. “Green governors: A framework for Continuously Adaptive DVFS”. In: *Int. Green Computing Conference*. 2011.

[29] *ZXCT1009: High-side Current Monitor*. Document number: DS33441 Rev. 12 - 2. Diodes Incorporated. 2011.

Recent licentiate theses from the Department of Information Technology

2012-008 Palle Raabjerg: *Extending Psi-calculi and their Formal Proofs*

2012-007 Margarida Martins da Silva: *System Identification and Control for General Anesthesia based on Parsimonious Wiener Models*

2012-006 Martin Tillenius: *Leveraging Multicore Processors for Scientific Computing*

2012-005 Egi Hidayat: *On Identification of Endocrine Systems*

2012-004 Soma Tayamon: *Nonlinear System Identification with Applications to Selective Catalytic Reduction Systems*

2012-003 Magnus Gustafsson: *Towards an Adaptive Solver for High-Dimensional PDE Problems on Clusters of Multicore Processors*

2012-002 Fredrik Bjurefors: *Measurements in Opportunistic Networks*

2012-001 Gunnika Isaksson-Lutteman: *Future Train Traffic Control – Development and deployment of new principles and systems in train traffic control*

2011-006 Anette Löfström: *Intranet Use as a Leadership Strategy*

2011-005 Elena Sundkvist: *A High-Order Accurate, Collocated Boundary Element Method for Wave Propagation in Layered Media*

2011-004 Niclas Finne: *Towards Adaptive Sensor Networks*

2011-003 Rebecka Janols: *Tailor the System or Tailor the User? How to Make Better Use of Electronic Patient Record Systems*

UPPSALA
UNIVERSITET

Department of Information Technology, Uppsala University, Sweden