

國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science


National Taiwan University

Master Thesis

平行程式於記憶體共享架構之自動相位檢測

Automatic Phase Detection for Parallel Applications on Shared

Memory Architectures

The seal of National Taiwan University is a circular emblem. It features a central bell (the 'University Bell') with two vertical bars on either side. The top arc contains the Chinese characters '臺灣' (Taiwan) and the bottom arc contains '大學' (University). The outer ring of the seal contains the university's name in Chinese: '國立臺灣大學' (National Taiwan University) and the motto '勵品敬人' (Encourage good character, respect people).

黃崇智

Hung Chung-Chih

指導教授：楊佳玲 博士

Advisor: Yang Chia-Lin, Ph.D.

中華民國 99 年 7 月

July, 2010

國立臺灣大學碩士學位論文  
口試委員會審定書

平行程式於記憶體共享架構之自動相位檢測

Automatic Phase Detection for Parallel Applications on  
Shared Memory Architectures

本論文係黃崇智君（學號 R97922129）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 99 年 7 月 27 日承下列考試委員審查通過及口試及格，特此證明

口試委員：



龍吉昇

(指導教授)



洪士龍

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

系主任

呂育道

\_\_\_\_\_

# 謝詞

首先，能夠順利的畢業，我要感謝我的指導教授—楊佳玲老師，謝謝老師兩年來的教導，研究上老師的諄諄教誨以及待人接物上的許多意見及幫助，讓我獲益良多、更加成長。口試時，謝謝口試委員們所提供的指教與建議。

特別感謝博士班的學長林仲祥，給予我在論文上的許多方向與幫助，透過和他的討論，幫我找到處理問題的方法，以及從學長身上學到許多，從事研究的態度與方式。以及感謝實驗室的學長姐、同學陪我度過整個研究生涯。不吝給予我寶貴的意見與教導，在研究與生活上帶給我相當多啟發。另外感謝學弟在過程中，和我互相幫忙，彼此加油打氣。

最後要特別感謝我的家人與舊憶，無論遇到什麼事情，總是在一旁給予我支持與鼓勵，陪我一路走來，沒有你們的支持，也就沒有今天的我。

黃崇智 謹上

# 摘要

週期精確之軟體模擬器對於計算機結構設計相當重要。它允許設計者在早期設計時可以嘗試不同的計算機結構。然而，週期精確軟體模擬器，其模擬速度相當緩慢。而多核心處理器架構因較單核心系統有更多之 CPU 與其他系統元件，因此模擬速度更加緩慢，使得改善多核心系統之模擬器的效能極為重要。

在本論文中，我們探討取樣模擬的技術，並提出以此技術加速多核心系統模擬之機制。透過辨認程式中重複出現的行為，考慮每個具代表性的特徵點，來加速模擬的時間。對於偵測程式重複行為的問題，傳統上對單一執行緒程式，採用程式碼簽章的方式。然而對於平行程式，程式的行為不再只受執行指令的影響，執行緒間彼此的互動，也成為影響表現的重要因素。

因此在傳統程式碼簽章的方法之外，我們還利用紀錄執行緒間資料共享的模式，與共用資源爭搶的情況，來幫助偵測程式重複出現的行為特徵。藉由採計特徵點的行為與程式完整執行結果比較，我們所設計之多核心模擬加速方法，能將錯誤率控制在 2% 以下。

**關鍵字** — 多核心模擬(multi-code simulation)、程式行為注釋(application annotation)、特徵驅動取樣機制(profile-driven sampling)、取樣模擬(simulation sampling)、相位偵測(phase detection)

# Abstract

Cycle-accurate software-based simulation is critical for architecture design since it allows an architect to explore various architectural design points at the early stage of design cycles. However, simulation speed has always been an issue for cycle-accurate simulation. With the popularity of multi-core processors, improving multi-core simulation performance is critical to allow fast advances in multi-core architecture researches. In this work, we look into simulation sampling techniques to speed up multi-core architecture simulation.

Techniques have been proposed that automatically group similar portions of a program's execution into phases, where samples classified as the same phase have homogeneous behavior. Conventionally, a program is looked over code signatures to extract information about the phases and only the representative intervals are executed to analyze architectural selections. However, such methodologies are becoming inadequate in multi-core category. Because application's behavior is not dominated by the instructions only but also the communication structures between threads.

Hence, in this work we propose to utilize the interaction between threads for parallel program phase detection. Our results reveal that the inclusion of such information can increase the accuracy of the phase detection signifi-

cantly (The error rate of IPC is below 2%).

**Keywords** — multi-core simulation, application annotation, profile-driven sampling, simulation sampling, phase detection



# Contents

<b>Abstract</b>	<b>i</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Related Works</b>	<b>5</b>
<b>3 Phase Identification Method For Serial Programs</b>	<b>9</b>
3.1 Profiling Program Behavior . . . . .	11
3.2 Using $k$ -means for Phase Classification . . . . .	11
<b>4 Phase Behavior Analysis For Parallel Programs</b>	<b>14</b>
4.1 Data Level Parallelism Programs . . . . .	14
4.2 Task Level Parallelism Programs . . . . .	18
<b>5 Phase Identification Method For Parallel Programs</b>	<b>23</b>
5.1 Thread Interaction On CMPs . . . . .	24
5.2 Thread Interaction Aware Phase Detection . . . . .	27
<b>6 Experimental Setup</b>	<b>30</b>
6.1 Benchmarks . . . . .	31
6.2 Metrics for Evaluating Phase Classification . . . . .	34

<b>7 Experimental Results</b>	<b>36</b>
7.1 Proposed Scheme . . . . .	36
7.2 Comparison with Previous Work . . . . .	38
7.2.1 Baseline . . . . .	38
7.2.2 SBBV+CCV VS. ICV+TCV . . . . .	39
7.2.3 SBBV+CCV VS. ICV+CCV . . . . .	41
7.2.4 SBBV+CCV VS. SBBV+TCV . . . . .	42
7.3 Memory Contention Count Vector . . . . .	43
<b>8 Conclusion</b>	<b>45</b>
<b>Bibliography</b>	<b>46</b>





# List of Figures

1.1	characterizing program behavior [28]	2
4.1	phase behavior of data level parallelism program : quake	15
4.2	phase behavior of data level parallelism program : wupwise	16
4.3	phase behavior of data level parallelism program : swim	17
4.4	Loop-by-loop of data sharing [5]	17
4.5	phase behavior of task level parallelism program : dedup	19
4.6	phase behavior of task level parallelism program : ferret	20
4.7	phase behavior of task level parallelism program : x264	21
4.8	Traffic from cache in bytes per instruction for 1 to 16 cores. Data assumes a shared 4-way associative cache with 64 byte lines. [8]	21
5.1	timing alignment for parallel programs	24
5.2	data sharing pattern for parallel programs	25
5.3	resource contention for parallel programs	26
5.4	Communication Count Vector (CCV)	27
5.5	Memory Contention Count Vector	28
7.1	Error Rate of SBBV, CCV, SBBV+CCV	37

7.2	Phase-Based Standard Deviation of SBBV, CCV, SBBV+CCV	
	37	
7.3	Error Rate of SBBV+CCV, ICV+TCV . . . . .	40
7.4	Phase-based Standard Deviation of SBBV+CCV, ICV+TCV	40
7.5	Error Rate of SBBV+CCV, ICV+CCV . . . . .	41
7.6	Error Rate of SBBV+CCV, SBBV+TCV . . . . .	42
7.7	Error Rate of SBBV+CCV, SBBV+CCV+MCV . . . . .	43



# List of Tables

6.1	Profiled Machine Configurations . . . . .	35
6.2	Evaluation Machine Configurations . . . . .	35
7.1	Working Set of Applications . . . . .	43



# Chapter 1

## Introduction

Microarchitects and researchers generally need understanding the cycle accurate behavior of a processor during the execution of an application. Researchers typically obtain this information by means of employing detailed simulators, which simulate complete flow of each instruction going through the microprocessor pipeline, in order to capture the timing information of individual cycles more accurately.

However, cycle level of detail brings about the cost of speed, and simulating the complete execution of an industry standard benchmark can spend weeks or months to finish. To make things worse, architecture researchers find the set of features that provides the best trade-off between performance, complexity, area, and power by simulating each benchmark over a variety of architecture configurations. Hence, there is a need to develop a technique which can reduce the number of machine-months to estimate the impact of an architectural modification without coming at an unacceptable error rate or excessive simulator complexity.

Previous works have shown that programs exhibit cyclic behavior pat-

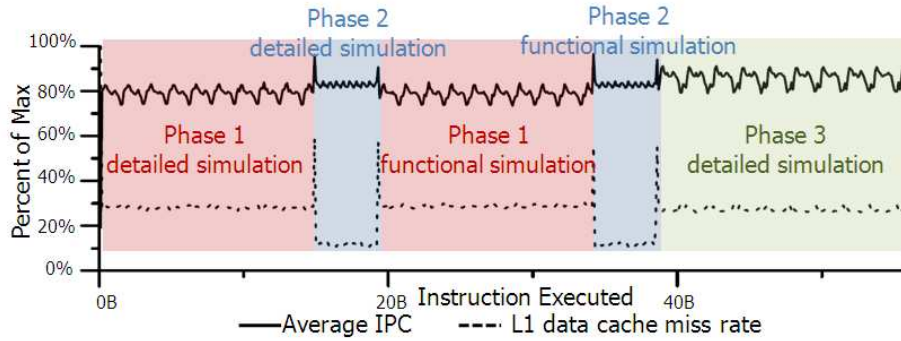


Figure 1.1: characterizing program behavior [28]

terns. So the behavior of a program is not random and based on such observation, the terminology phase is defined. An interval is a slice in time of a program's execution. A phase is a set of intervals within an application execution which performs homogeneous behavior. The intervals are not necessarily temporally contiguous. In other words, a phase can reoccur many times throughout the execution.

One of the most common means of using phase information is to reduce the architectural simulation time. At run-time, programs exhibit repetitive behaviors that change over time. These behavior patterns provide an opportunity to reduce simulation time. For example in fig 1.1, by identifying each of the repetitive behavior (Phase 1 to Phase 3), taking only a single sample of each repeating behavior and weighting each sample according to the relative size the phase represents from the complete execution (Phase 1:  $2/5$ , Phase 2:  $2/5$  and Phase 3:  $1/5$ ), we can perform very fast and accurate sampling (rather than running the application to completion, which takes considerable simulation time). Such approaches reduce the overall simulation time by order of magnitude. Simulating only these carefully chosen simulation points

can save hours to days of simulation time with very low error rates.

The phase detection technique for serial programs is already a mature area. Phase information has been extracted by utilizing code signatures (e.g. basic block vector ). Results show that the detection of phases through such technique is accurate. However, the microprocessor industry went through a paradigm shift in the recent years from a single core, complex design that pushed the power envelope and clock rates to multi-core designs with simpler cores. Many manufactures, including Intel, IBM, Sun and AMD, have produced microprocessors which incorporate multiple processing cores. The increasing number of cores has resulted in the communication pattern between cores playing an important role in determining the overall performance of the processor. The behavior of program is no longer dominant by the instructions only.

Hence, we propose to make use of information about interactions between threads to detect the execution phases. We introduce CCV (Communication Count Vector) which is a one-dimensional array recording the communication messages issued by corresponding core. In addition, we propose MCV (Memory Contention Count Vector) which is also a one-dimensional array recording the last level cache misses issued by corresponding core. Then, we combine CCV and MCV with the code signature based structure, BBV (Basic Block Vector) to detect parallel program phases. The experimental results show that on average the IPC error rate is below 2% and the standard deviation for IPC is reduced by 46.46% compared to program complete execution.

The rest of this thesis is organized as follows: Chapter 2 reviews the

related works on phase classification and analysis. Chapter 3 describes the phase identification method for serial programs. Chapter 4 shows the phase behavior analysis for parallel programs. Chapter 5 presents the proposed phase identification for parallel programs, including Sampled Basic Block Vector(SBBV) technique, Communication Count Vector(CCV), Memory Contention Count Vector (MCV) and combination of the above all techniques. Chapter 6 presents the experimental setup. Chapter 7 shows the experimental results on the accuracy of IPC metric and standard deviation reduction could be derived by the proposed methodology. Finally, Chapter 8 states the conclusion about this work.



# Chapter 2

## Related Works

Several researchers have examined phase behavior in programs. In this section we will give a brief description of studies related to phase identification and phase-based optimization.

Sherwood et al. [26] presented that applications show cyclic phase-based behavior over many architecture properties, such as cache behavior, branch prediction, value prediction, address prediction, IPC and RUU occupancy, etc. Repeating patterns were found in many programs, and the essential architecture metrics show similar behavior over time.

There are works utilizing hardware counters for phase detection. Isci et al. [16, 17, 18] used hardware performance counters to exploit phase behavior in programs. And they have shown the ability to dynamically identify the power phase behavior using power vectors. Deusterwald [13] et al. used hardware counters and other phase detection scheme to analyze program's phases.

The phase information can be used for dynamic reconfiguration. Balasubramonian et al. [7] proposed using hardware counters to collect miss



rates, CPI and branch frequency information for every hundred thousand instructions. They dynamically evaluated the program’s stability by utilizing the miss rate and the total number of branches executed for each interval. They used such technique to guide dynamic cache reconfiguration to save energy without sacrificing performance. Dhodapkar and Smith [12, 10, 11] utilized a relationship between phases and instruction working sets, and they found that phase changes at the same time as the working set changes. This directed them to propose dynamic reconfiguration of instruction cache, data cache and branch predictor in response to phase changes indicated by working set changes in order to save energy [10, 11].

Some works focus on how to appropriately define the granularity and similarity to perform phase analysis. Hind et al. [21] provided a framework to present the impact of granularity and similarity on phase analysis. And they showed how to perform phase analysis appropriately. Lau et al. [20] examined the influence of varying interval length in phase detection. Vandeputte and Eeckhout [30] provided phase complexity surfaces to characterize a program phase behavior across various time scales. Cho and Li [9] proposed an approach to quantitatively analyze the changing of phase dynamics across different time scales. And they presented a framework classifying phases which exhibit homogeneity in their scaling behavior.

The most common means of using phase information is to reduce the architectural simulation time. In [27, 28], Sherwood et al. proposed that programs have repeatable phase-based behavior, and such behavior can be automatically identified by only examining code execution. They used techniques from machine learning that are capable of finding and exploiting the

large scale behavior of program. They found that intervals of execution classified as the same phase had similar results over all architecture metrics. SimPoint [28] was developed to automatically pick a small set of intervals of execution in the program for detailed simulation. They also extended their work [19, 29] to perform hardware phase classification and prediction. Patil et al. [23] has inspected guiding simulation at Intel by utilizing simulation points which are picked by SimPoint. Davies et al. [6] exploited sampled information extracted by the Intel's VTune Performane Analyzer, in order to construct a representation of program execution within a given interval. Such information is collected at runtime on native hardware. Annavaram et al. [4] employed the VTune approach to examine phase behavior for database applications.

This idea is also extended for parallel programs. Perelman et al. [25] used the sampled information extracted by VTune to collect Extended Instruction Pointer Vectors. They produced Sampled Basic Block Vectors to examine applying phase analysis algorithms and how to adapt them to parallel applications running on shared memory processors. But they didn't take the communication pattern between threads into consideration. Zhang et al [31] proposed to utilize communication behavior to determine the phases of a parallel application running on Network-On-Chip architecture. Hence, they presented TCV (Taffic Count Vector) which is established by recording the number of packets going through each router. However, because we think there is still room for improvement, we propose Communication Count Vector (CCV) and Memory Contention Count Vector (MCV) (There are more details in chapter 5) in order to capture the interaction between threads more

accurately. We will regard their methodology as baseline to compare with our work.



# Chapter 3

## Phase Identification Method For Serial Programs

The phase detection technique for serial programs is already a mature area. Phase information has been extracted by utilizing code signatures (e.q. basic block vector) and the results show such technique is accurate. In this section, we will briefly describe the methodology utilized in [28] to detect serial program phases.

The following are definitions helping us to ground our discussion in a consistent vocabulary.

### 1. Interval

A contiguous portion of execution (a slice in time) of a program. Program execution is divided into contiguous non-overlapping intervals. All intervals are measured based on instruction count (total number of executed instructions).

### 2. Phase

A set of intervals within a program execution which perform homogeneous characteristics. A phase can consist of intervals that are not necessarily temporally adjacent, so a phase can reoccur many times throughout execution.

### 3. Frequency Vector

Each interval is represented by a frequency vector, which stands for the program behavior during that time slice. The Basic Block Vector (BBV) is the most commonly used structure. There would be more details about BBV in section 3.1.

### 4. Similarity Metric

The Euclidean distance between the two frequency vectors is regarded as the similarity between two corresponding intervals. The Euclidean distance can be calculated by considering each frequency vector to be a single point in D-dimensional space (D stands for the number of elements in each frequency vector), and calculating the straight-line distance between the two points. The formula for computing the Euclidean distance of two vectors  $a, b$  in D-dimensional space is given by:

$$EuclideanDistance(a, b) = \sqrt{\sum_{i=1}^D (a_i - b_i)^2} \quad (3.1)$$

### 5. Phase Classification

Intervals are classified into phases according to similarity metric. Intervals with similar behavior are grouped into the same phase.

### 3.1 Profiling Program Behavior

Each interval is represented by a frequency vector. Basic Block Vector (BBV)[27] is the most commonly used frequency vector. A basic block is a single-entry, single-exit section of code with no internal control flow. Basic Block Vector (BBV) is an one-dimensional array where each element represents the number of time a basic block is entered during the execution interval. At the beginning of each interval, Basic Block Vector (BBV) contains all zeros. And as the program executes, the number of times each basic block has been entered is recorded for current interval. The intuition behind this is that the behavior of the program at a given time is directly related to the code executed during that interval. In [28], they used the basic block vectors as signatures for each interval of execution: each vector gives the information about what portions of code are executed, and how frequently those portions of code are executed. By comparing the BBVs of two intervals, we can evaluate the similarity of the two intervals.

### 3.2 Using $k$ -means for Phase Classification

The basic block vectors provide a compact and representative summary of the programs behavior for intervals of execution. Because there are so many intervals of execution that are similar to one another, one efficient representation is to group the intervals together with similar behavior. This problem is analogous to clustering problem.

Clustering splits points into clusters, so that the points classified as the same cluster are similar with each other (by some metric, usually distance),

and points from different cluster are different from one another. In [28], they used the efficient and notable algorithm,  $k$ -means [22] to fast and accurately divide the program behavior into phases. The  $k$  in  $k$ -means stands for the number of clusters the algorithm will search for. The following steps brief the phase clustering algorithm at a high level.

1. Divide the program's execution into contiguous intervals, and profile the program by recording a frequency vector for each interval. Each frequency vector is normalized so that the sum of all the element equals 1.
2. Apply the  $k$ -means clustering algorithm on all of the collected-vectors for a set of  $k$ -values.
3. In order to select a well-formed clustering that also has a small number of clusters from these different clusterings, in [28], they used the *Bayesian Information Criterion (BIC)* [24] to compare and evaluate the different clusters formed for different value of  $k$ . The *Bayesian Information Criterion (BIC)* is regarded as a measure of "goodness of fit" of a clustering to a dataset. They selected the clustering with the smallest value of  $k$ , such that its BIC score is above some percentage of the range of scores that have ever seen. The final clustering result stands for the way the intervals are classified into phases.
4. The final step is to decide simulation points for the chosen clustering. For each phase, an interval that is the closet to the *centriod* (center of each cluster) is selected. Each simulation point also has a associated weight, which are related to the fraction of executed instructions in the

program its phase stands for.

5. Eventually, a weighted-average architecture metrics of concern (IPC, miss rate, etc) is found out. From the weights and the detailed simulation results of each simulation points, we can get the weighted-average architecture metrics. This information give us an accurate representation of the entire execution for specific application.





# Chapter 4

## Phase Behavior Analysis For Parallel Programs

In this section, we will analysis the phase characterization of data level parallelism program and task level parallelism programs, in order to realize the phase behavior of various kind of applications.

### 4.1 Data Level Parallelism Programs

Figure 4.1, 4.2 and 4.3 graphically show the phase behavior for the SPEC OMP benchmark *equake*, *wupwise* and *swim* when using 4 threads with respect to execution time. The  $x$ -axis shows the execution time. The  $y$ -axis which shows each thread's IPC (Instruction Per Cycle) is partitioned into four sections, one per thread. SPEC OMP is composed of data level parallelism applications. The majority of code sections were transformed into parallel form by searching for loops with fully independent iterations. So all the threads execute the identical tasks at the same time and there is

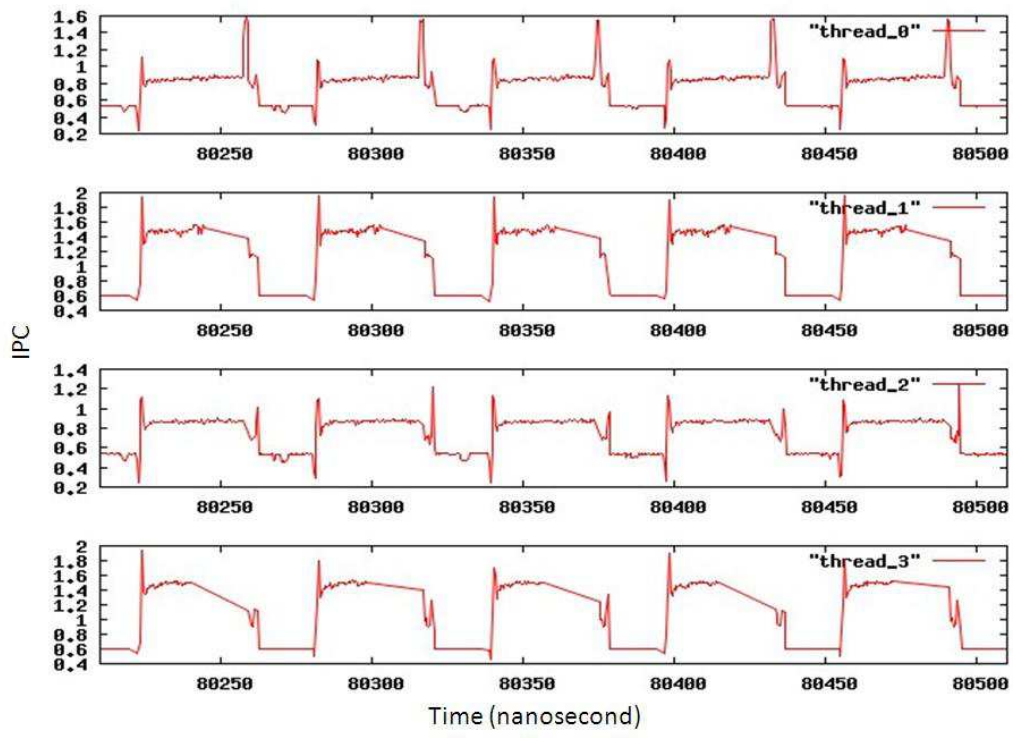


Figure 4.1: phase behavior of data level parallelism program : equake

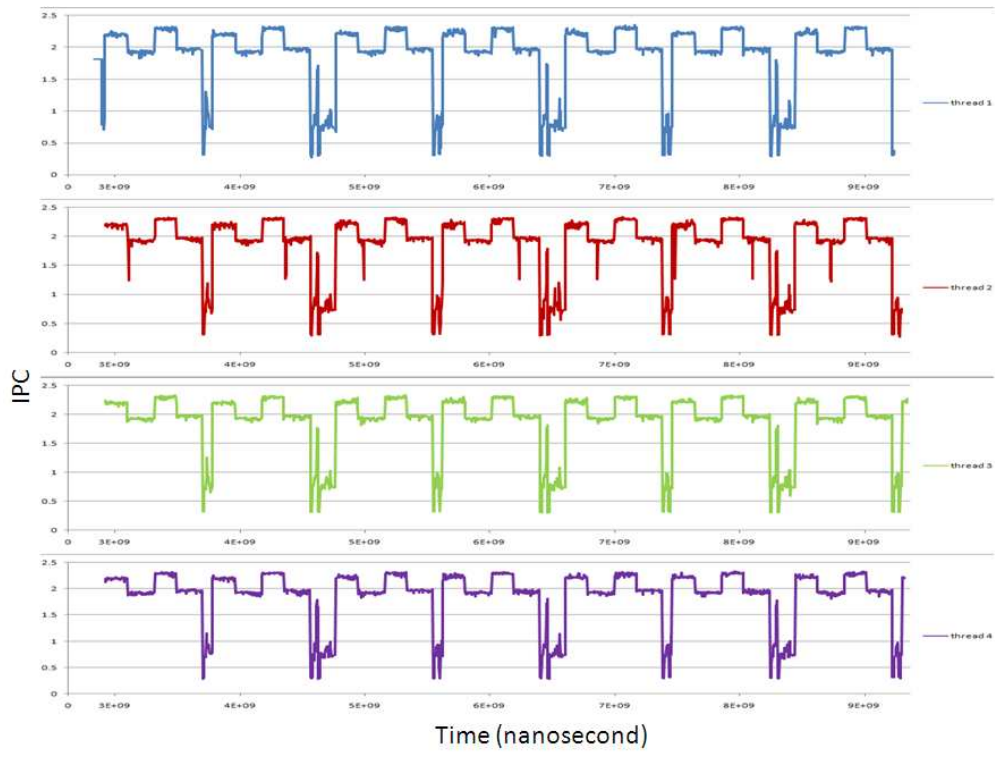


Figure 4.2: phase behavior of data level parallelism program : wupwise

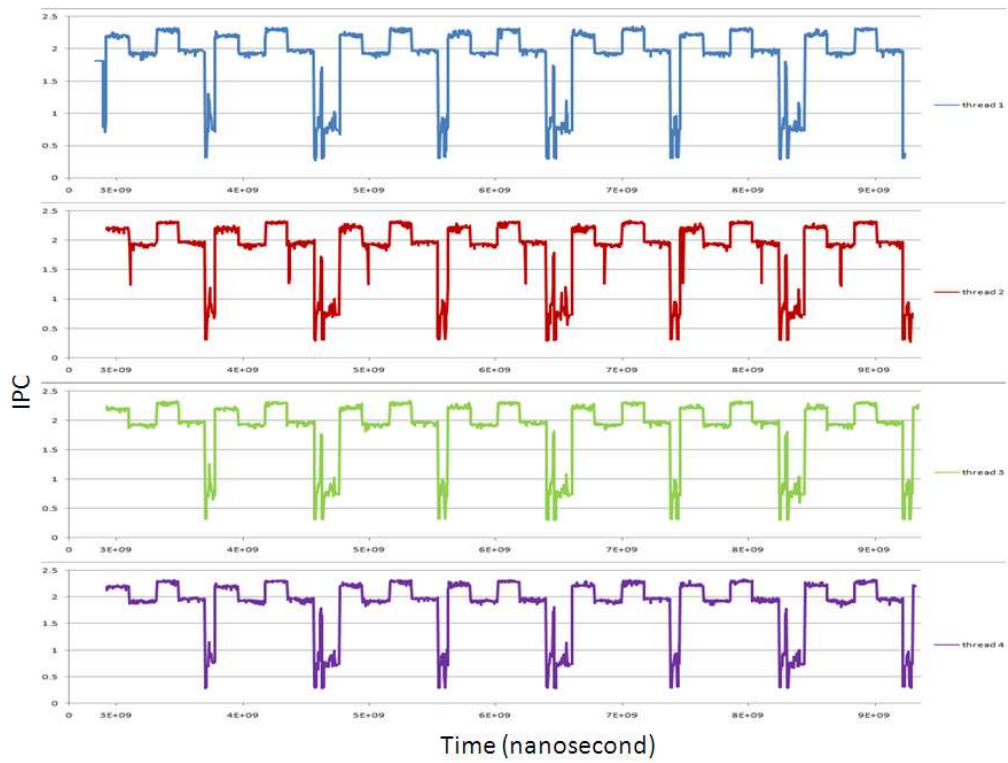


Figure 4.3: phase behavior of data level parallelism program : swim

Benchmarks	Loop Name	Invalidations			Copybacks		
		Seq.	2 Proc.	4 Proc.	Seq.	2 Proc.	4 Proc.
equake	Smpv-#0	0.00	0.00	0.00	0.00	0.00	0.00
	Main-#3	0.00	0.00	0.00	0.00	0.00	0.00
swim	Calc3-do#300	0.00	0.00	0.00	0.00	0.00	0.00
	Calc2-do#200	0.00	0.00	0.00	0.00	0.00	0.00
	Calc1-do#100	0.00	0.00	0.00	0.00	0.00	0.00
	Swim-do#400	0.00	0.00	0.00	0.00	0.00	0.00
wupwise	Muldoe-do#1	0.00	0.00	0.00	0.00	0.00	0.00
	Mudeo-do#1	0.00	0.00	0.00	0.00	0.00	0.00
	Zaxpy-do#1	0.00	0.00	0.00	0.00	0.00	0.00
	Zdotc-do#1	0.00	0.00	0.00	0.00	0.00	0.00
	Zcopy-do#1	0.00	0.00	0.00	0.00	0.00	0.00

Figure 4.4: Loop-by-loop of data sharing [5]

little communication between threads.

Figure 4.4 [5] shows measurements related to the level of data sharing in the parallelized loops in SPEC OMP programs. Such information indicates the interaction between threads in *wupwise*, *equake* and *swim* is very low. When several processors share the same data and one of the processor updates the data, the processor will ask all other processors to invalidate their data in order to prevent using the stale data. Such act is called invalidation transaction. Because Invalidation can cause cache missed, the updated data can be obtained from another processor which has the most up-to-date copy. Such act of copying a cache line from one processor's cache to another processor's cache is called a copyback transaction. Hence the number of *Invalidations* and *Copybacks* in figure 4.4 stand for the degree of data sharing between threads, and this figure reveals the interaction between these programs is little. Consequently, all threads are in the same phases at the same time for the majority of the execution. And the behavior of this kind of applications is highly dependent on the section of code be executed.

## 4.2 Task Level Parallelism Programs

Figure 4.5, 4.6 and 4.7 graphically show the phase behavior for the Parsec benchmark *dedup*, *ferret* and *x264* when using 4 threads with respect to execution time. These applications are task level parallelism program. Every thread executed different sections of code, and there is large amount of shared data passed from thread to thread. Figure 4.8 [8] analyzes how the task-level programs use its data. The chart shows what data is accessed and how intensely it used. PARSEC workloads use a significant amount of com-

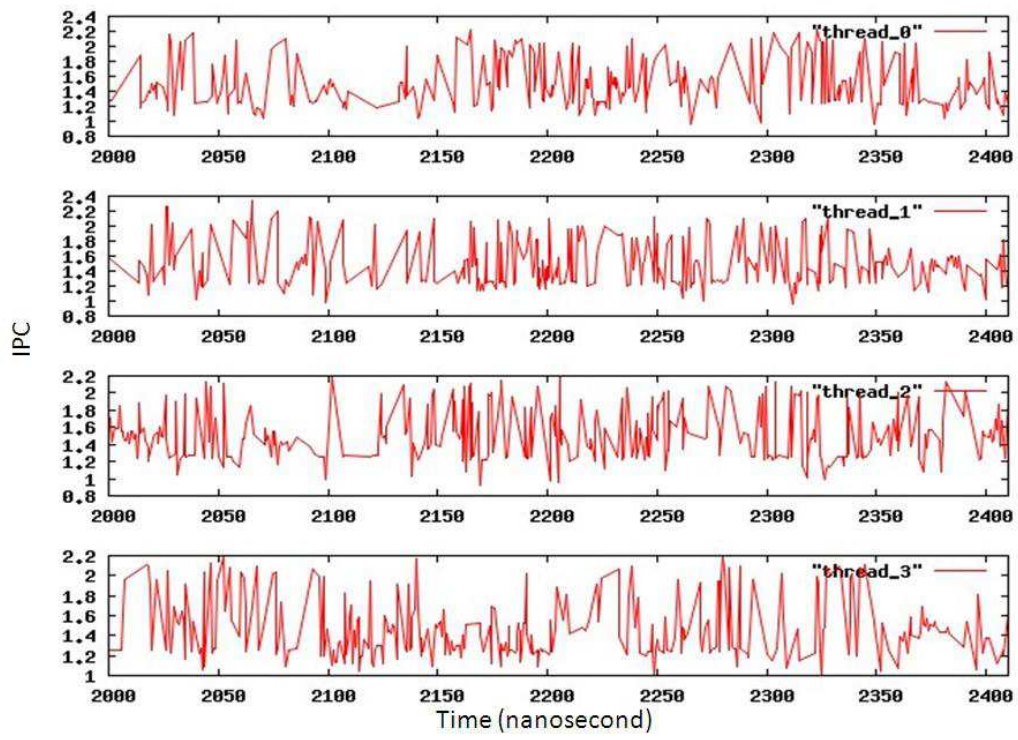


Figure 4.5: phase behavior of task level parallelism program : dedup

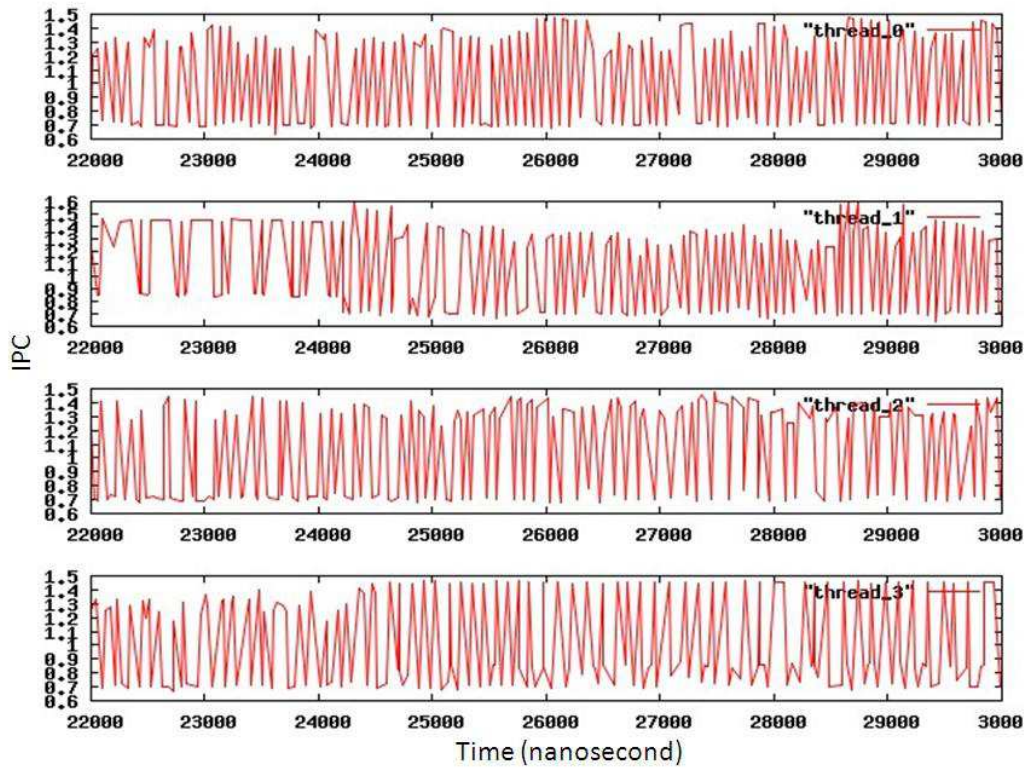


Figure 4.6: phase behavior of task level parallelsim program : ferret

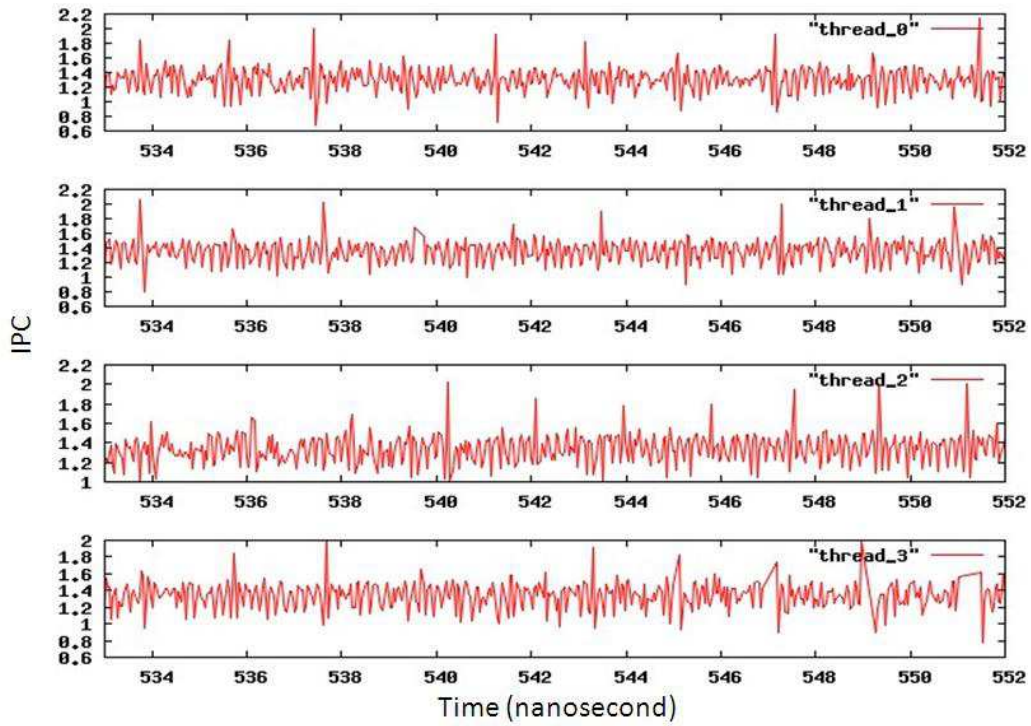


Figure 4.7: phase behavior of task level parallelism program : x264

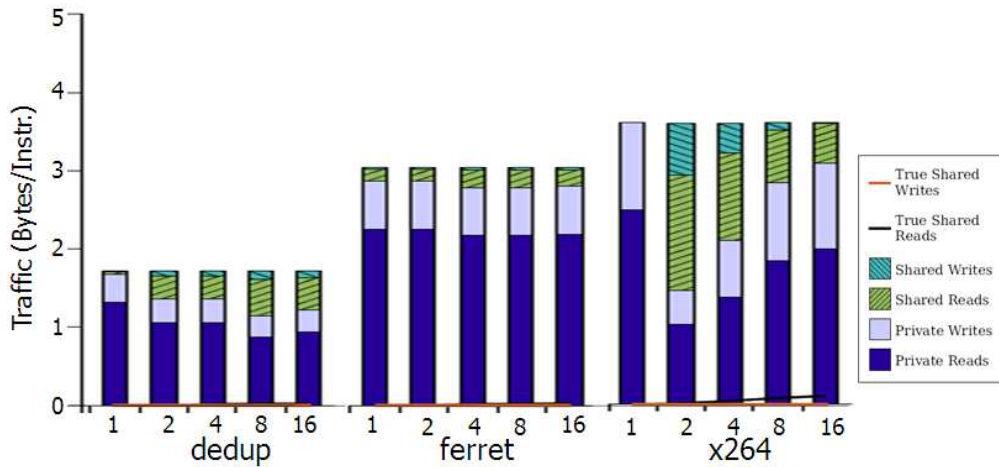


Figure 4.8: Traffic from cache in bytes per instruction for 1 to 16 cores. Data assumes a shared 4-way associative cache with 64 byte lines. [8]



munication, and in many cases the traffic between threads can be so high. For this reason, the phase characterization of this kind of applications is much more complex. The phase behavior is not only affected by the section of code be executed, but also influenced by the active interaction between threads. In the following section, we will describe how to utilize the communication structures between threads to capture parallel program's phase behavior.



# Chapter 5

## Phase Identification Method For Parallel Programs

From the intuition of that the program's behavior at a given time has a strong correlation to the section of code be executed, phases can be detected by utilizing information about basic blocks. The Basic Block Vector (BBV) is the most commonly used structure. Such phase detection technique is successful for serial programs.

However, when the number of cores scales up, generating the basic block vector becomes complicated. Recording which basic block is entered for each instruction is not only time-consuming, and would produce too much data to be maintained.

In [30], Lau et al. proposed that mapping the EIPs back to the static code constructs to create Sampled Code Vectors for each interval where each dimension was the number of times each static loop, procedure, or basic block was sampled. In this work, we construct BBV in a sample manner to detect program phases. At regular intervals, Perfmon2 interrupts execution to

### Situation 1

	Interval 1	Interval 2	Interval 3	Interval 4
Thread 1	Phase 1	Phase 2	Phase 1	Phase 2
Thread 2	Phase A	Phase B	Phase A	Phase B

### Situation 2

	Interval 1	Interval 2	Interval 3	Interval 4
Thread 1	Phase 1	Phase 1	Phase 2	Phase 2
Thread 2	Phase A	Phase B	Phase A	Phase B

Figure 5.1: timing alignment for parallel programs

record the interrupted instruction address. We produce the mapping by using Pin to instrument binaries and map the interrupted instruction addresses down to basic blocks. After that, we utilize this information to create Sampled Basic Block Vectors (SBBV). Then, we use the Sampled Basic Block Vectors (SBBV) to find the phase behavior in parallel programs.

However, only exploiting the information about Basic Blocks are becoming insufficient. In the multi-core era, the application execution is no longer dominated by the instructions only, but instead the interaction between threads in parallel application is becoming as important as the instruction behavior.

## 5.1 Thread Interaction On CMPs

There are three major issues should be concerned in parallel applications:

1. **Timing Alignment**

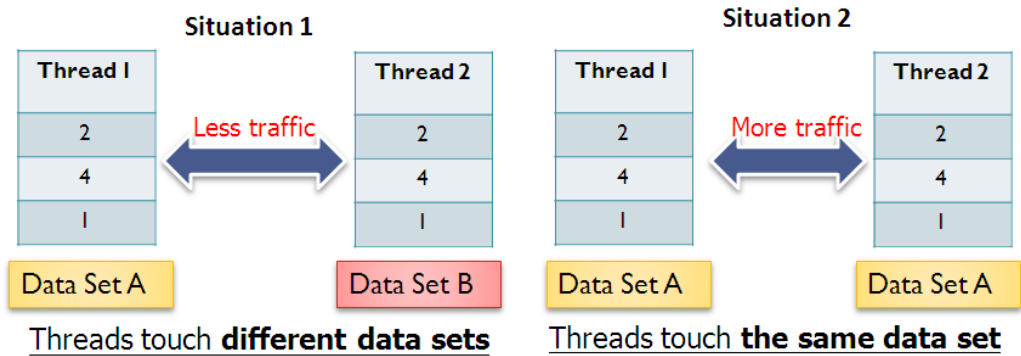


Figure 5.2: data sharing pattern for parallel programs

In order to identify the repeat pattern of parallel programs, we need to know which parts of each thread are executed at the same time, because the behavior of parallel programs is influenced by each individual thread's execution. For example in figure 5.1, since the phase behavior of the two situations is different, we must identify the difference between situation 1 and situation 2 to understand which parts of each thread are executed simultaneously on the system.

## 2. Data Sharing Pattern

To guarantee the correctness, accelerate the execution, and communicate between threads, there are shared data read or written by more than one thread in multi-threaded application. The data sharing pattern could have dominant influence on program's behavior. As we can see in figure 5.2, threads in the two situation execute the same basic blocks. But because threads in situation 2 touch the same data set, there must be much more traffic between threads than situation 1. And the phase behavior of these two situation would be distinct from each other. Therefore, capturing the data sharing pattern between threads

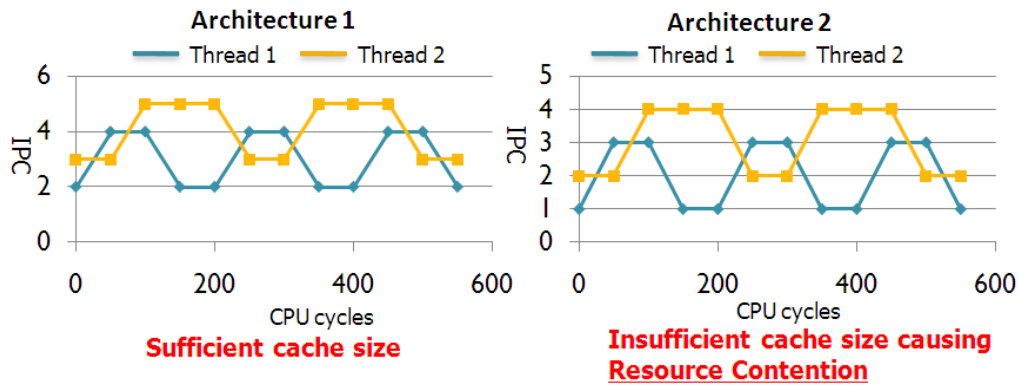


Figure 5.3: resource contention for parallel programs

plays an important role in parallel program phases detection.

### 3. Resource Contention

Through code signature, sampling points in single-threaded programs can be easily picked independently from detailed simulation. But for multi-threaded applications, if there are multiple threads running at the same time, threads share the hardware resources. And these threads may affect each other due to competition for shared resource. For example in figure 5.3, insufficient cache size would cause threads compete for shared resource in architecture 2. And such contention would affect not only the performance of the machine but individual thread execution. Contrary, threads in architecture 1 would not affect each other, since there is sufficient cache size. Therefore, be aware of the shared resource contention between threads is also important for parallel program phases detection.

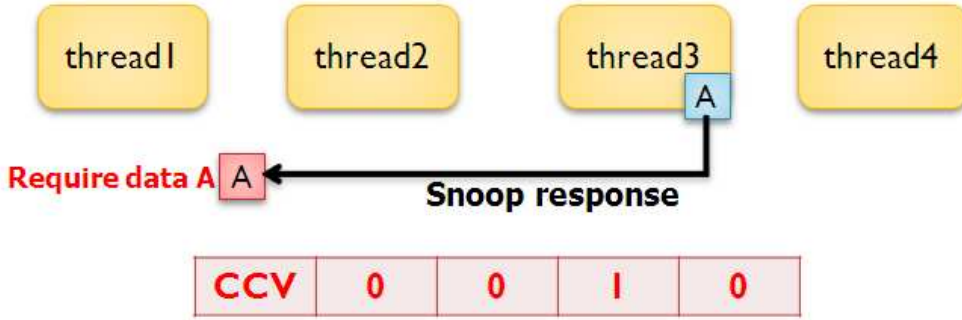


Figure 5.4: Communication Count Vector (CCV)

## 5.2 Thread Interaction Aware Phase Detection

In this section, we will describe the techniques proposed to deal with the above three issues.

### 1. Global Instruction Count

To obtain the information about which parts of each thread are executed together, the interval we use is based on global instruction count (total number of instructions executed by all of the cores). This information help us reconstruct the execution across all threads.

### 2. Communication Count Vector

To capture the data sharing pattern between threads, in this work we introduce Communication Count Vector (CCV), which is also a one-dimensional array. Each element in Communication Count Vector (CCV) records the number of coherence messages issued by the corresponding core during each interval. Such messages are responses to

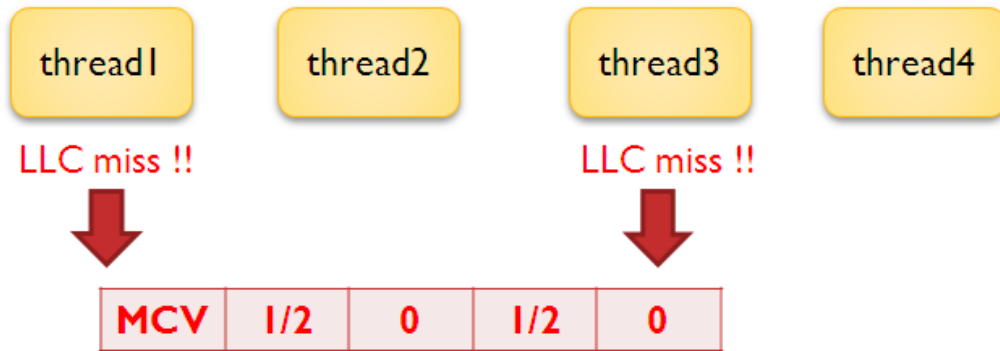


Figure 5.5: Memory Contention Count Vector

other core’s snooping for shared data, when corresponding core holds this data. For example in figure 5.4, thread 1 requires shared data *A*, and thread 3 has the most up-to-date of such data. Since thread 3 respond hit for data *A*, CCV would record this event in the third entry. Identical CCVs of two intervals would imply identical data sharing patterns of two intervals.

### 3. Memory Contention Count Vector

In order to be aware of the situation of hardware resource contention. In this work we introduce Memory Contention Count Vector (MCV), which is also an one-dimensional array. Each element in Memory Contention Count Vector (MCV) is related to the number of last level cache misses in each interval. Because the working set of parallel program is easily larger than size of last level cache and such event is highly influential for program performance, we focus on last level cache miss. As figure 5.5 shows, when thread 1 and thread 3 encounter last level cache miss, MCV would record such events in the first and third entry.

Then, we combine the SBBV, CCV and MCV into one vector and detect phases by using this hybrid profiling information.





# Chapter 6

## Experimental Setup

In this work, on the profiled machine we first use Perfmon2 [2] to profile the target application’s behavior, and then use SimPoint [14] to pick up a set of simulation points based on the collected information. After that, we calculate the phase-based architectural metrics on the evaluation machine by only taking the set of simulation points into account. Finally, we validate our methodology by comparing the phase-based architectural metrics with architectural metrics from complete execution. There are more details in section 6.2.

Perfmon2 [2] is able to non-intrusively analyze any application being executed on real hardware with little overhead. It collects information, such as IPC and interrupted instruction address, which are then used to perform code clustering, phase analysis, and validation. The underlying Perfmon2 driver could monitor a large amount of performance/code execution attributes stored in the embedded event counters of the Perfmon2-available processors during a program is running on native hardware. Our phase analysis framework processes the Perfmon2 output file which is collected from

program's execution. We implement python program processing such output file to get the necessary data , including hardware event counters related to our utilized techniques (SBBV, CCV and MCV) and performance data which allows us to later validate our work.

SimPoint [14] is utilized to automatically identify program behavior. We classify the execution intervals into a number of phases by using  $k$ -means algorithm of SimPoint. SimPoint clusters over a range of values to decide the number of phases and uses Bayesian Information Criterion (BIC) to measure the goodness of each clustering.

## 6.1 Benchmarks

We examine the applications with SPEC OMP [3] and Parsec[1] benchmark in this work. The former is composed by data-level parallelism applications and the latter is composed by task-level parallelism applications. The following are the applications we experimented :

### 1. OMP2001

SPEC OMP2001 consists of a set of OpenMP-based application programs, which represent the type of software used in scientific technical computing. Following is the OMP2001's applications we report in this work :

#### (a) WUPWISE

(Wuppertal Wilson Fermion Solver) is a program in the field of lattice gauge theory. Lattice gauge theory is a discretization of quantum chromodynamics. Quark propagators are computed within

a chromodynamic background field. The inhomogeneous lattice-Dirac equation is solved. Its Fortran source code is 2200 lines long.

(b) **Equake**

EQUAKE is an earthquake modelling program. It simulates the propagation of elastic seismic waves in large, heterogeneous valleys in order to recover the time history of the ground motion everywhere in the valley due to a specific seismic event. It uses a finite element method on an unstructured mesh [15]. Its C source code is 1500 lines long.

(c) **SWIM**

SWIM is a weather prediction model, which solves the shallow water equations using a finite difference method. Its Fortran source code is 400 lines long.

As mentioned in 4.1, the behavior of data-level parallelism applications is highly dependent on the section of code to be executed.

## 2. Parsec

Parsec provides a wide variety of applications. In the recent years, the large advancement in silicon technology has let many processing cores integrated on a single die, each with access to sizeable shared caches, drastically reducing the latency of inter-core communication. This important change has been taken into account during the design of the algorithms used in Parsec.

(a) **dedup**

Dedup is a kernel which uses a next-generation data compression method called deduplication. It combines local and global compression to achieve very high compression ratios. This workload was included in the PARSEC benchmark suite because deduplication is becoming a standard method for backup storage systems and bandwidth optimized network appliances.

(b) **ferret**

This application is based on the Ferret toolkit which is used for content-based similarity search. It represents emerging next-generation search engines for non-text document data types and is parallelized using the pipeline model.

(c) **x264**

X264 is a lossy video encoder based on the ITU-T H.264 standard. H.264 improves over previous video encoding standards with many new features that allow it to achieve a higher output quality at the expense of a significantly increased compression time. Next-generation Blue-ray video players already use H.264 video compression, but many other application areas are equally supported by the H.264 standard.

As mentioned in section 4.2, interaction between threads plays an important role in task-level parallelism applications.

## 6.2 Metrics for Evaluating Phase Classification

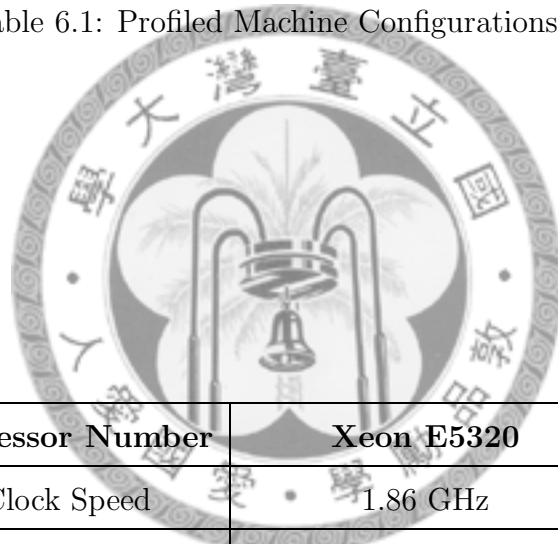
The metrics we examined is IPC (Instructions Per Cycle) which is the key metrics to help us understand the performance of multi-threaded applications. And we estimate the effectiveness of our phase clustering by inspecting the phase-based standard deviation of IPC, which stands for the similarity of examined metrics within each phase.

After the program's intervals are classified into phases on the profiled machine, we weight each simulation point according to the relative size the phase represents from the complete execution. Then on the evaluation machine, we get the phase-based IPC by combining each simulation point's IPC with corresponding weight. In addition, we get the phase-based standard deviation by combining each phase's standard deviation with corresponding weight. The phase-based IPC and standard deviation are compared with the IPC and standard deviation computed from the entire program execution. Better phase classification will result in lower per-phase standard deviation since the intervals within the same phase exhibit homogeneous behavior. For example, the phase-based standard deviation will be zero when all the intervals classified as the same phase perform exactly the same IPC.

Table 6.1 describes the specification of the machine profiled to get a set of simulation points for each target application. And Table 6.2 describes the specification of the machine, on which we evaluate our methodology.

<b>Processor Number</b>	<b>i7 920</b>
Clock Speed	2.66 GHz
L2 cache	256KB(private per core)
L3 cache	8MB(shared by all cores)
Intel QPI Speed	4.8 GT/S
Instruction Set	SSE 4.2

Table 6.1: Profiled Machine Configurations



<b>Processor Number</b>	<b>Xeon E5320</b>
Clock Speed	1.86 GHz
L2 cache	4MB(shared by two cores)
L3 cache	none
FSB Speed	1066 MHz
Instruction Set	64-bit

Table 6.2: Evaluation Machine Configurations

# Chapter 7

## Experimental Results

### 7.1 Proposed Scheme

We first compare the performance of the three frequency vectors (SBBV, CCV and SBBV+CCV) introduced in this work, in order to identify the effects of these techniques. Figure 7.1 shows the average IPC error rate of three approaches when compared to full application execution (i.e., running the applications to completion). In figure 7.1, first one is the error rate of phases detected by Sampled Basic Block profiling, we denote it as *SBBV* in the figure; second one is the error rate of phases detected by the vectors which record the number of coherence messages, we denote it as *CCV*; the last one is the error rate of phases detected by the combined vector which includes both of the above two kinds of profiling information and is denoted as *SBBV + CCV*. Figure 7.2 shows the phase-based standard deviation of the three approaches (SBBV,CCV and SBBV+CCV) compared with standard deviation of the program’s entire execution denoted as *RawData*. A small value in standard deviation means that the phase analysis succeeds in

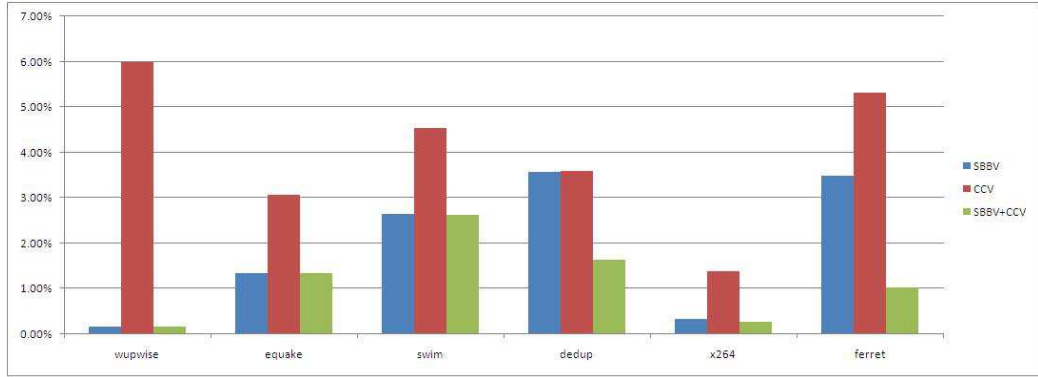


Figure 7.1: Error Rate of SBBV, CCV, SBBV+CCV

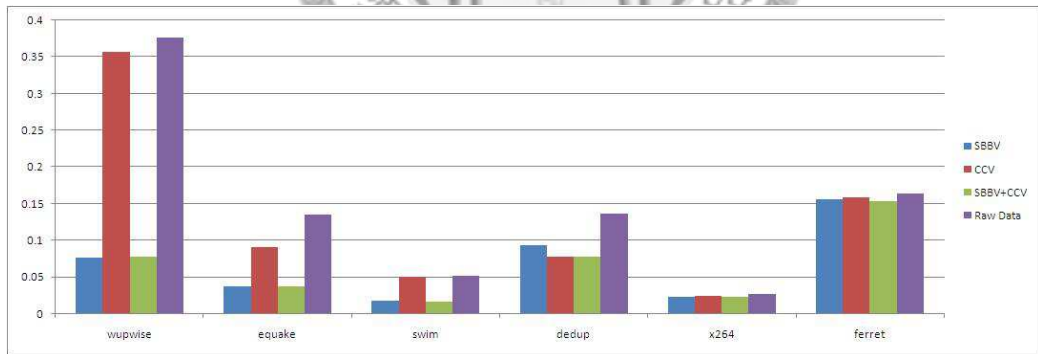


Figure 7.2: Phase-Based Standard Deviation of SBBV, CCV, SBBV+CCV



breaking varying program behavior into homogeneous phases.

SBBV technique shows good performance (1.38% error rate) in data-level parallelism applications. This is mainly because the interaction between threads in wupwise, equake and swim is very low. Hence, the behavior of data-level parallelism applications is highly dependent on the section of code be executed.

CCV technique itself can't capture phase behavior. The IPC error rate of CCV is high. However, SBBV with CCV is important for task level parallelism applications. SBBV+CCV technique achieves 47.43% more reduction in error rate compared to SBBV technique in task level parallelism applications. This is mainly because PARSEC workloads use a significant amount of communication. Hence, interaction between threads plays an important role in task-level parallelism applications. Contrary, since the communication between threads is little in data level parallel applications, the executed code takes a dominant position on these kind of applications' behavior. There are limited chances for SBBV+CCV technique to improve from SBBV technique. For x264, SBBV already shows good performance. This is because half of the spawned threads exhibit little communication. Hence, utilizing SBBV can capture most of the phase behavior.

## 7.2 Comparison with Previous Work

### 7.2.1 Baseline

In [31], Yu Zhang et al. claims that in the many-cores era application execution is not dominant by the instruction only. But instead the communication

structure of the application is as important as the instruction behavior. So Yu Zhang et al. proposed techniques to detect parallel program’s phases. They first collect information about the number of packets generated by each core during each interval to construct a frequency vector called Traffic Count Vector (TCV) on Network-On-Chip architecture. And they also utilized a common technique called Instruction Count Vector (ICV), which counts the number of instructions executed by each core during each interval. Besides, they combined the two vectors into a single vector to proposed a hybrid scheme.

We regard [31] as baseline to compare its performance with ours. For implementation of TCV, we use the number of off-core requests in place of number of routing packets, which is an architecture-dependent metric only available on NOC (Network On Chip) architecture. Because both the number of routing packets in [31] and the number of off-core requests on our experimental platform represent for the traffic from private cache to shared last level cache, it is the closet way to implement TCV on our target platform.

### 7.2.2 SBBV+CCV VS. ICV+TCV

In this section, we present the comparison between our proposed scheme (SBBV+CCV) with the hybrid technique used in [31]. Figure 7.3 shows the IPC error rate of ICV+TCV and SBBV+CCV techniques. Figure 7.4 shows the standard deviation of ICV+TCV and SBBV+CCV techniques. Our combination (SBBV+CCV) scheme provides better performace than the ICV+TCV technique. On average, the IPC error rate can be reduce by 62.60% and phase-based standard deviation can achieve 31.65% more

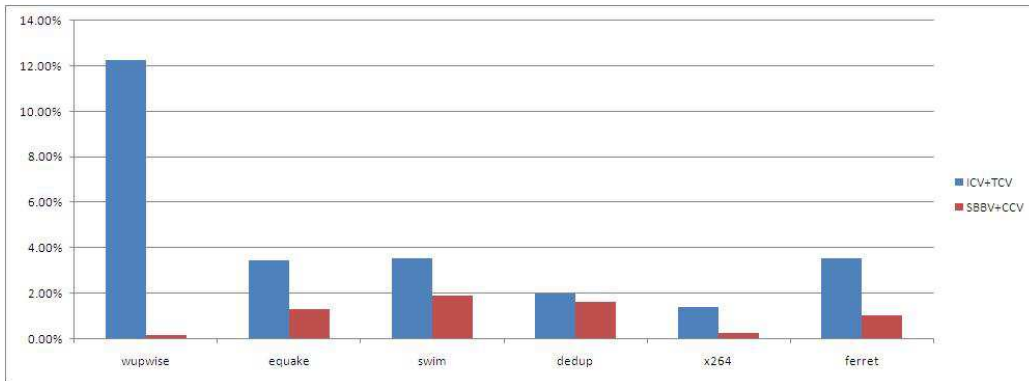


Figure 7.3: Error Rate of SBBV+CCV, ICV+TCV

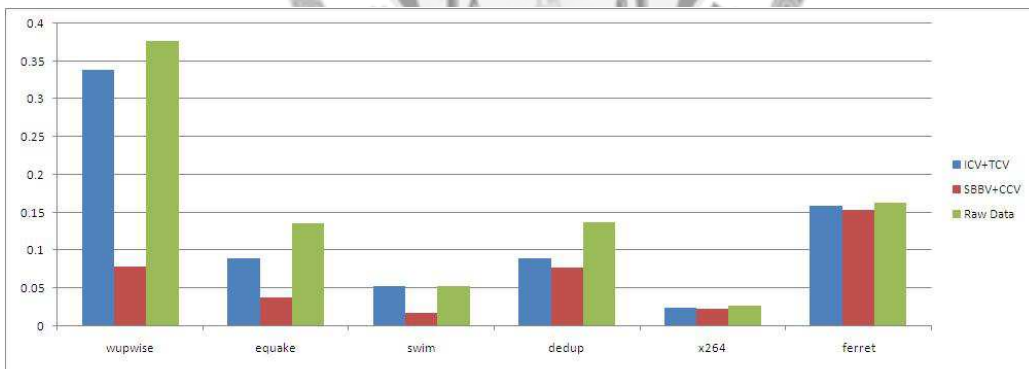


Figure 7.4: Phase-based Standard Deviation of SBBV+CCV, ICV+TCV

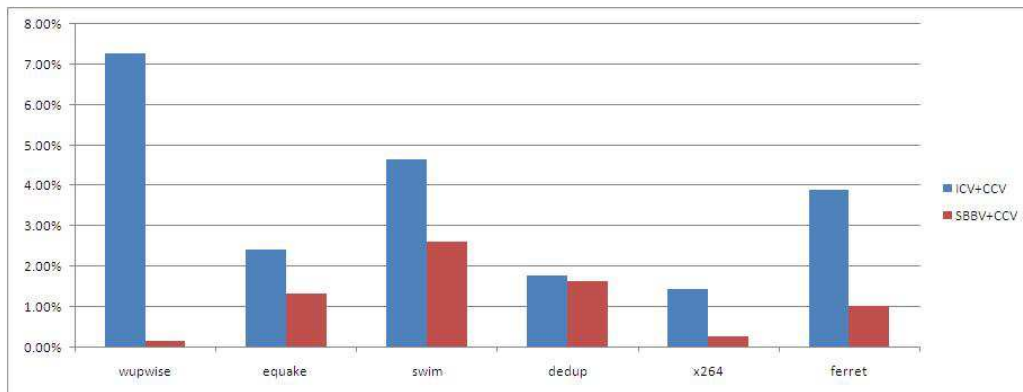


Figure 7.5: Error Rate of SBBV+CCV, ICV+CCV

reduction. For further analysis, we will replace the SBBV by ICV technique or replace CCV by TCV technique in our hybrid scheme. The next two sections would present how do these vectors affect the phase detection.

### 7.2.3 SBBV+CCV VS. ICV+CCV

We compare the accuracy achieved by ICV+CCV technique and BBV+CCV technique. The major difference between these two approaches is way to utilize the correlation between code signature and program behavior. In [31], Yu Zhang et al. formed the ICV by collecting the number of instructions executed within each interval. This is a much simpler and more straightforward than traditional BBV technique which becomes complicated as the number of core scales up. However, we think there is still room for improvement. In order to obtain the more detailed information about code signature without too much overhead, we construct the BBV in a sample manner. As Figure 7.5, we could see that the IPC error rate of SBBV+CCV technique is 58.02% lower than ICV+CCV technique on average. This is because SBBV

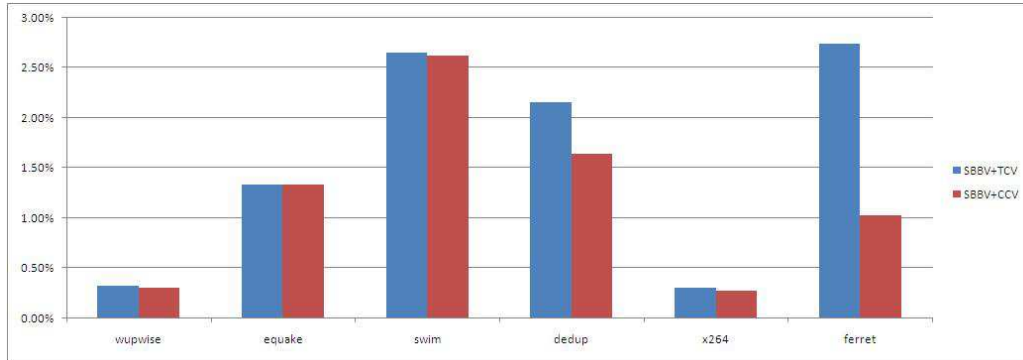


Figure 7.6: Error Rate of SBBV+CCV, SBBV+TCV

technique contains more precise information about what section of code has been executed than ICV technique. For dedup, CCV already captures lots of the phase information. Hence, the IPC error rate reduction of SBBV+CCV compared with ICV+CCV is not as much as other applications.

#### 7.2.4 SBBV+CCV VS. SBBV+TCV

In this section, we demonstrate the results of phase classifications with the aid of different frequency vectors which are related to interactions between threads. TCV (Traffic Count Vector) is established by recording the number of packets going through each router. And CCV technique contains the information about data sharing. Figure 7.6 shows the IPC error rate of SBBV+CCV and SBBV+TCV techniques. As we can see, the SBBV+CCV technique has much more reduction in IPC error rate than the SBBV+TCV technique for dedup and ferret which exhibit active communications between threads. This result indicates that CCV technique captures the pattern of interaction between threads more successfully than TCV.

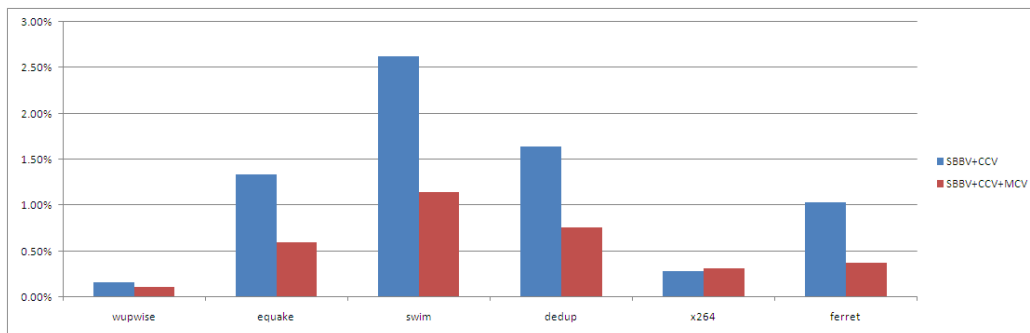


Figure 7.7: Error Rate of SBBV+CCV, SBBV+CCV+MCV

Application	wupwise	equake	swim	dedup	x264	ferret
Working Set	Up to 1.6GB	Up to 1.6GB	Up to 1.6GB	256MB	8MB	64MB

Table 7.1: Working Set of Applications

### 7.3 Memory Contention Count Vector

In this section, we show the effectiveness with the information of MCV which is aware of the situation of shared resource contention. Table 7.1 shows the working set of all the applications and figure 7.7 shows the IPC error rate of SBBV+CCV and SBBV+CCV+MCV techniques. As we can see, if application's working set is larger than the size of last level cache (8MB), there would be some performance gain for SBBV+CCV+MCV technique compared to SBBV+CCV technique. Because when application's working set is larger than the size of last level cache, threads would compete for the shared resource and the behavior of program would be affected. Hence, we utilize MCV to capture this kind of phase behavior. For example, all the applications except x264 have working set larger than last level cache, so using MCV can capture the situation of shared resource contention between threads. Contrary, because the working set of x264 does not exceed the size

of last level cache, the phase detection can not benefit from the information about MCV.



# Chapter 8

## Conclusion

In this thesis, we first identify the important issues for parallel programs phase detection. These issues include timing alignment, data sharing patterns and resource contention.

Then, we propose a methodology which takes above three issues into consideration to detect parallel application phases. We utilize global instruction count for timing alignment, CCV (Communication Count Vector) to collect the information about interactions between threads and MCV (Memory Contention Count Vector) to be aware of the situation of share resource contention between threads. In addition, we combine CCV and MCV with Sampled Basic Block (SBBV) into a single vector to develop a combination scheme.

We evaluate that the parallel phase analysis can be utilized to guide parallel program simulation by only considering the carefully chosen simulation points. The experimental results show that the IPC error rate is below 2% and we achieve 62.60% more reduction in IPC error rate compared with the technique used in [31].



# Bibliography

- [1] <http://parsec.cs.princeton.edu/>.
- [2] <http://perfmon2.sourceforge.net/>.
- [3] <http://www.spec.org/omp/>.
- [4] M. Annavaram, R. Rakvic, M. Polito, J.-Y. Bouguet, R. A. Hankins, and B. Davies. The fuzzy correlation between code and performance predictability. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–104, Washington, DC, USA, 2004. IEEE Computer Society.
- [5] V. Aslot. Performance characterization of the speccomp benchmarks.
- [6] M. P. B. Davies, J.Y. Bouguet and M. Annavaram. ipart: An automated phase detection and recognition tool. Technical report, 2003.
- [7] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 245–257, New York, NY, USA, 2000. ACM.

- [8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, New York, NY, USA, 2008. ACM.
- [9] C.-B. Cho and T. Li. Complexity-based program phase analysis and classification. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 105–113, New York, NY, USA, 2006. ACM.
- [10] A. Dhodapkar and J. E. Smith. Dynamic microarchitecture adaptation via codesigned virtual machines. 2002.
- [11] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 233–244, Washington, DC, USA, 2002. IEEE Computer Society.
- [12] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 217, Washington, DC, USA, 2003. IEEE Computer Society.
- [13] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 220, Washington, DC, USA, 2003. IEEE Computer Society.

- [14] e. a. Hamerly, G. Simpoint 3.0: Faster and more flexible program phase analysis. 2005.
- [15] O. G. L. F. K. D. R. O. J. R. S. Hesheng Bao, Jacobo Bielak and J. Xu. Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers.
- [16] C. Isci and M. Martonosi. Identifying program power phase behavior using power vectors. In *In Workshop on Workload Characterization*, 2003.
- [17] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. 2003.
- [18] C. Isci and M. Martonosi. Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques. 2006.
- [19] S. S. J. Lau and B. Calder. Transition phase classification and prediction. In *In 11th International Symposium on High Performance Computer Architecture*, pages 278–289. IEEE Computer Society, 2005.
- [20] e. a. Lau, J. Motivation for variable length intervals and hierarchical phase behavior. In *In IEEE International Symposium on Performance Analysis of Systems and Software*, 2005.
- [21] V. R. M. Hind and P. Sweeney. Phase shift detection: A problem classification. 2003.
- [22] J. MacQueen. Some methods for classification and analysis of multivariate observations. pages 281 – 297, 1967.

- [23] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel®itanium®programs with dynamic instrumentation. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, Washington, DC, USA, 2004. IEEE Computer Society.
- [24] J. M. Pelleg and A. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. pages 727 – 734, 2000.
- [25] e. a. Perelman, E. Detecting phases in parallel applications on shared memory architectures. In *In International Parallel and Distributed Processing Symposium*, pages 25–29, 2006.
- [26] T. Sherwood and B. Calder. Time varying behavior of programs. Technical report, 1999.
- [27] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.
- [28] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, New York, NY, USA, 2002. ACM.

- [29] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 336–349, New York, NY, USA, 2003. ACM.
- [30] F. Vandeputte and L. Eeckhout. Phase complexity surfaces: Characterizing time-varying program behavior. 2008.
- [31] G. M. J. K. Yu Zhang, Berkin Ozisikyilmaz and A. Choudhary. Analyzing the impact of on-chip network traffic on program phases for cmps. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 218–226, 2009.

