EFFICIENT SIMULATION TECHNIQUES FOR LARGE-SCALE APPLICATIONS

A Dissertation
Presented to
The Academic Faculty

By

Jen-Cheng Huang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
in
Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
August 2015

EFFICIENT SIMULATION TECHNIQUES FOR LARGE-SCALE APPLICATIONS

Approved by:

Dr. Hyesoon Kim, Advisor
Associate Professor,
School of Computer Science
Georgia Institute of Technology

Dr. Hsien-Hsin S. Lee
Adjunct Professor,
School of ECE
Georgia Institute of Technology

Dr. Moinuddin K. Qureshi
Associate Professor,
School of ECE
Georgia Institute of Technology

Dr. Karsten Schwan
Professor,
School of Computer Science
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
Professor,
School of ECE
Georgia Institute of Technology

Date Approved: June 2015

Dedicated to my parents

ACKNOWLEDGMENTS

With the support and help from my mentors and friends, I finally finished my Ph.D. journey. I would like to take this opportunity to thank them.

First of all, I thank my advisor, Prof. Hyesoon Kim. In the summer of 2012, she provided guidance me when I was desperately looking for help. Over these years, she has provided valuable advice on all aspects of a research project: identifying a research problem, forming my ideas, writing a conference paper, making presentation slides, and so on. Her advice was always sharp and directly pointed out the deficiencies I needed to improve. She kept helping me improve my research skills. Honestly, I could not have finished my dissertation without her advice. Her attitude at work always motivates me. To her, she is very persistent, focused, and hard working. To me, she is very caring. She gave me the freedom to work on the research projects that I am interested in. Whenever I was in trouble, she is generous with her help. All the things that I learned from her will continue to influence me for the rest of my life.

I thank my former advisor Prof. Hsien-Hsin S. Lee for introducing me to the computer architecture field. I am grateful that he guided me through the early years of my graduate study. I would like to thank my mentors from HP Labs: Matteo Monchiero and Yoshio Turner. Even after my internship, they still spent time talking with me every week to help me finish the project. Without their guidance, I would never have been able to finish my first research project. I thank my mentor Ching-Tsun Chou from Intel Labs. He has provided a lot of technical help so that I could finish my intern project on time. I would also like to thank all my committee members: Prof. Hyesoon Kim, Prof. Hsien-Hsin S. Lee, Prof. Moinuddin K. Qureshi , Prof. Sudhakar Yalamanchili, and Prof. Karsten Schwan for serving on my committee and providing me valuable comments to improve my dissertation.

I thank my colleagues from HPArch: Nagesh Lakshminarayana, Sunpyo Hong, Jaekyu Lee, Pranith Kumar, Jaewoong Sim, Dilan Manatunga, Joo Hwan Lee, Hyojong Kim, Nimit

Nigania, Prasun Gera, Lifeng Nai, and Ramyad Hadidi for their feedback on improving my papers and the discussions during the weekly meetings. Lifeng Nai, Joo Hwan Lee, and Hyojong Kim, coauthored the papers with me and helped me develop research ideas. Dilan Manatunga helped me improve the writing quality of my papers and introduced me to the world of anime. Nagesh Lakshminarayana, Prasun Gera, and Pranith Kumar provided me much technical support on simulation and system issues. Jaewoong Sim challenged me with insightful feedback to improve my projects. I thank my colleagues from MARS: Sungkap Yeo, Mohammad Hossain, Tzu-Wei Lin, Guanhao Shen, Dong Hyuk Woo, Dean Lewis, Eric Fontaine, Manoj Athreya, Andrei Bersatti, Nak Hee Seong. I thank Sungkap Yeo and Mohammad Hossain for letting me co-author their work. Sungkap and I joined the lab at the same time. I am grateful for his encouragement over the years. I thank Dong Hyuk Woo gave me research suggestions and a valuable opportunity to intern in the Intel lab.

I thank all my friends who have helped me over these years. You-Chi Cheng, Hwa-You Hsu and Chieh-Yu Lee are my closest friends in Atlanta. I will not forget the happy times we had hiking and grocery shopping. I would like to thank Chi-Ti Hsieh, Ping-Chang Shih, Yu-Hsien Hsu and Cheng-Lin Tsao who helped me resolve many issues during my early years in the graduate school. I thank Yuejian Xie, Tzu-Han Hung and Chun-Ming Chen for providing numerous suggestions on research and life issues. I thank Xinwei Chen and He Xiao for spending many coffee breaks with me. I thank Tricia Grindel for helping me edit my papers to improve the readability.

Finally, I would like to thank my parents: Chiung-Tang Huang and Hsiu-Min Weng. They have never given me any pressure to finish the degree and always let me choose the path that I wanted. My younger brother, Yu-Cheng Huang, has shared many life tips in the U.S. with me. My girlfriend, Hsiao-Ling Lin, has been supporting me and gone through many ups and downs with me. They always believe in me and give me unconditional love and support for which I am very grateful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Architectural simulation is one of the most important performance modeling approaches for architecture design. Compared to the native execution speed, the simulation speed has 10,000x times slowdown. Therefore, the techniques that reduce the simulation time are required. Prior techniques target for the conventional benchmark suites, which are either single-threaded or having limited parallelism. However, the emerging workloads, such as GPGPU kernels and data analytic workloads, feature massive parallelism and huge data volume. Prior simulation techniques are not applicable to these workloads or could still incur long simulation time. In this dissertation, three simulation techniques were proposed to simulate these large-scale applications accurately and efficiently.

First, this dissertation presents the sampling techniques, inter-launch and intra-launch sampling, for reducing the simulation time of GPGPU kernels. In inter-launch sampling, a new feature vector is proposed to capture the control and memory divergence, which are the unique characteristics of GPGPU kernels. In intra-launch sampling, the dynamic sampling approach is proposed to sample the instruction intervals that are non-deterministic due to warp scheduling effect.

Second, even with sampling, the number of simulated instructions is proportional to the number of threads in the system. The longer simulation time is expected with more hardware threads integrated onto the GPU. To further reduce the simulation time, an abstracted simulation technique were proposed. Instead of using detailed timing simulation, the simulation model is abstracted using functional simulation and analytical modeling. This technique models the performance of a multithreaded GPU core using the execution behaviors of a single warp. It achieves low modeling error of performance compared to a detailed timing simulation while achieving two orders of magnitude speedup.

Finally, to reduce the simulation time of data analytic workloads, a sampling framework were proposed to select the simulation points. Prior sampling approaches focused on

capturing the performance difference caused by code changes. However, because of the data diversity of the data analytic workloads, the sampling approach needs to take the data impact on performance into account. The framework profiles the workloads with negligible overhead on a real machine for fast analysis of phase behaviors. It uses stratified random sampling, a statistical sampling technique, to account not only the performance impact of code, but also other factors, such as data access patterns. Furthermore, it provides the input sensitivity test to further reduce the simulation time when simulating multiple inputs.

CHAPTER 1

INTRODUCTION

1.1 The Problem: Reducing the simulation time of large-scale applications

Architecture simulation is an important performance modeling technique. Modeling the hardware components with sufficient detail helps developers to identify both hardware and software bottlenecks. It is more flexible than hardware counters because it can be modified to provide some program statistics that hardware counters may not provide, such as instruction-level parallelism (ILP). In addition, simulation is a required tool for designing future processors before they are implemented in the hardware.

However, the major issue of architectural simulation is the huge simulation time. Since architecture simulation uses software to model hardware behavior, its speed is several order of magnitude slower than real hardware. For example, the typical speed of an architecture simulator is from 10,000 instructions per second up to several million instructions per second while the slowdown compared to the actual execution is around 10,000x. Thus, it is important to reduce the simulation time in order to make simulation a feasible solution for performance study.

The speed gap between simulation and real hardware is growing wider and wider for the following reasons: (1) hardware complexity, (2) massive parallelism and (3) massive data.

In terms of hardware complexity, the future processor will become more complex and accommodate more hardware components to improve performance. For example, the network-on-chip (NoC) design will be more complex to reduce the communication overhead between cores. Memory hierarchy will become deeper to avoid off-chip memory accesses. Since more complex hardware designs require more lines-of-code to model their behaviors, the cost of simulating one instruction becomes higher.

In terms of massive parallelism, the technology trend is to integrate more cores into a chip to increase performance. Furthermore, the new computing platform, GPGPU, executes thousands of threads concurrently using a large number of simple cores. To match the hardware parallelism, a straightforward approach is to parallelize the simulator. However, the speedup is still limited since memory system is shared creating a single point of bottleneck. To accurately model the memory contention between threads, synchronization is required to ensure the correctness of their access orders. In addition, periodic synchronization, e.g., every 1000 cycles, is required to make sure the threads progress at the same speed. The synchronization overhead negates the benefits of parallelization, especially when the number of cores is high.

In terms of massive data, the data analytic workloads, which are the services deployed in data centers, process a huge amount of data on multiple nodes. For such workloads, more extensive resources are required to simulate because of the involvement of multi-node simulation and the large input set. Another challenge of simulating the data analytic workloads is to deal with the managed runtime, e.g., Java, applications, since the workloads are usually written in high-level languages to achieve platform independence.

Prior studies have proposed various techniques to reduce the simulation time, such as sampling [3, 1] and simulation abstraction [4]. However, they all focus on traditional applications or hardware without accounting for the existence of higher parallelisms in the modern hardware and software. From a hardware perspective, they target platforms with a small number of out-of-order cores (up to 8 cores [5, 6]), that are very different from some modern platforms, e.g., GPGPU, featuring a large number of simpler in-order multithreaded cores. From a software perspective, the emerging workloads, GPGPU kernels and the data analytic workloads, have very different characteristics from the conventional benchmarks, e.g., SPEC [7] and PARSEC [8]. The conventional benchmarks are purely

CPU-bound, native executed without managed runtimes and have limited thread-level parallelism. In this dissertation, several mechanisms have been innovated to efficiently simulate the GPGPU kernels and data analytic workloads, which have massive thread and data parallelism, with small modeling errors.

1.2 The Solution: Domain-specific simulation mechanisms

The solution to the emerging large scale workloads is to design domain-specific simulation techniques. Based on the characteristics of the applications and the execution model of the platform, we design a set of simulation-based modeling techniques for GPGPU kernels and data analytic workloads. They are based on mathematical theory to provide trustworthy simulation results. The highlights of the proposed techniques are as follows.

1.2.1 Sampling technique for GPGPU kernels

To reduce the simulation time of GPGPU simulation, we have developed TBPoint, the sampling technique that can deal with the massive thread-level parallelism (TLP). Simulating the GPGPU kernels may incur a huge overhead since GPUs typically have a large number of threads and high throughput. However, since the prior sampling techniques do not account for massive parallelism, the sample size is likely to be more than required.

The proposed sampling technique, TBPoint, contains inter-launch sampling and intra-launch sampling. Inter-launch sampling selects the representative kernel launches to be simulated while the performances of the remaining kernel launches are predicted as the same as the representative ones. Then, intra-launch sampling selects and simulates the instructions within a kernel launch. The proposed sampling technique can quickly adapt to different levels of thread-level parallelism incurring little overhead.

1.2.2 Hybrid modeling technique for GPGPU kernels

Even though the simulation time can be reduced through sampling, the number of simulated instructions of each simulation point may be high depending on the number of threads in

the system. By contrast, to quickly model the performance, analytical modeling is likely to have higher errors.

The proposed technique, GPUMech, leverages both functional simulation and analytical modeling to balance the accuracy and the modeling speed. Based on the concept of interval analysis [9, 10], an analytical model of multithreaded core is developed to model the performance while the cache simulation is used to identify the stall events of a thread. The proposed technique balances the speed and accuracy. In addition, it provides the CPI stack for analyzing the performance bottlenecks in a GPGPU kernel.

1.2.3 Sampling technique for data analytic workloads

Because of the era of big data, the data analytic workloads, which have the computations scale out to multiple nodes, have growing importance. Even though many frameworks have been developed for those workloads, it is still a challenge in the architecture community to simulate those workloads. Although some sampling tools exist, they are inadequate for those workloads that are implemented using high-level languages and that have long running time.

The proposed technique, SimProf, is a profiling tool for analyzing the phase behaviors and selecting the representative simulation points. It profiles the Java applications using Java call stacks and the hardware counter values. To select the simulation points, SimProf uses the stratified random sampling approach to deal with the non-homogeneous phase behaviors.

1.3 Thesis Statement

The modern software and hardware that feature massive parallelism can be simulated efficiently without sacrificing much accuracy by designing mechanisms based on their execution characteristics.

1.4 Organization of This Thesis

Chapter 2 provides the related work of different categories of simulation techniques. Chapter 3 presents the sampling technique for GPGPU kernels. Chapter 4 presents the hybrid modeling technique that combines functional simulation and analytical modeling for GPGPU kernels. Chapter 5 presents a sampling framework for data analytic workloads based on modern computing platform, e.g., Hadoop and Spark. Chapter 6 concludes this document.

CHAPTER 2

RELATED WORK

Multiple techniques have been proposed to reduce simulation time including simulation sampling, statistical simulation and parallel simulation, each of which is explained as follows.

2.1 Simulation Sampling

Simulation sampling, one of the most popular techniques to reduce the simulation time, selects the sampling units to be simulated in detail while "fast-forwarding" the other sampling units. During fast-forwarding, the detailed timing simulation is not performed, but the functional simulation is still required to guarantee the functional correctness. The sampling unit size is a variable depending on the sampling technique. Sampling techniques can be classified into statistical sampling and profile-driven sampling described as follows.

2.1.1 Statistical Sampling

Statistical sampling takes samples based on statistical theories. Two sampling techniques have been proposed for statistical sampling: random sampling and systematic sampling (periodic sampling). Conte et al. [11] proposed random sampling which selects the sampling units at random. This approach provides the confidence interval and error bounds, but it is based on unrealistic assumptions about the micro-architecture, e.g., perfect cache hierarchy or branch prediction. Wunderlich et al. [3] proposed SMARTS, which chooses systematic sampling over random sampling to take samples since it is more intuitive to be implemented in an event-driven simulator. In addition, SMARTS uses the z-score of the central limit theorem to estimate the required sample size to achieve the specified confidence interval without assuming the target population. SMARTS uses a small sampling unit size, 1000 instructions, to minimize the instructions simulated in detail. For such a

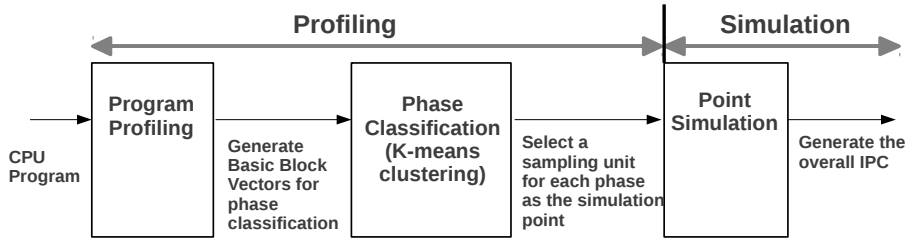


Figure 1: Simpoint overview [1]

small sampling unit size, the requirement of the reconstruction of cache and branch predictor states is more important compared to a large sampling unit size. Thus, SMARTS applies "functional warming," which continuously warms the components with long history to guarantee correct micro-architecture states at all times. The major bottleneck of SMARTS is the speed of functional warming since the authors found that the required instructions for detailed simulation are relatively small (< 1%).

Carlson et al. [6] and Ardestani et al. [5] extend SMARTS to support multithreaded applications on multi-core platforms. The challenge of sampling multithreaded applications is to decide the relative speed of each thread during fast-forwarding. The IPC of a thread in the previous period is used as the IPC of the fast-forwarding period of the thread. In addition, the authors adopt systematic sampling but take samples based on cycles instead of instructions and claim better accuracy than instruction-based sampling like SMARTS.

2.1.2 Profile-driven Sampling

Profile-driven sampling takes samples based on the profiled information. The idea is to pick a sampling unit from each program phase as the simulation point. Since other sampling units in the same phase are predicted to have the same behavior as the selected point, the overall IPC can be predicted. The profiling is used to cluster sampling units into phases. Thus, the key to good accuracy is to extract the features of each sampling unit that can accurately cluster the sampling units with similar behaviors into phases. Sherwood et al. [1] developed Simpoint, the most popular profile-driven sampling technique. Because they found that the instructions being executed have a strong correlation with performance [12],

Basic Block Vector (BBV) is used as the feature vector of each sampling unit. Equation 5 shows the representation of a BBV. BBV characterizes the basic blocks executed in a sampling unit and their execution counts. By clustering the BBVs using k-means, a clustering algorithm, the sampling units are grouped into clusters, while each cluster is a phase. For each cluster, a sampling unit closest to the centroid of the cluster is selected as the simulation point of the phase. Equation 3 shows the overall CPI. The weight of a phase is determined by the number of sampling units in the phase, as shown in Equation 2.

$$BBV =< BB1 : \frac{\#BB1\_insts}{\#insts}, BB2 : \frac{\#BB2\_insts}{\#insts} ... BBN : \frac{\#BBN\_insts}{\#insts} > \quad (1)$$

$$phase\_weight_i = \frac{\#sampling\_units_i}{total\_sampling\_units} \quad (2)$$

$$Total\_CPI = \sum_{i \in phases} (simulation\_point\_CPI_i \times phase\_weight_i) \quad (3)$$

Pinpoint [13] leverages the Pin tool [14], a dynamic instrumentation tool, to collect BBVs. Sherwood et al. proposed a processor architecture based on the concept of BBV with online phase detection. Perelman et al. [15] extended Simpoint to provide statistically valid simulation points with confidence interval analysis. Biesbrouck et al. [16] proposed co-phase matrix, which detects phases when multi-programmed workloads are running. Lau et al. [17] extended Simpoint to support variable-length sampling units so that the sampling unit size is better matched with the period of a phase. In addition to BBV, Eeckhout et al. [18] consider several micro-architecture independent metrics to find simulation points. They also explored picking simulation points cross-input and cross-program to further reduce the number of simulation points. Similarly, Lau et al. [19] use program semantics, called graph and loop graph, to identify the phase changes in the software level. For multi-threaded benchmarks with barriers, Carlson et al. [20] consider each inter-barrier region as a sampling unit and use BBV to pick the inter-barrier regions needed to be simulated.

SMARTS and Simpoint can be compared based on (1) implementation easiness, (2) accuracy and (3) efficiency. From an implementation easiness standpoint, the simulation

points selected by Simpoint are easier to be used by simulators since no warming techniques need to be implemented while SMARTS requires functional warming integrated into simulators and statistical analysis to be performed. From an accuracy perspective, Yi et al. [21] concluded that SMARTS is slightly more accurate than Simpoint. The reason is that Simpoint determines the similarity between sampling units using code signature (BBV) without considering micro-architecture states. On the other hand, SMARTS determines required sample size based on the central limit theorem using the actual CPI variation. Wunderlich et al. [22] applied stratified random sampling using BBVs and found that it does not provide much benefit over simple random sampling when the sampling unit size is small. From an efficiency perspective, SMARTS is slower than Simpoint due to the overhead of functional warming. In addition, SMARTS requires running the entire benchmark, while in Simpoint, the simulation can be terminated early once all simulation points have been simulated.

2.2 Statistical Simulation

Different from sampling, statistical simulation reduces the simulation time by generating and simulating synthetic traces rather than simulating the original programs. To generate synthetic traces, a set of program characteristics is profiled and represented as distributions. This process is called statistical profiling. Then, statistical simulation used a simple simulator to simulate the synthetic traces to get performance numbers. Because the length of a synthetic trace is much shorter than the original program, the simulation time can be greatly reduced.

Statistical profiling characterizes the program statistics into micro-architecture dependent and independent statistics. The micro-architecture independent statistics include instruction mix and dependency distance distribution. Statistical profiling also needs to consider control flow behaviors. Oskin et al. [23] propose using control flow graph tagged with transition probabilities. Eeckhout et al. [24] propose the statistical flow graph (SFG),

which uses a Markov chain to further improve the accuracy. Using SFG, the execution paths caused by different control flows are better captured. The micro-architecture dependent statistics include the locality events, such as cache and branch predictor behaviors. The initial designs of statistical simulation [25, 26] to capture the locality events do not consider the delayed hits and the memory level parallelism (MLP). Genbrugge et al. model the delayed hits by collecting the cache line reuse distributions and model MLP through the global history of cache events. Functionality-wise, the synthetic trace simulator is similar to the full blown simulator but it is less complex since it only needs to simulate the information encoded in the traces.

2.3 Parallel Simulation

Eeckhout [27] categorizes parallel simulation into (1) parallel sampled simulation and (2) parallel simulation, explained as follows.

Parallel sampled simulation divides a simulation into multiple time chunks. Simflex [28] speeds up SMARTS using checkpoints. According to Simflex, 99% of the overhead of SMARTS comes from functional warming. Thus, the authors proposed "live points" to store the micro-architecture states in addition to the functional states. Instead of checkpointing the whole micro-architecture states, only the states of cache hierarchy and branch predictor need to be stored since most of micro-architecture states can be reconstructed through a few thousand instructions of warming. The simulation can be parallelized by running different checkpoints in parallel. One limitation of this approach is that the stored micro-architecture states are not flexible and may not be used in different hardware configurations, e.g., different cache sizes. Using checkpoints to parallelize the simulation. Parallelizing simulations using checkpoints has been used in other work [29].

Parallel simulation is becoming popular for simulating multi-core platforms. It is natural to parallelize by cores because of the independence between core executions. The

intuitive way is to synchronize between cores every cycle. Although this approach incurs no errors, the synchronization overhead outweighs the benefits of parallelization. The cycle-by-cycle synchronization can be relaxed using quantum-based synchronization [30], which synchronizes between threads every quantum size, e.g. N cycles. If the quantum size is smaller than the latency of propagating an event from one thread to another, the accuracy is the same as the cycle-by-cycle synchronization. For multicore processors, since the communication latency between threads is small, the quantum size needs to be in the range of tens of cycles without losing accuracy [31]. Similarly, the small quantum size limits the benefit of parallelization. To address this limitation, some studies trade accuracy for speed by further relaxing the synchronization. Slacksim [32] uses the concept of "slack" defined as the cycle count difference between threads. The simulation can tolerate a certain amount of slack without barrier synchronization. Zsim [33] uses the "bound-weave" approach, which divides the synchronization into two steps. In the first step, the threads run in parallel to determine the interaction between threads on shared resources. In the second step, since the interaction between threads is known from the first step, the synchronization can incur much lower overhead. Zsim also uses Pin tool [34], a binary instrumentation tool, to speed up the functional simulation and applies an instruction-driven timing model, which moves the timing model to the instrumentation stage for low-overhead timing simulation.

2.4 Analytical Modeling

2.4.1 Analytical Models of Single-threaded Cores

The foundation of interval analysis was proposed by Karkhanis et al. [9] and Eyerman et al. [10]. The basic idea is that the performance of a processor is equal to the issue rate of a processor (a sustained performance) unless disruptive miss events occur such as branch mispredictions or cache misses. Performance is then estimated by subtracting the stall cycles due to different stall events from the maximum issue rate. Figure 2 illustrates interval analysis. An *interval* is defined as a sequence of instructions with the maximum issue rate followed by stall cycles. Functional simulators are used to detect stall events.

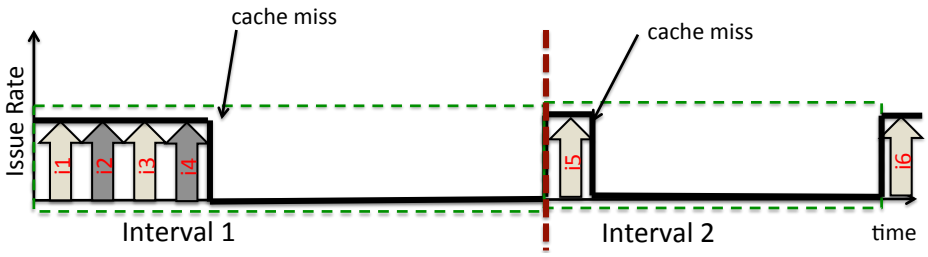


Figure 2: Example of interval analysis. (i: instruction.)

Several studies improved or applied the interval analysis technique. Genbrugge et al. [35]. proposed "interval simulation," which improves the simulation speed by abstracting the out-of-order execution using interval analysis. Eyerman et al. [36, 37] proposed performance counter architectures for out-of-order processors and SMT processors based on interval analysis. Chen et al. [38] improved the accuracy of the technique considering pending cache hits, prefetching, and MSHRs. Breughe et al. [39] applied the technique to in-order processors.

2.4.2 Analytical Models of Multi-threaded Cores

Chen and Aamodt proposed a first-order performance model of a multithreaded core [40]. They performed a Markov chain analysis to predict the performance of a multithreaded core similar to a GPU core except with vector processing. To begin with, a single thread is modeled as a random variable with two states: *activated* and *suspended*. Activated means that the thread can issue an instruction at the cycle. Otherwise, it is stalled. The transition probability from activated to suspended is p, while the probability from suspended to activated is $\frac{1}{M}$, where M is the number of cycles of a thread being suspended. By performing the Markov chain analysis, the probability of being in any state at any cycle can be known.

In the past few years, several GPU performance models have been proposed. Baghsorkhi et al. [41] proposed using a work flow graph (WFG), an extension of the control flow graph, to estimate the performance. Zhang and Owens [42] proposed a model to measure the execution time of the instruction pipeline, shared memory, and global memory respectively using micro-benchmarking. However, these prior works did not model the

cache hierarchy that is equipped in all modern GPUs.

GPUPerf [43] is a performance analysis framework used to predict the performance bottlenecks of GPGPU kernels. The authors use benefit metrics to indicate the relative importance of different performance bottlenecks: B_{fp}, B_{itilp}, B_{memlp}, and B_{serial}. They model the benefits of removing the inefficient computation, increasing TLP and MLP, and removing the synchronization and resource contention. B_{fp} represents the benefits of removing the inefficient computation; B_{itilp} represents the benefits of increasing the thread-level parallelism; B_{memlp} represents the benefits of increasing the memory-level parallelism; and B_{serial} represents the benefits of removing the synchronization and resource contention. The benefit metrics are generated by the performance model extended from the MWP-CWP model [44]. But, similar to the model for a multithreaded core, they do not model the queuing delay due to resource contention in the memory system nor do they model the scheduling policy. This affects not only the modeling accuracy, but also the reported benefit metrics. For example, the model may suggest increasing the memory-level parallelism (MLP), but it may instead hurt the performance due to increased queuing delays. In addition, the proposed benefit metrics do not show the performance bottlenecks in detail, e.g., micro-architecture level.

Eq. 4 shows that the total execution time is composed of T_{comp}, T_{mem} and $T_{overlap}$. T_{comp} is the execution time to excute compute instruction. First, the total exeuction cycles of a single warp is counted. Because the execution cycles from multiple warps can overlap, such effect can be modeled by estimating the TLP. Similarly, to compute T_{mem}, the overlapping effect can be modeled by estimating the memory-level parallelism (MLP). $T_{overlap}$ represents the degree of how much the memory access cost can be hidden by warped execution. $T_{overlap}$ is equal to $min(T_{comp}\times, T_{mem})$.

$$T_{exec} = T_{comp} + T_{mem} - T_{overlap} \tag{4}$$

In addition to performance models, several GPU cache models have been proposed.

Baghsorkhi et al. [45] applied Monte Carlo simulation for a finite number of times to mimic the non-deterministic schedule deviation between thread blocks. Tang et al. [46] applied the reuse distance theory on a single thread block to model the cache miss rate without considering MSHRs. Nugteren et al. [47] proposed a cache model for L1 cache based on the reuse distance theory. They emulate per-warp memory traces with the round-robin scheduling policy. In addition, they modeled MSHRs accounting for a limited number of outstanding misses.

2.5 Input Set Selection

Prior work on input sets focused on selecting the representative input sets. Eeckhout et al. [48] used principle component analysis (PCA) to select the representative program-input pairs. They characterize each program-input pair using 20 program characteristics, such as instruction mix and cache miss rates. Breughe et al. [49] used BBV to select the representative inputs for microprocessor design space exploration. Hsu et al. [50] found that procedure coverage and microarchitecture behaviors are different between the training and reference sets of SPEC 2000.

CHAPTER 3

TBPOINT: REDUCING SIMULATION TIME FOR LARGE-SCALE GPGPU KERNELS

This chapter first describes the problem of long simulation time of GPGPU kernels. To reduce the simulation time, this chapter introduces TBPoint [51], which is an infrastructure that reduces the instructions to be simulated by sampling GPGPU kernel launches and thread blocks. On average, the solution leads to 1.74% of sampling error compared to full simulation while simulating only 2.6% of total instructions.

3.1 Introduction

Recently, one of the main scientific computing paradigms in addition to Titan [52] and CSCS (Swiss National Super- computing Center) [53] is General-purpose graphics processing units(GPGPU), which has large-scale computing power. To take advantage of this computing power, a wider range of algorithms have been converted to GPGPU kernels. Future workloads of GPGPU kernels will be much more complex and much larger scale.

To design new architectures optimized for GPGPU kernels, computer architects use a cycle-level simulator to gain insights into kernel behaviors. Using cycle-level simulation can help not only architects but also application developers to understand the performance bottleneck in applications and architectures.

However, because GPGPU architectures have many cores to simulate, cycle-level simulation takes a significant amount of execution time. Table 1 shows comparisons of GPGPU execution times and expected simulation times of a cycle-level GPGPU simulator. The GPGPU execution times are from Burtscher et al. [54], and we used the Macsim simulator for the GPGPU simulations [55]. The slowdown of Macsim running on the Intel Ivy-bridge is around 80,000x compared to native NVIDIA Quadro processors. Even for kernels with

Table 1: Comparison of GPGPU execution time and Simulation Time. GPU time is for NVIDIA Quadro 6000.

Time	NB	SP	SSSP	PTA	TSP	DMR	MM
GPU (msec)	28557	18779	7067	4485	4456	3391	881
Simulation	3.78 weeks	2.48 weeks	6.54 days	4.15 days	4.13 days	3.14 days	19.58 hours

a few seconds of running time, the simulation time takes days. As a result of this slowdown, simulation is an unattractive approach. Although most GPGPU kernels have short kernel sizes, the overhead comes from simulating a massive number of threads. Because of the computing power of GPU processors, an 80,000x slowdown is reasonable. GPGPU applications can easily have 1GFLOPS or even higher performance. If a simulator can simulate 1,000FLOPS/sec (which is similar to 10K instructions per sec in typical cycle-level simulators), a 10^6 slowdown is typical. To accelerate simulation time, one can parallelize a cycle-level GPGPU simulator, but when large-scale GPGPU systems are simulated, the required resources for such simulations become significant.

In this chapter, we first introduce the GPGPU simulation model and the terminology. Then, we discuss the existing sampling techniques on reducing simulation time.

3.2 Background and Motivation

3.2.1 Target Platform and Terminology

Figure 3 illustrates GPGPU simulations and the following terminology used throughout the chapter.

- **Occupancy:** For a kernel, "system occupancy" refers to the maximum number of concurrent thread blocks, while "SM occupancy" refers to the maximum number of concurrent thread blocks on one SM. For example, Figure 3 shows that SM occupancy is 1 and system occupancy is 4 for both Kernels A and B.

- **Kernel Launch (Launch):** A "kernel launch" is an instance of a GPGPU kernel that can be executed multiple times. For each kernel launch, thread blocks are dispatched

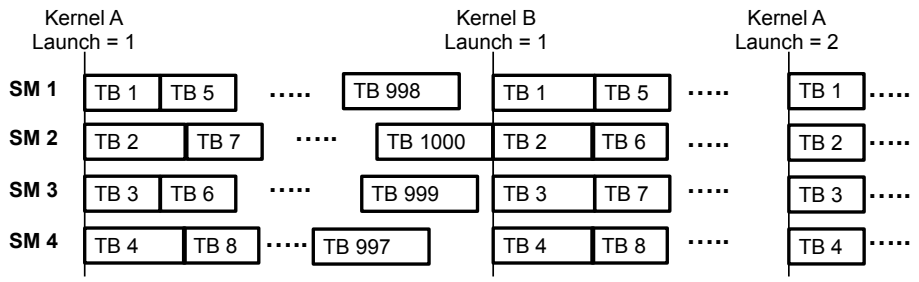


Figure 3: GPGPU Simulation Overview

in a sequence based on their thread block IDs. The global thread block scheduler dispatches the thread blocks in a greedy fashion. All thread blocks of the kernel launch need to be executed before the next kernel launch.

- **Warp/Thread Instruction:** "Warp instructions" or "instructions" are those executed by a warp, and "thread instructions" are those executed by a thread within a warp.
- **Interval:** An interval is a series of contiguously executed instructions.

3.2.2 Motivation

In profiling-based sampling, the idea is to profile a program and find the instruction intervals likely to have homogeneous behaviors based on the program characteristics. The simulation time can be reduced by simulating one interval to represent the performance of the other intervals with homogeneous behaviors.

The requirements of profiling-based sampling are as follows.

- **Hardware independence:** The profiling should have no constraints on the platform where the profiling can be performed.
- **One-time profiling:** The profiling needs to be executed only once for each program/input combination, and the results can be used in different hardware configurations.

- **Mathematical model support:** The sampling approach should be supported by detailed mathematical modeling to confirm its accuracy.

The most popular profiling-based sampling technique targeting single-threaded CPU applications is Simpoint [1], the functionality of which is described as follows.

- At profile time, the program is divided into sampling units that contain a fixed number of instructions, such as one million instructions. Then, the profiler collects a basic block vector (BBV) for each sampling unit. A dimension of the vector represents a basic block in the program. The value of a dimension is the executed instruction counts of the basic block normalized by the total instruction counts, as shown in Equation 5. The BBVs, used as feature vectors, are fed into the clustering algorithm, k-means, to group the sampling units into clusters. For each cluster, a sampling unit that is selected as the simulation point represents the performance of the other units in the cluster.

- At simulation time, only the sampling units selected as simulation points need to be simulated. The other sampling units can be skipped by fast-forwarding. Each simulation point could have different weights, depending on the number of sampling units in the cluster. The overall CPI can be predicted by Equation 5.

$$
\begin{aligned}
BBV =< BB1 : \frac{\#BB1\_insts}{\#insts}, BB2 : \frac{\#BB2\_insts}{\#insts} ... BBN : \frac{\#BBN\_insts}{\#insts} > \\
Total\_CPI = \sum_{i \in phases} (representative\_unit\_CPI_i \times phase\_weight_i) \\
phase\_weight_i = \frac{\#sampling\_units_i}{total\_sampling\_units}
\end{aligned}
\tag{5}
$$

Since GPGPU kernels can be considered as multithreaded applications, Pinpoint [14], an extension of Simpoint for sampling multi-threaded applications, could be applicable to GPGPU kernels. In Pinpoint, BBVs are collected on a per-thread basis by actually executing all threads on a real system. However, it does not meet the requirements of profiling-based sampling for the following reasons.

- The profiling needs to be redone for different hardware configurations since the profiling results (simulation points) can only be applied to the simulated platform, which has the same hardware configurations as the profiling platform.

- Although the BBV has a strong correlation with performance in single-thread applications [12], it is uncertain whether the same would be true for GPGPU kernels because of the warp scheduling effect.

Table 2 summarizes the comparisons of different profiling-based sampling techniques. To satisfy all of the requirements of profiling-based sampling, we propose a sampling approach, TBPoint. In terms of hardware independence, TBPoint uses GPUOcelot [56] as the profiling tool, which performs the functional simulations of GPGPU kernels and collects the information about each thread block. In terms of one-time profiling, for different hardware configurations, such as a different number of warps and SMs, TBPoint simply needs to re-perform clustering while reusing the profiling results, incurring low overhead. To model the warp scheduling effect, we use a Markov Chain model that accounts for the performance impact of the effect.

Table 2: Comparisons of profiling-based sampling techniques

	applicability to GPGPU	hardware independence	one-time profiling	math model support
Simpoint [1]	N	Y	Y	Y
Pinpoint [14]	Y	N	N	N
TBPoint	Y	Y	Y	Y

To reduce the GPGPU simulation time, inter-launch sampling and intra-launch sampling are proposed. To reduce the number of kernel launches, the former selects the kernel launches that need to be simulated. If different kernel launches have homogeneous behaviors, only one of them needs to be simulated to represent the others. To reduce the simulation time of a kernel launch, intra-launch sampling selects the thread blocks that need to be simulated within a kernel launch. inter-launch sampling and intra-launch sampling are

explained in the following sections.

3.3 Inter-Launch Sampling

Using hierarchical clustering, inter-launch sampling groups the kernel launches with homogeneous performance. We simulate only one kernel launch within each cluster and predict that the performance of the other kernel launches within each cluster will be the same as the simulated launch, thus reducing simulating time.

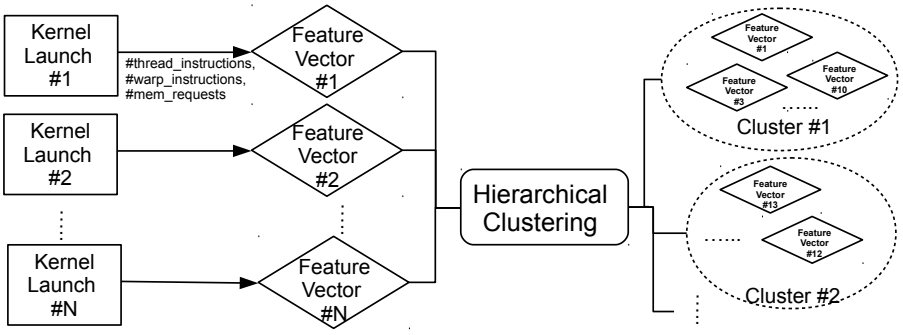


Figure 4: Procedure of Inter-Launch Sampling.

Figure 4 shows the procedure of inter-launch sampling. First, each kernel launch is represented as a feature vector, which is the input to the clustering algorithm. A feature vector describes the characteristics of a kernel launch that contains four features (described below), each of which belongs to one dimension of the vector. Then, hierarchical clustering processes the feature vectors and groups them into clusters. Since each feature vector represents a kernel launch, the kernel launches within the same cluster are believed to have homogeneous performance (IPCs).

The design of feature vectors is important since a feature vector should correctly describe the characteristics of a kernel launch so that the hierarchical clustering can group the kernel launches with homogeneous performance into a cluster. The features with which we chose to compose a feature vector and their performance impact are shown as follows.

- **Kernel launch size**: The number of *thread instructions* of a kernel launch is used as a feature to capture the size of a kernel launch.

- **Control flow divergence**: Because simply capturing the number of thread instructions of a kernel launch does not reflect its degree of control flow divergence, the number of *warp instructions* of a kernel launch is used as a feature to capture the degree of control flow divergence. Even if two kernel launches have the same number of thread instructions, they may have different IPCs due to different degrees of control flow divergence. For example, Kernel Launch 1 executes 32 thread instructions in one warp instruction, while Kernel Launch 2 executes 32 thread instructions in 32 warp instructions.

- **Memory divergence**: Because kernel launches with different numbers of memory requests are likely to have different IPCs, the number of *memory requests* of a kernel launch is used as a feature to capture the degree of memory divergence. The degree of memory divergence is independent of the number of thread blocks and the control flow divergence. For example, a warp instruction that contains 32 thread instructions can issue at least one and up to 32 memory requests if none of the accesses can be coalesced.

- **Thread block variations**: The coefficient of variations (CoV) of thread block sizes of a kernel launch is used as a feature to capture the variations of thread block sizes. Thread block size is defined as the number of thread instructions in a thread block. All the above features are designed as if only one thread block were running per kernel launch. However, a kernel launch typically has multiple thread blocks, and different thread blocks may have a different number of instructions. For example, let's assume that kernel launch 1 has two thread blocks with the number of thread instructions 100 and 100, respectively, and that kernel launch 2 has two thread blocks with the number of thread instructions 160 and 40, respectively. Even though both kernel launches may have the same size (200 thread instructions), they may perform differently because of distinct thread block interleaving situations.

Equation 6 shows the inter-feature vector composed of the above features, each of which is normalized with its average value across all kernel launches so that they have the same order of magnitude.

For kernel launch i,

$$\begin{aligned} inter\_feature\_vector_i =< & Kernel\_Launch\_Size, Control\_Flow\_Divergence, \\ & Memory\_Divergence, Thread\_Block\_Variations > \\ =< & \frac{\#thread\_insts_i}{avg\_thread\_insts}, \frac{\#warp\_insts_i}{avg\_warp\_insts}, \\ & \frac{\#mem\_reqs_i}{avg\_mem\_reqs}, CV\_TB\_size > \end{aligned} \tag{6}$$

Hierarchical clustering takes all inter-feature vectors and groups them into clusters. For each cluster, the kernel launch with the inter-feature vector closest to the center of the cluster is selected as a simulation point that will be sampled by intra-launch sampling.

We chose hierarchical clustering instead of the k-means algorithm used by Simpoint for the following reason. The number of clusters can be determined automatically by setting the *distance threshold* σ, which is the maximum distance between any two points in a cluster. The higher threshold results in fewer clusters, which decreases the total sample size, but the variations within each cluster could be higher, which increases the sampling errors. The appropriate value of the distance threshold depends on the required accuracy and hardware configurations. On the other hand, the k-means algorithm requires a pre-defined number of clusters as an input, which needs another index, such as Bayesian information criterion (BIC) score, to set. In addition, while the results of hierarchical clustering are deterministic, the results of k-means could be affected by the selection of the initial seeds. For all hardware configurations that we tested, the distance threshold = 0.1 for inter-launch sampling can minimize the sampling errors to less than 10% on average.

The proposed inter-feature vector has the following advantages over BBVs, which were used in Simpoint. First, it provides more insight into performance behavior. We found that BBVs are less correlated with performance on GPGPU programs. GPGPU kernels often have very few basic blocks and even the same basic blocks show very distinct performance

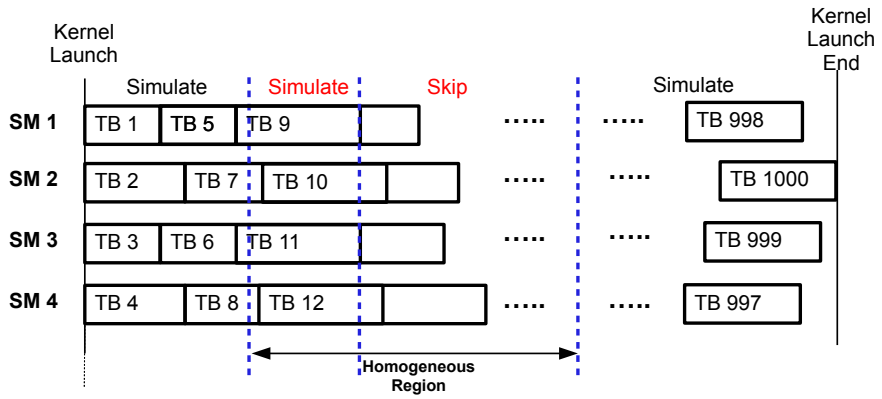


Figure 5: Intra-Launch Sampling

behaviors because of memory divergence, thread block variations, and other behaviors. Furthermore, the same kernel can be launched multiple times but each invocation of the kernel shows particular behaviors, e.g., reduction kernel. Hence, although BBVs can be useful to detect program behavior, the sources of performance variations cannot be solely obtained through BBVs. On the other hand, the proposed vector is also more computationally efficient since it has only four dimensions, while the BBV has a number of dimensions equal to the number of basic blocks in the kernel launch. [1]

3.4 Intra-Launch Sampling

Once inter-launch sampling selects a kernel launch from each cluster for simulation, intra-launch sampling can further reduce the simulation time by sampling the selected kernel launch.

Figure 5 shows a high-level view of intra-launch sampling. Within a kernel launch, our goal is to sample *homogeneous regions*, which have homogeneous performance across multiple thread blocks. In a homogeneous region, a few thread blocks are simulated while the others are skipped so as to reduce the simulation time. The IPC collected from the simulated thread blocks is predicted to be the IPC of the entire homogeneous region. The

[1] The BBV can be added as another feature for improving accuracy with the cost of increased total sample size. The study of such extension is left for our future work.

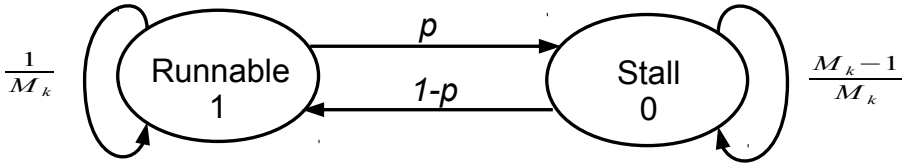


Figure 6: The State Diagram of a Warp. $p = \frac{mem\_insts}{total\_insts}$, $M_k \sim N(\mu, \sigma^2)$

thread blocks not in any homogeneous regions are simulated as usual.

The design presents the following challenges.

- How do we define a homogeneous region? (Section 3.4.1)
- During profiling, how do we identify the location of a homogeneous region? (Section 3.4.3)
- During simulation, how do we sample a homogeneous region? (Section 3.4.4)

3.4.1 The Design of Intra-Launch Sampling

Our design of intra-launch sampling identifies a homogeneous region through the mathematical model that quantifies the IPC variations under different warp interleaving situations that we assume are caused by variable memory latencies due to resource contention and/or queuing delay. Based on our model, such IPC variation has proven to be small. The IPC of a homogeneous region can be predicted as equal to one of its homogeneous intervals. The definitions and proofs are as follows.

Definition 1:

- p is the **stall probability** that is the probability of a warp being stalled, and M is the **average stall cycles** consumed by a stall event. p is modeled as a constant while M is modeled as a random variable following Gaussian distribution. N is the number of warps in an SM.
- A **homogeneous interval** is a sequence of executed instructions from concurrent warps, and each warp has the same p and M.

- A **homogeneous region** is the region with consecutive homogeneous intervals with the same p and M.

Lemma 1 *The IPC variation of a homogeneous interval under different warp interleaving situations caused by random variable M is within a 10% difference of the average IPC.*

Definition 1 shows all definitions that are required for the model. As an example of the input parameter p and M, let us assume that 10% of the instructions of a warp are long latency instructions, and each of which consumes 400 cycles on average. Then, p is a constant equal to 0.1. M is a random variable following $N(\mu, \sigma^2)$ where the $\sigma = \frac{0.1 \times \mu}{1.96}$ so that 95% of randomly picked Ms is within ±10% of μ (400 cycles). Figure 6 shows the state diagram of a warp, which is the basic building block of the model.

Lemma 1 is proven by modeling the IPC variation of a homogeneous interval that includes two steps. First, the IPC of a homogeneous interval is predicted by the Markov chain, which considers the warp interleaving effect. Second, the IPC variation caused by variable M can be predicted by the Monte Carlo method, which performs the Markov chain analysis a finite number of times.

$$
\begin{aligned}
S_{i,j} &= \prod_{x=1}^{N} f(A_i[x],\ A_j[x]), A_i[x],\ A_j[y] \in \{0,1\},\ 0 \le i, j < 2^N - 1 \\
f(A_i[x],\ A_j[x]) &= \begin{cases} A_i[x] \times p + (1 - A_i[x]) \times \dfrac{1}{M_x}, & A_i[x] \neq A_j[x] \\ A_i[x] \times (1-p) + (1 - A_i[x]) \times (1 - \dfrac{1}{M_x}), & A_i[x] = A_j[x] \end{cases} \\
V_i &=< R_0, R_1, R_2 ... R_{2^N-1} >=< 0, 0, 0, ...1 > \\
V_s &= \lim_{n \to \infty} V_i T^n \\
T &= \begin{bmatrix} S_{0,0} & S_{0,1} & ... & S_{0,2^N-1} \\ S_{1,0} & S_{1,1} & ... & S_{1,2^N-1} \\ ... & ... & ... & ... \\ S_{2^N-1,0} & S_{2^N-1,1} & ... & S_{2^N-1,2^N-1} \end{bmatrix} \\
IPC &= 1.0 \times (1 - R_0)
\end{aligned}
\tag{7}
$$

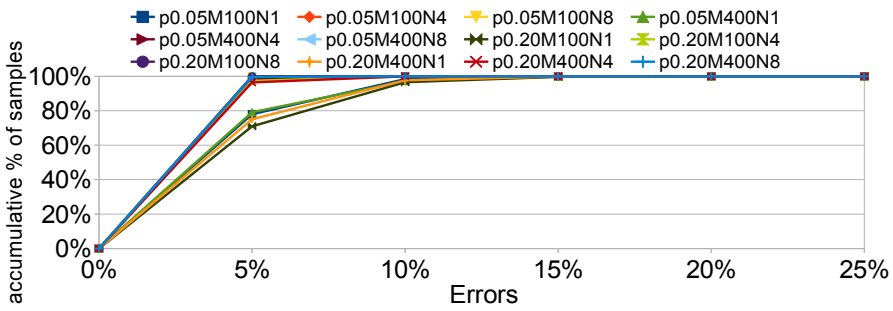


Figure 7: IPC Variation (Each legend shows the p, M, and N values. For example, $p0.05M100N4$ means $p = 0.05$, $M = 100$ and $N = 4$)

To predict the IPC under different warp interleaving situations, the Markov chain is used, as shown in Equation 7. Let us assume that each warp is an independent and identically distributed (i.i.d.) random variable. The size of the transition matrix is $2^N \times 2^N$, as each warp has two states and the number of warps in an SM is N. The transition probability $S_{i,j}$ is the element of the transition matrix T and its calculation is shown as follows. For example, $S_{6,2}$ is the transition probability from 0110 (6) to 0010 (2), in which each bit represents a warp such that Warp 1 is the most significant bit, Warp 2 is the second most significant bit, and so on. In this case, $S_{6,2}$ is the probability of Warp 2 transiting from a runnable to stall state while other warps remain in their current states since the second most significant bit is flipped from 1 (runnable) to 0 (stall) while other bits remain unchanged. After constructing the transition matrix, steady state vector V_s and the expected IPC can be calculated using Equation 7. The initial state vector V_i is $< 000...1 >$ since all warps are initially in the runnable states.

After the IPC is predicted by Markov chain model, we need to quantify the IPC variation caused by variable M using the Monte Carlo method, which performs the Markov Chain a finite number of times (samples). For each sample, M of each warp is randomly selected. The total number of samples is set to 10,000. Figure 7 shows that the IPC variation of a homogeneous interval is low since more than 95% of the samples have less than a 10% difference of the average IPC. In this figure, increasing the memory intensity by increasing p and M leads to higher IPC variation since the mean of IPC is decreasing. By

contrast, increasing the number of warps decreases the IPC variation based on the central limit theorem.

By Lemma 1 , we can conclude that for a homogeneous interval or region, its IPC is stable and not sensitive to different warp interleaving situations. The model provides a theoretical range of IPC values of different warp interleaving situations caused by random variable Ms of a homogeneous interval: For more than 95% of samples with randomly picked Ms, the IPCs are within 10% error of the average IPC.

Our model shares some similarities with other Markov chain models that predicts the IPC of a multithreaded core [40]. However, in these models, M is modeled as a constant, which is unrealistic for the stall events such as DRAM accesses, which have variable latencies resulting from a queuing delay. Our model provides a detailed study that examines IPC variation caused by variable stall latencies M.

3.4.2 Implementation of Intra-Launch Sampling

Our implementation of intra-launch sampling has two components: (1) homogeneous region identification and (2) homogeneous region sampling. The former identifies the homogeneous regions during profiling. The locations of homogeneous regions are stored in the homogeneous region table. The latter samples the homogeneous regions using the homogeneous region table during simulation.

Since profiling is done at the thread block level, to reduce the complexity of implementation, we change the definitions of homogeneous regions and intervals from the warp level to the thread block level. For example, the definition of a homogeneous interval becomes a sequence of executed instructions from concurrent thread blocks with the same p and M.

3.4.3 Homogeneous Region Identification

The basic idea of homogeneous region identification is shown as follows. During profiling, because the thread blocks within a homogeneous region must have the same p and M, identifying a homogeneous region requires the information of which thread blocks are

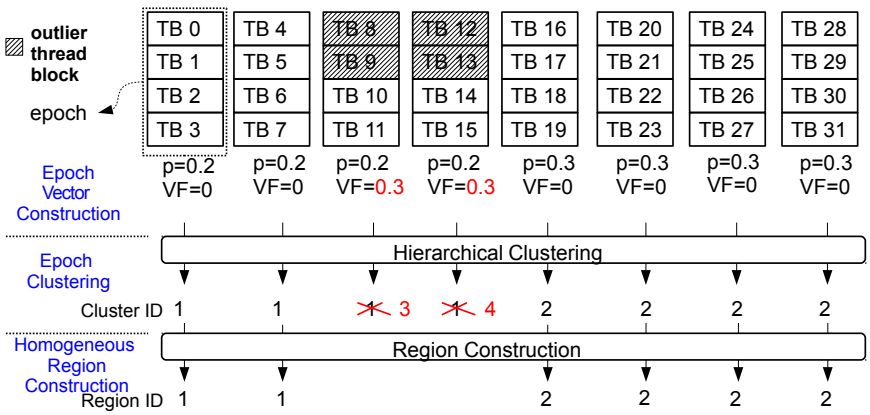


Figure 8: Example of Homogeneous Region Identification

concurrently running at any time period. Since we observe that thread blocks having closer thread block IDs are likely to be running concurrently, we group several thread blocks with closer IDs into an "epoch", as shown in Equation 8. The size of an epoch is equal to the system occupancy. For example, the epoch size of Figure 5 is four thread blocks. An epoch with all thread blocks that have equal p and M is a homogeneous interval. If consecutive epochs has equal p and M, a homogeneous region is constructed by those epochs.

$$epoch_i = \{TB_{occupancy*i}, TB_{occupancy*i+1} ... TB_{occupancy*i+occupancy-1}\} \tag{8}$$

Implementing the idea contains three steps: (1) epoch vector construction, (2) epoch clustering, and (3) homogeneous region construction.

Epoch vector construction: Epoch vector construction converts each epoch into an intra-feature vector that is used to find epochs with the same average stall probability (p) and average stall cycles (M). An intra-feature vector uses the stall probability of an epoch, which is the stall probability averaged over all thread blocks in an epoch, as the only feature. The stall probability (p) and average stall cycles (M) of each thread block are collected as follows. Equation 9 illustrates intra-feature vector.

- **Stall probability (p)** The stall probability of a thread block is approximated using the ratio of the number of memory requests to the total number of instructions collected

for each thread block. The types of memory requests that we consider are global and local memory accesses.

- **Average stall cycles (M)** The average stall cycles of a thread block is not collected since the average stall cycles M cannot be determined without detailed timing simulations. Instead, we assume that if two epochs have the same stall probability (p) for the thread blocks, the average stall cycles (M) from two epochs are also equal since the same kernel code is executed.

$$
\begin{aligned}
X_{epoch_j} &= \{x_{TB_i} | TB_i \in epoch_j\} \\
x_{TB_i} &= The\ number\ of\ memory\ requests\ in\ TB_i \\
Y_{epoch_j} &= \{y_{TB_i} | TB_i \in epoch_j\} \\
y_{TB_i} &= The\ number\ of\ warp\ instructions\ in\ TB_i \\
stall\_probability_{epoch_j} &= \frac{\sum_{TB_i \in epoch_j} (\frac{x_{TB_i}}{y_{TB_i}})}{|X_{epoch_j}|} \\
intra\_feature\_vector_{epoch_j} &= < avg\_stall\_probability_{epoch_j} > \\
variance\_factor_{epoch_j} &= max(CoV(X_{epoch_j}), CoV(Y_{epoch_j}))
\end{aligned}
\tag{9}
$$

Epoch clustering: Hierarchical clustering groups epochs using their intra-feature vectors and generates a cluster ID for each epoch. The epochs with the same cluster ID are believed to have the same p and M. However, some thread blocks with distinct stall probabilities and/or number of instructions, called outlier thread blocks, in an epoch may result in different performance and they may not be captured by hierarchical clustering. Thus, after grouping, post-processing must be done to capture the epochs with outlier thread blocks using the variation factor (VF), which quantifies the dissimilarity between thread blocks using the coefficient of variation (CoV), as shown in Equation 9. If the variation factor of an epoch is larger than some threshold, indicating the existence of outlier thread blocks, the epoch should be removed from the cluster it belongs to and assigned its own cluster.

Homogeneous region construction: After every epoch has been assigned a cluster, a homogeneous region is constructed for a sequence of consecutive epochs that share the

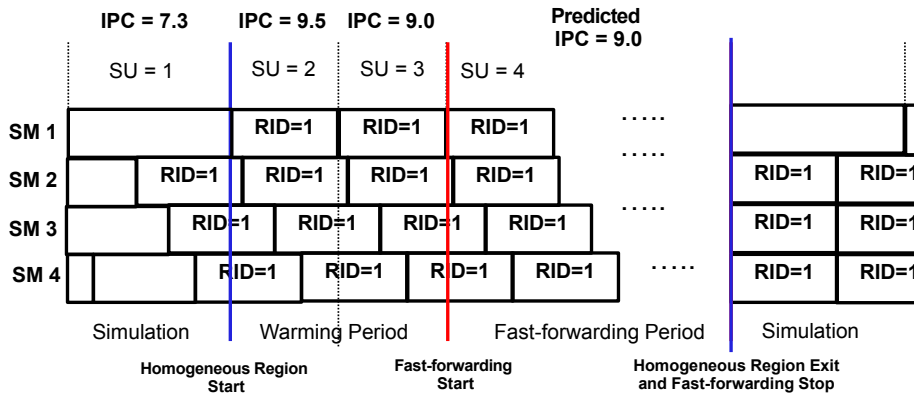


Figure 9: Example of Homogeneous Region Sampling. SU = sampling unit ID. RID = homogeneous region ID.

same cluster. The ID of the cluster, used as the region ID, is assigned to every thread block in a region. For every homogeneous region, all its thread blocks and their corresponding region IDs are stored in the homogeneous region table, as shown in Table 3.

Figure 8 illustrates homogeneous region identification. During epoch vector construction, because the first four epochs have the same stall probability (0.2), their intra-feature vectors are equal. Similarly, the remaining four epochs have the same intra-feature vectors. During epoch clustering, the first four epochs are grouped into one cluster while the remaining epochs are grouped into the other cluster. Because the variation factors of the third and fourth epochs are high indicating the existence of outlier thread blocks, the epochs are removed from the cluster. During homogeneous region construction, two homogeneous regions are identified. The first and second epochs belong to one homogeneous region while the last four epochs belong to the other homogeneous region.

Table 3: Example of Homogeneous Region Table

Region ID	Start TB ID	End TB ID
1	0	40
2	50	120

3.4.4 Homogeneous Region Sampling

After the locations of homogeneous regions are determined, we need to sample homogeneous regions using the homogeneous region table during simulation. The basic idea is to simulate only a few thread blocks within a homogeneous region and skip the other thread blocks in the region. The IPC of the region is predicted as equal to the IPC of the simulated thread blocks. For the thread blocks not in any homogeneous regions, they are simulated as usual.

To sample a homogeneous region, we must define the size of a sampling unit. The IPC of a sampling unit represents the IPC of a homogeneous region. We define a sampling unit as the interval between the start and end of a specified thread block. The first specified thread block is the very first dispatched thread block when the simulation begins. Once the current one is retired, another thread block will be specified. Compared to the design of sampling units with a fixed number of instructions, this design ensures that every sampling unit has a similar stall probability since the specified thread block executes the whole kernel code, which potentially captures the behaviors of the whole kernel. In addition, as it requires no instruction counting for determining the length of an interval, the design simplifies the complexity of implementation.

Sampling a homogeneous region contains three steps: (1) entering, (2) sampling and (3) exiting a homogeneous region. Each step is described as follows.

Entering: Entering a homogeneous region happens when all concurrently running thread blocks belong to the same homogeneous region in the homogeneous region table.

Sampling: Once a homogeneous region is entered, sampling a homogeneous region breaks into two periods: (1) a *warming period* and (2) a *fast-forwarding period*. During the warming period, the thread blocks are simulated as usual, and the IPC of the current sampling unit is recorded. If the IPC difference between the current and previous sampling units is less than 10%, the cache states are considered stable and the fast-forwarding period

Table 4: IPCs of Inter-Launch and Intra-Launch Sampling

Inter-Launch sampling
$$total\_CPI = \sum_{cluster_k \in clusters} representative\_kernel\_launch\_CPI_{cluster_k} \times cluster\_weight_{cluster_k}$$ $$cluster\_weight_{cluster_k} = \frac{\sum_{launch_p \in cluster_k} \#kernel\_launch\_insts_{launch_p}}{total\_insts}$$
Intra-Launch sampling
$$launch\_insts = \sum_{i \in simulated\_TBs} \#TB\_insts_i + \sum_{j \in skipped\_TBs} \#TB\_insts_j$$ $$launch\_cycles = \sum_{i \in simulated\_TBs} \#TB\_cycles_i + \sum_{j \in skipped\_TBs} \frac{\#TB\_insts_j}{homogeneous\_region\_IPC_j}$$ $$launch\_CPI = \frac{launch\_cycles}{launch\_insts}$$

begins. Otherwise, the warming period continues. During fast-forwarding period, the dispatched thread blocks are skipped while the IPC of the homogeneous region is predicted to be the IPC of the last sampling unit in the warming period.

Exiting: The homogeneous region exits when the newly dispatched thread block region ID differs from the current homogeneous region ID. Then, the simulation continues.

Figure 9 illustrates homogeneous region sampling. Initially, because not all thread blocks of sampling unit 1 are in a homogeneous region, the homogeneous region is not entered until sampling unit 2. Sampling units 2 and 3 belong to the warming period of sampling step. Since the IPC difference between sampling units 2 and 3 is less than 10%, the cache states are stable and the fast-forwarding period starts at sampling unit 4. During fast-forwarding period, the remaining thread blocks from the region are skipped. The homogeneous region exits when the thread block that does not belong to the region is dispatched. Then the simulation continues as usual.

Table 4 summarizes how the overall IPC is predicted when inter-launch and intra-launch sampling techniques are applied. Note that these two sampling techniques, which are orthogonal, can be applied independently.

3.5 Evaluation

3.5.1 Evaluation Configurations

We use Macsim, a cycle-accurate trace-driven simulator, as our simulation platform in which the homogeneous region sampling technique is implemented. The detailed simulation configurations, based on the NVIDIA Fermi architecture, are listed in Table 5.

Table 5: Simulation Configuration.

Number of cores	14
Front End	Fetch width: 1 warp-instruction/cycle, 4KB I-cache, 5 cycle decode
Execution core	1.15 GHz, 1 warp-instruction/cycle, 32-wide SIMD execution unit, in-order scheduling instruction latencies are modeled according to the CUDA manual
On-chip caches	16 KB software managed cache 16 KB L1 cache, 128B line, 8-way assoc 768 KB L2 cache, 128B line, 8-way assoc
DRAM	1.15GHz, 2 KB page, 16 banks, 6 channels, FR-FCFS scheduling policy

Table 6 shows the evaluated benchmarks, long running benchmarks from several benchmark suites. For those with multiple kernels, we select the kernel that has the longest running time. Figure 10 depicts our method of classifying regular and irregular kernels based on thread block sizes. The X axis is the thread block ID, while the Y axis is the thread block size. Types (a) is a *regular kernel* since the thread block sizes exhibit particular patterns. Type (b) is an *irregular kernel*.

We evaluate the following sampling techniques.

- **TBPoint:** This approach applies both inter-launch sampling and intra-launch sampling. The distance threshold of hierarchical clustering is for the former 0.1 and for the latter 0.2. The variation factor is 0.3.

Table 6: Evaluated Benchmarks (Type I: irregular kernel, Type II: regular kernel)

Benchmark	Suite	Type	# Kernel launches	# Thread blocks	Abbreviation
BFS	lonestar	I	41	10619	bfs
SSSP	lonestar	I	49	12691	sssp
MST	lonestar	I	9	2331	mst
MRI-Gridding	parboil	I	7	18158	mri
SPMV	parboil	II	50	38250	spmv
LBM	parboil	II	6	108000	lbm
CFD	rodinia	II	100	50600	cfd
Kmeans	rodinia	II	30	58080	kmeans
Hotspot	rodinia	II	1	1849	hotspot
StreamCluster	rodinia	I	21	2688	stream
BlackScholes	sdk	II	87	41760	black

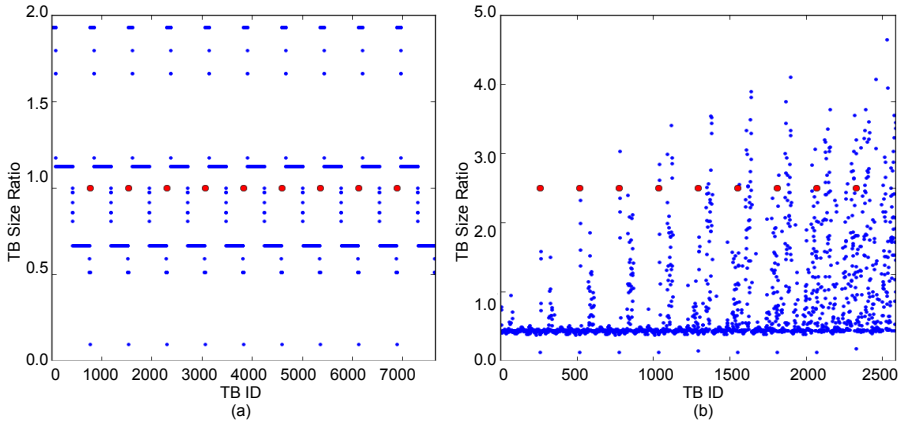


Figure 10: Different Kernel Types. (a) Regular. (b) Irregular kernel. A red dot indicates the start of a kernel launch while a blue dot indicates a thread block. Thread block size ratio is the thread block size, which is the number of thread instructions in a thread block, normalized by the average thread block size across all thread blocks.

- **Random sampling (Random):** We conduct a full simulation in which we collect IPC for every sampling unit with one million instructions and randomly select 10% sampling units.

- **Ideal-Simpoint:** For Ideal-Simpoint, we collect the BBV and IPC for every sampling unit with one million instructions. Then, we use the Simpoint tool for clustering BBVs and simulation point selection. The overall IPC can be calculated using Equation 5. The difference between Ideal-Simpoint and the original Simpoint is that

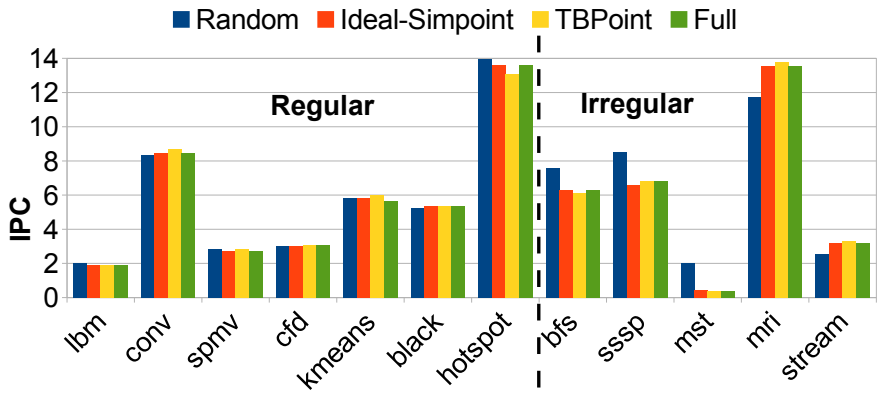


Figure 11: Overall IPC. (The overall IPC is defined as $\sum_{k \in SMs} \frac{\#warp\_insts_k}{\#cycles_k}$).

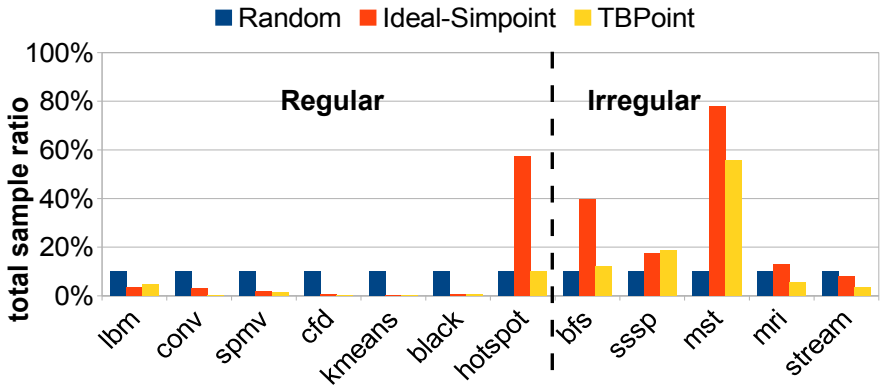


Figure 12: Total Sample Size (Ratio). (The total sample size is defined as $\sum_{k \in SMs} \frac{\#simulated\_warp\_insts_k}{\#warp\_insts_k}$).

the full timing simulation is performed in Ideal-Simpoint to collect BBV from concurrent warps in every sampling unit, so Ideal-Simpoint is not a viable solution for the GPGPU platform. Without a full timing simulation, what instructions are executed by each warp in every sampling unit is unknown because of the unpredictable effect of warp scheduling.

- **Full:** The total IPC is collected through the full simulation with no sampling techniques applied.

3.5.2 Comparisons

Figure 11 shows the total IPCs of three approaches. The geometric mean of the sampling errors of Random, Ideal-Simpoint, and TBPoint are 7.95%, 1.74%, and 0.47%, respectively. Random has a much higher error rate, especially for the irregular kernels. The sampling errors of Ideal-Simpoint and TBPoint are less than 2% for all benchmarks, except mst. Ideal-Simpoint has the highest errors (8.5%) for mst because the BBVs cannot detect the thread-level parallelism (TLP) changes caused by the outlier thread blocks, which have considerably more instructions than the others.

Figure 12 shows the total sample size of the three approaches. The geometric mean of the total sample size of Random, Ideal-Simpoint, and TBPoint are 10%, 5.4%, and 2.6%, respectively. For regular kernels, while Random takes many more samples than needed since it cannot detect the regularity in a kernel, the other approaches have a similar sample size. For irregular kernels, the average sample size of TBPoint is 50.5% of the sample size of Ideal-Simpoint because the intra-feature vector captures changes in stall probabilities. mst has a high sample size (55%) because to achieve high accuracy, TBPoint needs to simulate the epochs with outlier thread blocks.

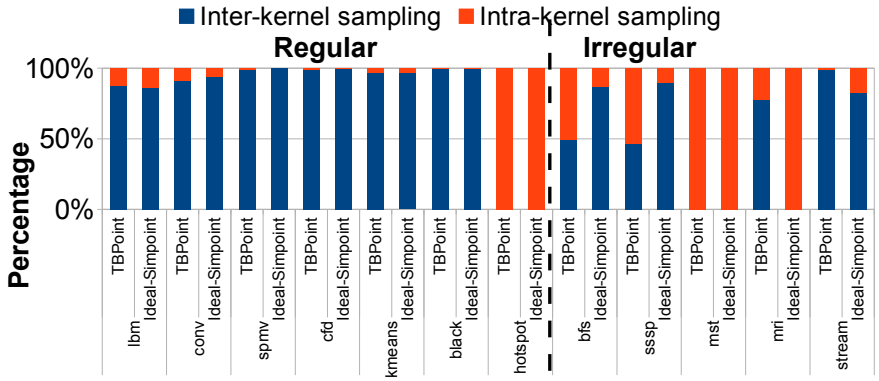


Figure 13: Breakdown of the Relative Percentage of Skipped Instructions from Inter-Launch and Intra-Launch Sampling.

Figure 13 shows the relative percentage of skipped instructions from inter-launch and

intra-launch sampling. For regular kernels, most savings come from the inter-launch sampling for both approaches because all kernel launches are homogeneous, except binomial and hotspot, which only have one kernel launch. For irregular kernels, the percentage of inter-launch sampling decreases since different kernel launches are not homogeneous. For mst, most savings come from intra-launch sampling since different kernel launches have different sizes. For stream, hundreds of homogeneous kernel launches cause the most savings to come from inter-launch sampling.

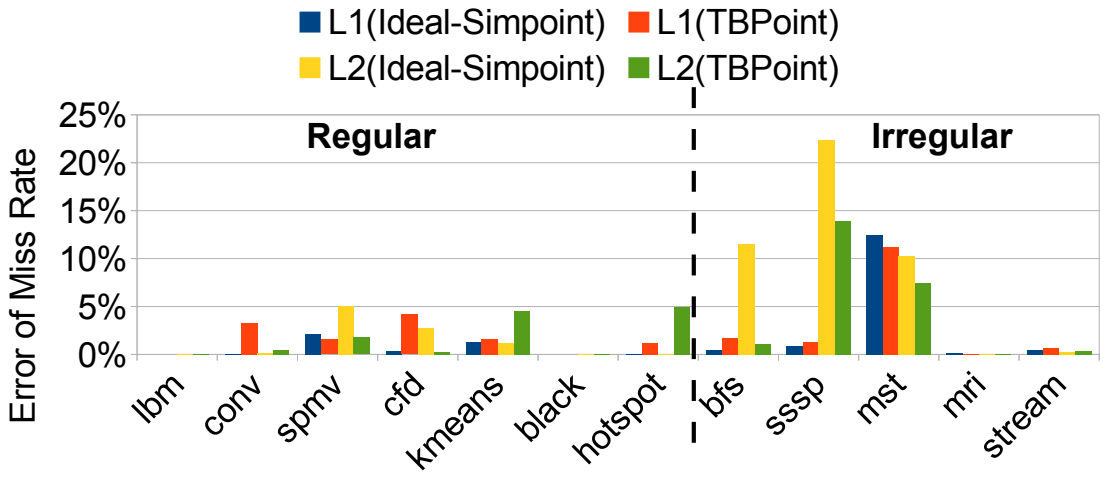


Figure 14: Errors of L1/L2 cache miss rates.

Figure 14 shows the errors of L1/L2 cache miss rates. The average errors of the L1 miss rates of Ideal-Simpoint and TBPoint are 1.5% and 2.2%, respectively, while the average errors of the L2 miss rates are 4.5% and 2.9%, respectively. The errors are higher for irregular kernels because of the random memory accesses. Some kernels have errors close to or equal to 0 because they have no data localities, which causes a very high miss rate, such as black and lbm. The L2 error is similar to the L1 error but it is much higher for some kernels, such as sssp, because of the low L2 miss rates, leading to errors that are more sensitive to the mispredictions of cache miss rates.

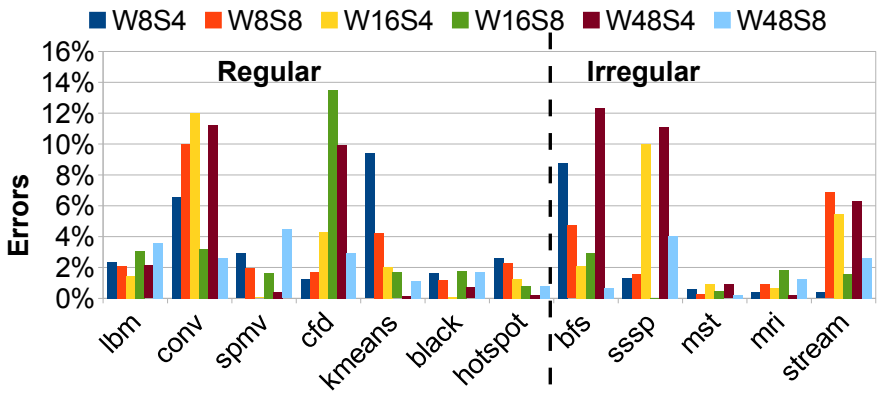


Figure 15: Sampling Errors of Different Hardware Configurations. W is the number of warps in an SM, and S is the number of SMs

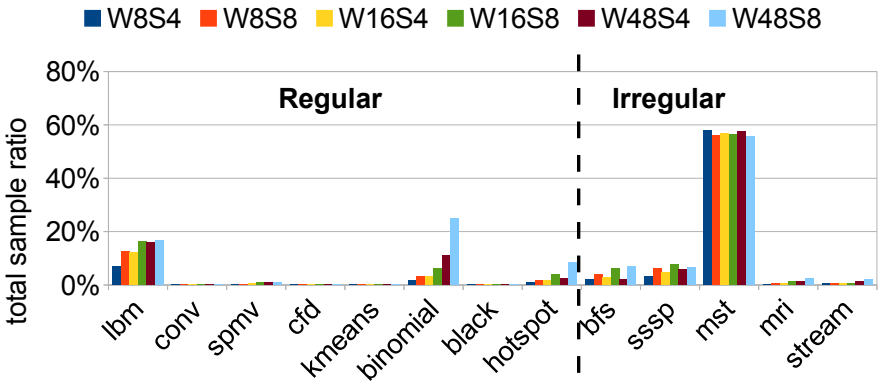


Figure 16: Total Sample Sizes (Ratios) of Different Hardware Configurations

3.5.3 Sensitivity Analysis

TBPoint can quickly adapt to the hardware configurations with system occupancy change, such as those with a different number of SMs or warps. The required computations are shown as follows. For intra-launch sampling, the homogeneous region identification needs to be redone since the epoch size changes according to system occupancy. The overhead is small compared to that of profiling all thread blocks using GPUOcelot, which only needs to be done once, regardless of the system occupancy. For inter-launch sampling, because the kernel characteristics do not change when the system occupancy changes, the clustering of inter-launch sampling needs to be done only once.

Figure 15 shows the errors of the hardware configurations with different system occupancies. The maximum error rate is less than 14%. Some kernels exhibit high variation in the error rate for the following reasons. First, the cache states are not fully constructed because of the lack of cache accesses during fast-forwarding, leading to inaccurate IPC after fast-forwarding. Second, if a kernel is more memory intensive, IPC variation is higher. However, IPC variation decreases if the system occupancy increases.

Figure 16 shows the sample sizes of the hardware configurations with different system occupancies. For regular kernels, when the system occupancy is low, the total sample size is lower since the size of the epoch size is proportional to the system occupancy. However, for irregular kernels, low system occupancy may have a high sample size because of the longer warming period, which occurs when the sampling units in the warming period exhibit high IPC variations resulting from incomplete cache states. In the cases of the cache-sensitive kernels, such as bfs and sssp, the low system occupancy usually takes a longer warming period.

3.6 Summary

The proposed TBPoint system raises the possibility of simulating large-scale GPGPU applications by significantly reducing the GPGPU simulation time while achieving low sampling error and total sample size. Moreover, the design of TBPoint achieves three requirements of a good profiling-based sampling technique: hardware independence, one-time profiling, and mathematical model support.

As a greater range of algorithms is converted to GPGPU kernels, finding the performance bottlenecks using detailed timing simulations has growing importance. TBPoint provides an efficient method of simulating large-scale GPGPU kernels.

CHAPTER 4

GPUMECH: GPU PERFORMANCE MODELING TECHNIQUE BASED ON INTERVAL ANALYSIS

Chapter 3 introduces a sampling methodology for GPGPU kernels. To reduce the simulation time, an orthogonal approach is to improve the simulation speed. Instead of using detailed timing simulation, this chapter presents an abstracted simulation approach, GPUMech [57], which leverages both functional simulation and analytical modeling to model the performance. On average, the overall speed is 97x faster than the detailed simulation while having only 13.2% sampling error compared to the full simulation. In addition, it can visualize the performance bottlenecks in microarchitecture level using CPI stack, which help architects gain insights into workload behaviors. While detailed timing simulation is irreplaceable, the proposed approach can be leveraged in the early stage of processor design for design space exploration.

4.1 Introduction

Performance models are attractive solutions to speed up simulation speed, especially for exploring early-stage design options. The models can provide greater insight for understanding architectures and applications. For example, analytical models have been proposed for both CPUs and GPUs [58, 59, 60, 40]. However, because of the relatively short history of GPU architectures, fewer analytical models and fast-simulation methods have been developed for GPUs.

Even though several GPU performance models are proposed [44, 41, 42, 43], unfortunately, most of them have focused on helping software optimizations. These models use hardware performance counters or program analysis to understand the performance bottlenecks of software, but they have not been used to explore different GPU architecture design options. Jia et al. [61] presented a regression-based GPU model that can explore the

design space with various hardware parameters, but this model lacks in providing insight into performance changes.

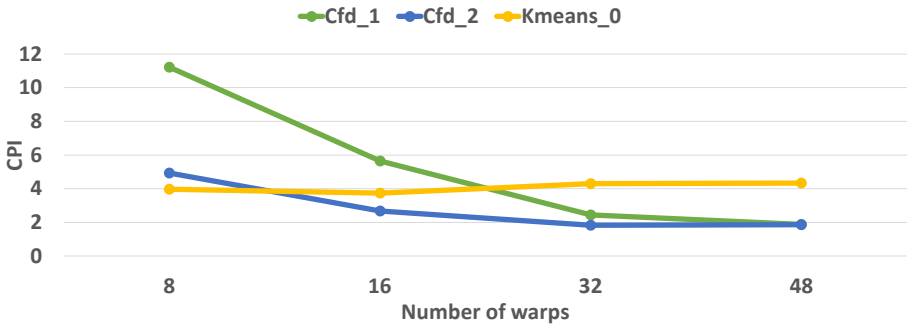


Figure 17: The CPIs of different number of warps in an SM

For example, Figure 17 shows CPIs with different number of warps in an SM for three kernels from Rodinia suite [62]. The regression-based model cannot provide the information on what limits the performance at each design point. In addition, it incurs high overhead since training the model requires thousands of detailed simulations.

Analytical models are simple and fast enough to get a first-order performance estimation, but they often have higher errors than detailed timing simulations. As an alternative solution to traditional analytical models, interval analysis was proposed [9, 10], which uses both trace-driven functional simulators and an analytical model to estimate core-level performance. The key difference between traditional analytical models and interval analysis is that while traditional analytical models use the average (or total summation) value of events, interval analysis traverses the instruction trace and tracks the performance degradation events, such as cache misses and branch mispredictions, and then a model is used to estimate the performance impact of each event. Tracking the events only requires functional simulations, e.g., cache simulation, which results in at least a 100x speedup compared to cycle-level simulations [38, 63, 39]. Hence, the interval analysis technique is faster than detailed cycle-level simulation while having a higher accuracy than traditional analytical models. Furthermore, interval analysis also provides the *CPI stack*, which shows the breakdown of different types of performance loss events and can provide an insight into

performance behavior.

Nonetheless, no previous interval analysis technique has been proposed for GPU architectures. Prior interval analysis techniques cannot be naïvely applied to GPU architectures for two reasons. First, multithreading is not considered. Techniques have been developed for single-core or multi-cores in which only one thread executes per core but not for multi-threaded architectures in which multiple threads concurrently execute to hide the memory latency. Second, control and memory divergence are not considered, since they are unique behaviors in SIMD/SIMT architectures.

To overcome these limitations, we propose an interval analysis technique, called *GPUMech*, the first interval analysis technique for GPU architectures. GPUMech profiles the instruction trace of every warp and uses a clustering algorithm to identify the representative warp. This is critical for kernels that have control-flow divergent warps. To model multithreading, GPUMech introduces the concept of *non-overlapped instructions* to accurately model the latency hiding capability of a scheduling policy. To handle performance degradations resulting from memory divergence, the resource contention in the memory system is also modeled.

4.2 Background and Motivation

4.2.1 Motivation

Multithreading: To apply CPU-based interval analysis to an in-order multithreaded architecture, e.g., GPU, several challenges exists. First, the previous interval analysis techniques have only been developed for single-threaded applications. Extending the model from a single-threaded architecture to a multithreaded architecture requires emulating instruction scheduling from multiple threads, which could incur high overhead.

$$IPC_{core} = IPC_{single-warp\_performance} \times \#warps \tag{10}$$

Alternatively, as shown in Eq. 10, a naïve approach to predict the performance of a multithreaded architecture is to use interval analysis for a single warp, which is then multiplied

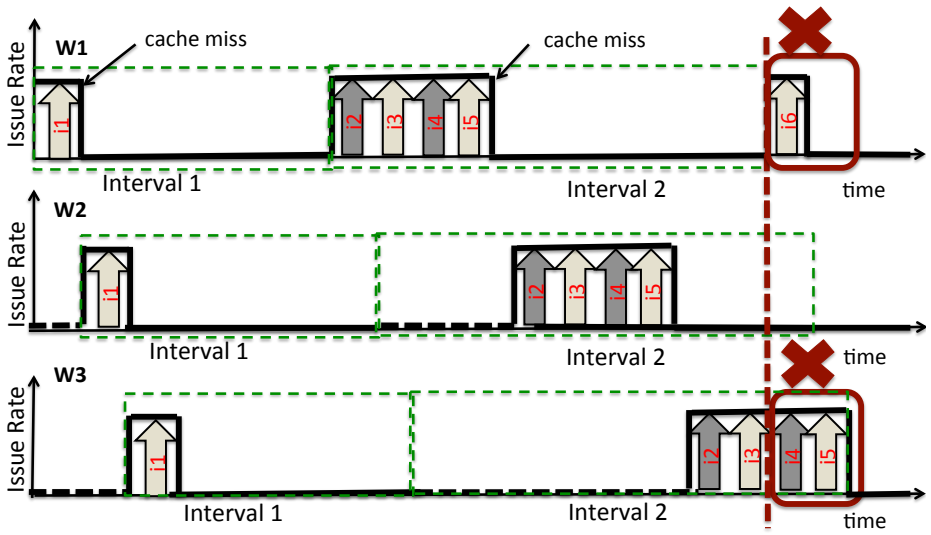


Figure 18: The case of interval analysis with multiple warps. A green box represents an interval. The red boxes contain the instructions that do not overlap with the stall cycles. (W: warp, i: instruction.)

by the number of warps to get the final performance of multithreading. It assumes that the total cycles of a single warp remain unchanged while all instructions from the other warps can be issued during the stall cycles of the single warp to hide the stall cycles. However, this approach does not accurately model the warp scheduling policy, in which case the total cycles may change since *not all instructions from the other warps can overlap with the stall cycles*. The extra cycles, which are needed to issue the instructions that do not overlap with stall cycles, result in performance loss.

Figure 18 shows an example. Assume that 3 warps are running on a core and each of which has two intervals: Interval 1 has 1 instruction while Interval 2 has 4 instructions. Both intervals have 10 stall cycles. The issue rate is 1 instruction per cycle. During interval 1, a total of three instructions can be issued, so the IPC for the core is 3/11, which is the same result as using Eq. 10 ($1/11 \times 3$). However, this is an optimistic assumption. For Interval 2, warp 3 cannot issue instructions 4 and 5 during the stall cycles of warp 1. To issue those instructions, extra issue cycles are required. In Section 4.4.1, we will illustrate and model two widely used scheduling polices: round-robin (RR) and greedy-then-oldest (GTO) policies to consider the extra issue cycles incurred in both polices.

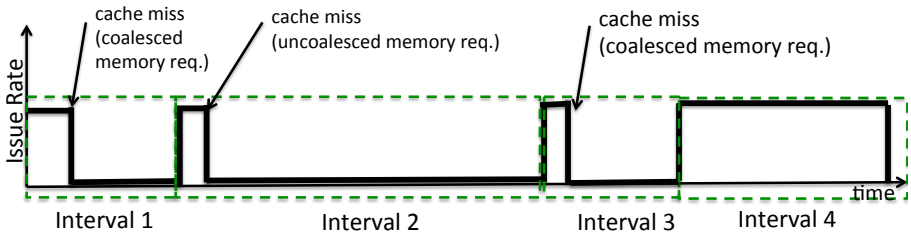


Figure 19: Interval analysis with different degrees of memory divergences.

Resource Contention in the Memory System: The second issue of the naïve approach is that it ignores resource contention in the memory system, which is likely to occur with multithreaded architectures. One of the unique features of a GPU architecture is that the number of memory requests from a SIMD/SIMT instruction varies significantly depending on the degree of *memory divergence*, referred to as uncoalesced memory accesses. Figure 19 illustrates that stall cycles can vary significantly because of queuing delays caused by memory divergence. To address the problem, we model the queuing delays of limited MSHR entries and DRAM bandwidth, shown in Sections 4.4.2.1 and 4.4.2.2.

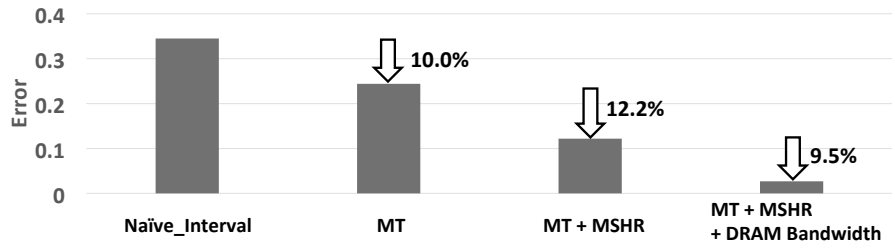


Figure 20: The errors of a kernel from the SRAD benchmark. The arrows represent the error reduction from its left-side configuration.

A Case Study: To provide a more concrete example, we show how modeling different components reduces the error gradually. Figure 20 shows the error of interval analysis compared with a detailed timing simulation for an SRAD kernel that has divergent memory accesses from Rodinia [64]. `Naive_Interval` bar shows the error when using Eq. 10. `MT` models the round-robin scheduling policy while `MSHR` and `DRAM Bandwidth` model resource contention for the MSHRs and DRAM bandwidth, respectively. As the result shows, modeling both scheduling policy and resource contention is critical to improve the

accuracy of the interval analysis technique for GPUs.

4.3 Single-Warp Model

4.3.1 GPUMech Overview

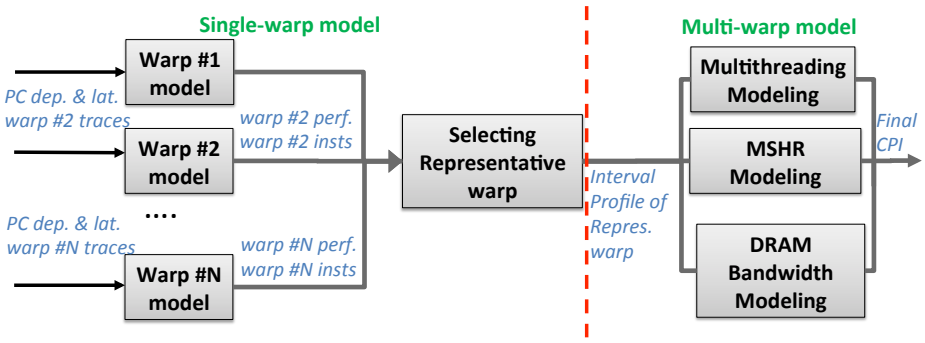


Figure 21: GPUMech Overview

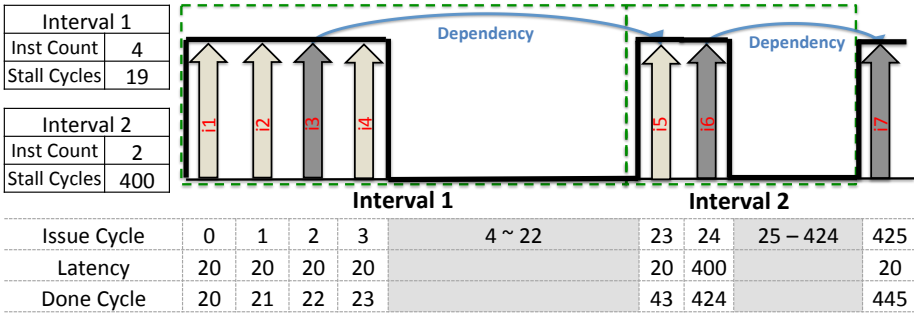
Figure 21 shows the performance model of GPUMech, including single-warp and multi-warp models. The input of the single-warp model is generated by the input collector described in Section 4.5. For the *single-warp model*, the *interval algorithm*, described in the next subsection, models an in-order execution of a warp and forms its intervals. "*Selecting the representative warp*" chooses the representative warp that has an interval profile similar to that of the majority of warps.

Eq. 11 shows an *interval profile* of a warp, defined as a collection of intervals constructed by the interval algorithm. Each interval has the information of the number of instructions and stall cycles. The interval profile of the representative warp is sent to the multi-warp model.

$$interval\_profile = \{[\#interval\_insts_i, stall\_cycles_i] \quad , i \in intervals\} \tag{11}$$

Eq. 12 shows the final CPI, which is the sum of the CPIs of multithreading and resource contention, predicted by the *multi-warp model*.

$$CPI_{final} = CPI_{multithreading} + CPI_{rc\_contention} \tag{12}$$



Issue Cycle	0	1	2	3	4 ~ 22	23	24	25 – 424	425
Latency	20	20	20	20		20	400		20
Done Cycle	20	21	22	23		43	424		445

Figure 22: Intervals of a warp. (The shaded boxes indicate the stall cycles in which no instructions are issued. The instructions in dark gray are the ones that lead to stall cycles.)

4.3.2 Interval Algorithm

The purpose of the interval algorithm is to construct a warp's interval profile assuming an in-order execution model. The inputs of the interval algorithm are (1) the instruction latency per static instruction and (2) the instruction trace of a warp tagged with dependency information. The generated interval profile of each warp will be used to select the representative warp.

Figure 22 illustrates intervals of a warp. The done cycle is equal to the issue cycle plus the instruction latency. instruction 3 ($i3$) leads to stall cycles because instruction 5 ($i5$) depends on it. On the other hand, other instructions in the first interval have no dependent instructions, so they do not cause any stall cycles.

The interval algorithm traverses every instruction in the instruction trace of a warp. It determines the issue cycle of each instruction using Eq. 13, assuming that one instruction can be issued every cycle. An interval is formed if the issue cycle of the current instruction is not equal to the issue cycle of the previous instruction plus one, since it indicates that the stall cycles are incurred between the two instructions. In that case, the previous instruction is included into the newly formed interval, while the current instruction will belong to the interval that will be formed next. The algorithm proceeds until every instruction in a warp belongs to an interval.

$$issue\_cycle(inst_{k+1}) = max(issue\_cycle(inst_k) + 1, \quad done\_cycle(source\_inst_{k+1}) + 1) \tag{13}$$

4.3.3 Selecting Representative Warp

Once the interval profile of each warp is collected, we select one warp to predict the overall performance. However, as some warps may have different degrees of control divergences, their interval profiles could be quite different. Using the interval profile from a random warp as an input to the multi-warp model can lead to high error, since the warp may not be representative of the other warps. Therefore, to reduce the errors caused by control divergences, we attempt to identify the most representative warp using clustering.

Specifically, GPUMech uses the k-means algorithm, which requires two inputs: (1) the number of clusters and (2) the feature vector of each warp. We set the number of clusters to two. One cluster is to capture the majority warps with similar interval profiles while the other cluster is to capture the outlier warps.

$$warp\_perf = \frac{\sum_{i \in intervals}(\#interval\_insts_i)}{\sum_{i \in intervals}(\frac{\#interval\_insts_i}{1.0(issue\ rate)} + stall\_cycles_i)} \tag{14}$$

A feature vector is used to characterize the interval profile of a warp. One simple approach is to represent each interval as one dimension of the vector. Then the number of dimensions is equal to the number of intervals of a warp. However, clustering the vectors is not scalable, since a feature vector for a long-running warp may have thousands or more dimensions (intervals). Instead, we use the **warp performance**, which is the IPC when a warp is running alone on a core, as shown in Eq. 14, as one dimension of the vector. Our assumption is that if two warps have different interval profiles, their warp performances are different, and vice versa. However, the warp performance may not capture the warps with different number of intervals, e.g., distinct iteration counts, even though their performances are the same. Thus, **the number of instructions** of a warp is also used as another dimension of the vector. The final feature vector of a warp is shown in Eq. 15. Both features are normalized by the respective average value over all warps.

$$feature\_vector_w = [\frac{warp\_perf_w}{avg\_warp\_perf}, \frac{\#warp\_insts_w}{avg\_warp\_insts}], w \in warps \tag{15}$$

The representative warp is selected as the warp closest to the center of the largest cluster,

since its interval profile is more likely to be representative of the majority of warps and thus will be used as the input of the multi-warp model.

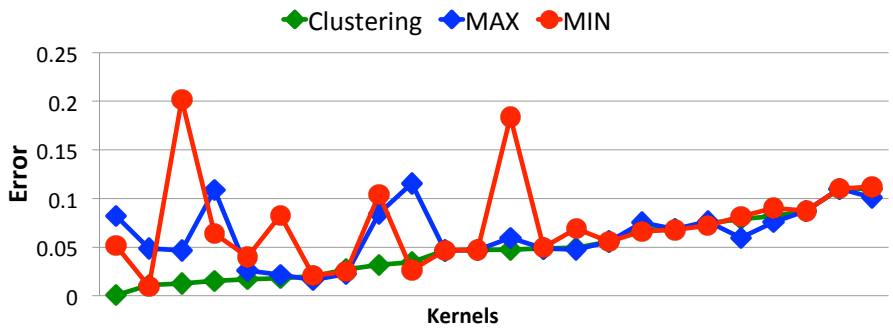


Figure 23: Errors from different representative warp selection methods. Each tick represents a control divergent kernel and data points are sorted by the errors of Clustering approach.

Figure 23 shows the errors of GPUMech when it uses three different methods to select a representative warp. (The baseline configuration is shown in Table 7). The three selection methods are (1) *MAX*: selecting the warp with the maximum warp performance; (2) *MIN*: selecting the warp with the minimum warp performance and (3) *Clustering*: selecting the warp using clustering. For some kernels, the three cases have similar errors, indicating that the difference of interval profiles of warps is negligible. For the others, the clustering method usually has the best accuracy.

4.4 Multi-Warp Model

The multi-warp model uses the interval profile of the representative warp and predicts the performance under multiple warps, which is the typical situation on a GPU core. Specifically, it models the performance improvement due to multithreading and the degradation due to resource contention.

4.4.1 Modeling Multithreading

We model multithreading by assuming that multiple warps are running on a core without resource contention. The key idea is to count the number of instructions of the remaining warps that hide the stall cycles of the representative warp. Later, we add resource

contention modeling. Ideally, all instructions in the remaining warps hide the stall cycles. However, some instructions do not hide the stall cycles, leading to sub-optimal performance. In the following, we illustrate and model two popular warp scheduling polices: round-robin (RR) and greedy-then-oldest (GTO) [65] polices.

In our terminology, among the warps assigned to a given core, the warps other than the representative warp are referred to as the **remaining warps**. From the remaining warps, the instructions that hide the stall cycles of the representative warp are referred to as **overlapped instructions** while the other instructions are called **non-overlapped instructions**. The "intervals" and "stall cycles" are referred to as those of the representative warp, unless otherwise specified. We also differentiate between "schedule" and "issue". A warp is scheduled means that a warp is selected by the scheduler to issue instructions. But it may not be able to issue due to stalls. In this case, the next candidate warp is scheduled in the same cycle until the warp that can issue instructions is scheduled.

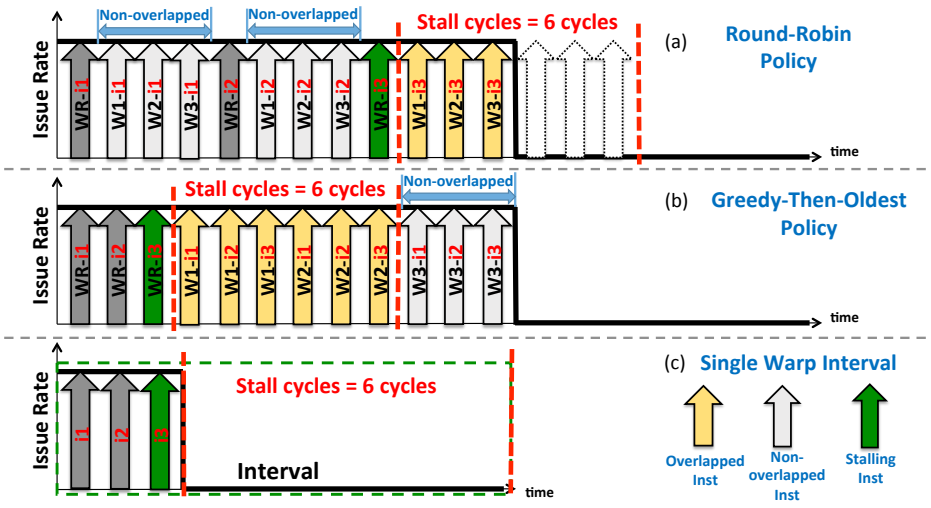


Figure 24: The cases of non-overlapped instructions of RR and GTO policies. (WR: representative warp.)

4.4.1.1 The cases of non-overlapped instructions

Figure 24 illustrates how non-overlapped instructions are incurred. To begin with, we assume that four warps are running on a core with the issue rate equal to 1 instruction per

cycle, and they all have the same interval profile. For simplicity, we use one interval that has 3 instructions and 6 stall cycles, as shown in Figure 24(c).

Figure 24(a) illustrates the case for the round-robin scheduling policy, which issues an instruction every cycle in round-robin fashion. Since the instructions are issued *regardless whether the representative warp is stalled or not*, some instructions from the the remaining warps do not hide the stall cycles. As shown in the example, three instructions from the remaining warps hide the stall cycles of the representative warp while the other instructions are interleaved with the first two instructions from the warp. The number of overlapped instructions is 3 (the instructions after WR-i3, i.e., W1-i3, W2-i3, W3-i3) while the number of non-overlapped instructions is 6 (W1-i1, W2-i1, W3-i1, W1-i2, W2-i2, W3-i2). The remaining stall cycles is 3 (the number of stall cycles minus the number of overlapped instructions). By contrast, the remaining stall cycles of the naïve prediction is 0 since all instructions from the remaining warps overlap with the stall cycles ($9 = 3 \times 3 > 6$).

Figure 24(b) illustrates the case for the greedy-then-oldest scheduling policy, which issues instructions from the same warp until it stalls. Then, the warp that has the oldest instruction is issued next and so on. As shown in the example, all instructions from W1 and W2 are overlapped with the stall cycles. However, the non-overlapped instructions are still incurred since 3 instructions (W3-i1, W3-i2, W3-i3) are not overlapped with any stall cycles. Ideally, those instructions could potentially hide the stall cycles of the next interval. But in this case, since W3 has the oldest instruction, the representative warp has to wait even though it is ready to issue.

With consideration of non-overlapped instructions, Eq. 16 shows the calculation of the multithreading CPI. The non-overlapped instructions become extra cycles added to the total cycles of the representative warp (*total_cycles*) since they do not overlap with the stall cycles. For the ease of explanation, we again assume that the issue rate is 1 instruction/cycle and the warp scheduler can issue an instruction from a warp every cycle.

$$CPI_{multithreading} = \frac{total\_cycles + \frac{\#total\_nonoverlapped\_insts}{1.0(issue\ rate)}}{\#warps \times \sum_{i \in intervals} \#interval\_insts_i} \tag{16}$$

Eq. 17 shows how the total non-overlapped instructions is counted. The basic idea is to aggregate per-interval non-overlapped instructions, which are calculated probabilistically based on the scheduling policy. The probabilistic counting takes into account cases in which the intervals of warps could randomly interleave while Figure 24 illustrates the case where the warps are well aligned.

$$\#total\_nonoverlapped\_insts = \sum_{i \in intervals} \#nonoverlapped\_insts_i \tag{17}$$

Before modeling the non-overlapped instructions for a given scheduling policy, we first define the **issue probability** as the probability that a warp can issue an instruction in a cycle, as shown in Eq. 18. Issue probability is computed with a representative warp, and we assume that all other remain warps have the same uniform distribution.

$$issue\_prob = \frac{\sum_{i \in intervals}(\#interval\_insts_i)}{\sum_{i \in intervals}(\frac{\#interval\_insts_i}{1.0(issue\ rate)} + stall\_cycles_i)} \tag{18}$$

Next, we explain how to estimate the number of non-overlapped instructions of an interval when a scheduling policy, round-robin or greedy-then-oldest, is used.

4.4.1.2 Modeling Round-Robin Policy

To model the non-overlapped instructions in the round-robin policy, we identify the instructions that are issued within several "waiting slots", where a "waiting slot" is the time period between scheduling two instructions from the representative warp during an interval. For example, in Figure 24(a), there are two waiting slots, one between WR-i1 and WR-i2 and another between WR-i2 and WR-i3. Eq. 19 shows the number of waiting slots of an interval i.

$$\#waiting\_slots_i = \#interval\_insts_i - 1, i \in intervals \tag{19}$$

Within a waiting slot, every remaining warp is scheduled because of the round-robin scheduling. Eq. 20 shows the expected number of instructions issued from all remaining

warps within all waiting slots of interval *i*, which is equal to the number of non-overlapped instructions of the interval.

$$\#nonoverlapped\_insts_i = issue\_prob \times (\#warps - 1) \times \#waiting\_slots_i \tag{20}$$

Finally, the CPI of multithreading can be calculated using Eq. 17 and 16.

4.4.1.3 Modeling Greedy-Then-Oldest Policy

To model the non-overlapped instructions in the greedy-then-oldest policy, we identify the instructions that are issued after the stall cycles of an interval are fully overlapped. To count the number of non-overlapped instructions of the interval i, we first need to count the total number of instructions issued before the representative warp is re-scheduled, as shown in Eq. 21. This number is estimated as the multiplication of the number of issued instructions for a remaining warp (*#avg_interval_insts*) and the number of warps issued before the representative warp is re-scheduled ($\#issue\_warps\_in\_stall_i$). Both terms are explained in Eq. 22 and Eq. 23.

$$\#issue\_insts\_in\_stall_i = avg\_interval\_insts \times \#issue\_warps\_in\_stall_i \tag{21}$$

The number of issued instructions of a remaining warp is the number of instructions in the interval that it is currently executing. Since this is unknown information, we instead estimate that value by taking the average number of instructions of an interval as the number of issued instructions of a remaining warp, as shown in Eq. 22.

$$avg\_interval\_insts = \sum_{i \in intervals} \frac{\#interval\_insts_i}{\#intervals} \tag{22}$$

Eq. 23 shows the number of warps that issues instructions during the stall cycles of interval i. The number of remaining warps (*#warps − 1*) is multiplied based on the assumption that during the stall cycles, every remaining warp will be scheduled since the "oldest" policy equalizes the probability of each warp being scheduled to prevent starvation.

$$\#issue\_warps\_in\_stall_i = issue\_prob\_in\_stall_i \times (\#warps - 1) \tag{23}$$

Eq. 24 shows the issue probability of a remaining warp during the stall cycles. Recall that we assume an uniform distribution of the issue probability.

$$issue\_prob\_in\_stall_i = min(issue\_prob \times stall\_cycles_i, 1) \tag{24}$$

Finally, Eq. 25 shows the number of non-overlapped instructions of interval i by subtracting the number of stall cycles from the total issued instructions (Eq. 21). The non-overlapped instructions are incurred if the number of issued instructions is more than the stall cycles.

$$\begin{aligned} \#nonoverlapped\_insts_i = max(\#issue\_insts\_in\_stall_i - \\ stall\_cycles_i \times 1.0(issue\ rate), 0) \end{aligned} \tag{25}$$

Similar to the round-robin policy, the CPI of multithreading can be calculated using Eq. 17 and 16.

4.4.2 Modeling Resource Contention

One of the most prominent cause of resource contentions in GPUs is *memory divergence* (a.k.a. uncoalesced memory accesses). We do not model the resource contention for normal operations by assuming that *in a balanced GPU design, the resources used for normal operations are sufficient for each warp.* For example, if the number of threads is 32 in a warp, then the floating-point units should be at least 32 (or 16 if a warp takes two cycles to issue) to make all threads progress equally.

To model the queuing delays caused by memory divergence, different scheduling policies may affect the queuing delays as the instruction orderings are different. However, we find that the instruction orderings matters only when the degree of contention is low. Otherwise, different instruction orderings cannot change the queuing delays much because the incurred queuing delays are far more than any instruction ordering can overlap. Thus, our resource contention models, described below, are applied to both round-robin and greedy-then-oldest policies.

Specifically, we model the queuing cycles caused by the contention for (1) MSHRs and (2) DRAM bandwidth. Similar to the multithreading model, we leverage the interval profile of the representative warp to predict the queuing delays. Eq. 26 shows the queuing delay calculated per-interval basis.

$$CPI_{rc\_contention} = \frac{\sum_{i \in intervals}(MSHR\_delay_i + Bandwidth\_delay_i)}{\sum_{i \in intervals} \#interval\_insts_i} \quad (26)$$

4.4.2.1 Modeling MSHRs

Figure 25 illustrates the queuing delays resulting from a limited number of MSHRs. Assume that the number of MSHR entries are 6 and all warps have the same interval profiles containing two compute instructions and one load instruction. The load instruction issues two memory requests and which both result in an L1 cache miss. The MSHR entries are saturated after three warps issue (W1-W3) the load instructions. Thus, W4 incurs queuing delays before its load instruction can be issued.

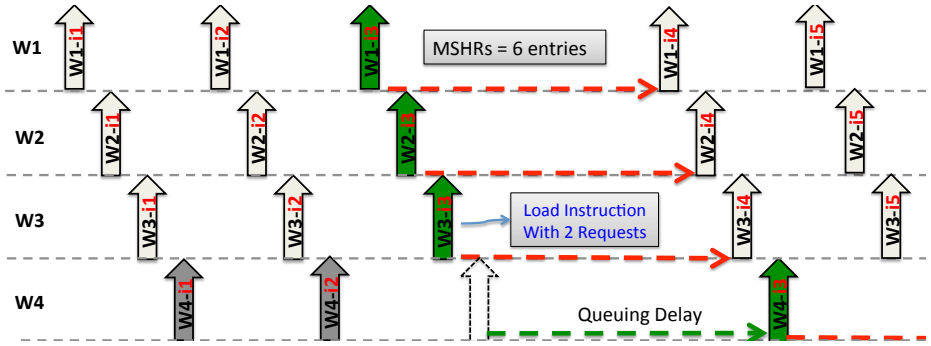


Figure 25: The queuing delays caused by a limited number of MSHR.

Since the queuing delay of a memory instruction varies depending on the number of the remaining warps issuing memory requests beforehand, we probabilistically estimate the queuing delay of every interval of the representative warp. First, we estimate the number of concurrent memory requests of a core in an interval ($\#core\_reqs_i$). Second, we calculate the expected queuing delay of a memory request ($exp\_queuing\_delay_i$). Finally, by multiplying the expected queuing delay with the number of memory instructions in an interval, we get

the queuing delay of the interval due to limited MSHR entries ($MSHR\_delay_i$). Details are as follows.

Eq. 27 shows the number of concurrent memory requests of a core in interval i. For each interval, we count the expected number of concurrent memory requests as the total requests issued from all warps in a core.

$$\#core\_reqs_i = \#warp\_mem\_reqs_i \times \#warps, i \in intervals \tag{27}$$

Eq. 28 shows the expected latency of a memory request in interval i considering a limited number of MSHRs. *avg_miss_latency* is the average L2/DRAM access latency of all memory instructions without MSHR contention. Since the distribution of the L1/L2 miss rates of a memory instruction can be calculated from the input collector (Section 4.5.2), we take the average L2/DRAM access latency across all memory instructions to get *avg_miss_latency*. Then, we estimate the latency of a memory request with index j in MSHRs as "$avg\_miss\_latency \times \left\lceil \frac{j}{\#MSHR} \right\rceil$". For example, in Figure 25, the latency of the memory requests from W1, W2 and W3 is *avg_miss_latency* while the latency of those from W4 is $avg\_miss\_latency \times 2$ since the memory requests from W1, W2 and W3 can be serviced concurrently while W4 has to wait until the MSHR entries are freed. By taking the average, we get the expected latency of a memory request. Finally, by subtracting *avg_miss_latency* from the expected miss latency, we get the expected queuing delay of a memory request caused by limited MSHRs.

$$exp\_queuing\_delay_i = \frac{\sum_{j=1}^{\#core\_reqs_i} avg\_miss\_latency \times \left\lceil \frac{j}{\#MSHR} \right\rceil}{\#core\_reqs_i} - avg\_miss\_latency \tag{28}$$

Eq. 29 shows the queuing delay of interval i in a core due to MSHR contention. The queuing delay only occurs when the expected number of memory requests exceeds the number of MSHRs. Note that $\#warp\_mem\_insts_i$ represents the number of memory instructions in interval i from a warp. We count the queuing delay *per-memory instruction* not

per-memory request since a divergent memory instruction has multiple memory requests issued concurrently, thereby overlapping the queuing delay.

$$MSHR\_delay_i = \begin{cases} 0, & \#core\_reqs_i \leq \#MSHR \\ exp\_queuing\_delay_i \times \#warp\_mem\_insts_i, & \#core\_reqs_i > \#MSHR \end{cases} \quad (29)$$

In Eq. 28, we assume that all warps issue the memory instructions of an interval at the same cycle. However, in reality, memory requests are issued at different cycles depending on how warps are scheduled. In that case, some warps may incur less queuing delay if they issue the memory instructions at a later time. As the difference of the progresses of different warps tend to be much smaller than the incurred stall cycles, the difference of queuing delay between warps is ignored.

The approach used to model queuing delays can be generalized to model other components with resource contention problems, such as the special functional unit (SFU). Applying the approach for these components is left for the future work.

4.4.2.2 Modeling DRAM Bandwidth

As a large number of memory requests from the warps are likely to be issued within a short amount of time, limited DRAM bandwidth is another major bottleneck in the memory system. However, the queuing delay model of MSHRs cannot be applied to the DRAM queue since the DRAM queue has a much shorter service time. For example, the service time of a memory request in MSHRs is the miss latency of the request while that in the DRAM queue is the transmission time of a cache line on the DRAM bus. In the case of short service times, estimating the arrival time between different requests is crucial since it can affect the queuing delay.

To approximate the arrival time of each request, we leverage M/D/1 queue, an approach used to model queuing delays in parallel simulations [66], as shown in Eq. 30. It states that the arrival rate follows a Poisson process and the service time is constant. The expected DRAM queuing delay of interval i composed of the utilization (ρ) and the arrival rate (λ).

However, for some intervals with a large number of memory requests, ρ could be equal to 1 resulting in unlimited queuing delay. To prevent this, we cap the queuing delay by assuming that a request arrives at the queue in which half of the maximum number of requests are ahead of it ($\frac{\#core\_reqs_i \times \#cores}{2}$).

$$Bandwidth\_delay_i = min(\frac{\lambda_i s^2}{2(1-\rho)}, s \times \frac{\#core\_reqs_i \times \#cores}{2}) \tag{30}$$

Eq. 31 shows the utilization ρ in which the service time s is represented in cycles as $freq_{core} \times \frac{L}{B}$ while the service time of the request in the DRAM queue is $\frac{L}{B}$ where L is the cache line size and B is the DRAM bandwidth.

$$\rho(utilization) = \lambda_i s = \lambda_i (freq_{core} \times \frac{L}{B}) \tag{31}$$

Eq. 32 shows the aggregated arrival rate from all cores. To estimate the arrival rate λ, we assume that the memory requests of a given interval from all warps can be issued within the total cycles of the interval ($\frac{\#interval\_insts_i}{1.0(issue\ rate)} + stall\_cycles_i$).

$$\lambda_i(arrival\_rate) = \frac{\#core\_reqs_i \times \#cores}{\frac{\#interval\_insts_i}{1.0(issue\ rate)} + stall\_cycles_i} \tag{32}$$

4.5 Input Collector

4.5.1 Overview

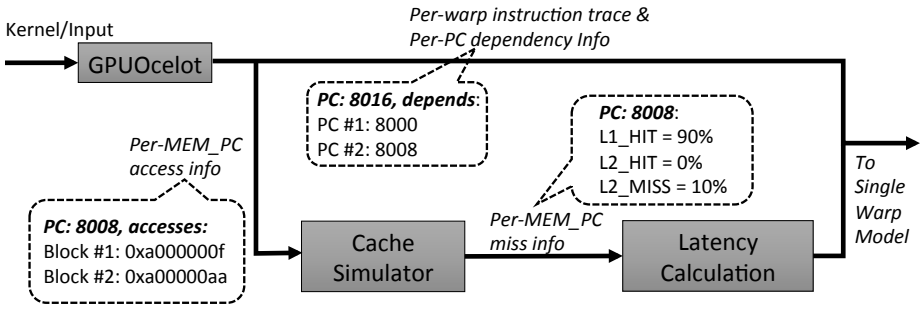


Figure 26: The input collector

Figure 26 shows the input collector of GPUMech. GPUOcelot [56] is used as a functional simulator, which executes a GPGPU kernel and generates *per-warp* instruction traces

with dependency information. The cache simulator reads the memory instructions and their addresses from the trace of each warp in a round-robin fashion to get the *per-memory PC*[1] miss rates. The cache simulator models a system with the number of warps and cores equal to that of the modeled system without timing information. Once the latency of a PC is determined (Section 4.5.2), both per-warp instruction trace and the per-PC latency are used as the inputs to the single-warp model.

4.5.2 Instruction Latency per PC

To determine the latency of per PC, we classify the PCs into *compute* and *memory* PCs. The latencies of compute PCs are fixed and based on the system configurations. On the other hand, the latencies of memory PCs are determined as follows.

First, we simulate L1/L2 caches to collect the *distribution of miss events* for every memory PC, e.g., (L1 hit: 0%, L2 hit: 10%, L2 miss: 90%). For a divergent memory instruction, it may have some cache hits at different levels of the memory hierarchy. In this case, the miss event of the memory instruction is determined by the memory request with the longest latency. Second, the distribution of miss events is collected by counting the miss events of every memory instruction across all warps. Lastly, the latency of a memory PC is equal to the *average memory access time* (AMAT) of the PC. For example, a memory PC hits L2 cache (120 cycles) for 90% of the executions and misses L2 cache (420 cycles) for 10% of the executions, the latency of the memory PC is equal to $150 = 0.9 \times 120 + 0.1 \times 420$ cycles.

In this work, we do not model front-end stall events, such as I-cache misses and synchronization overhead. Because warps share the same kernel (SPMD programming model), the I-cache miss rate is very low for all kernels. In most kernels, synchronization usually occurs occasionally within a thread block, e.g. by calling synchthreads(). Since the warps in a thread block are likely to make similar progress, the within-thread-block synchronization overhead is typically low. Synchronization across thread blocks is rarely used in

[1] We use the term PC to indicate a static instruction.

GPGPU kernels since the operation requires atomic instructions, which cause significant slowdown. Please note that control divergence is not considered as one of the stall events since it does not stall the entire warp (some threads in a warp still make progress).

4.6 Evaluation

4.6.1 Methodology

We use Macsim [67], a cycle-level simulator to validate GPUMech. The simulation configurations are listed in Table 7. The latency of the network-on-chip is included as part of the L2 latency. The validation is done by calculating the *relative error* of the CPI predicted by the evaluated model with that of detailed timing simulation.

To demonstrate the wide applicability of our model, we evaluated the kernels from Rodinia 2.1 [64], Parboil 2.5 suites [68] and NVIDIA SDK (40 kernels in total). The evaluated kernels have at least $3 \times$ *system_occupancy* thread blocks to have enough length of simulation.

Table 7: Simulation Configuration.

Number of cores	16
Front End	Fetch width: 1 warp-instruction/cycle, 4KB I-cache
Execution core	1.0 GHz, SIMT width: 32, Warp size: 32 threads, Maximum threads: 1024 threads, Issue width: 1 warp-instruction/cycle, Instruction latencies are modeled according to the CUDA manual (normal FP instructions are 25 cycles)
On-chip caches	16 KB software managed cache 32 KB L1 cache, 128B line, 25 cycles, 8-way assoc, 32 MSHR entries 768 KB L2 cache, 128B line, 120 cycles 8-way assoc
DRAM	DRAM Bandwidth: 192 GB/s Access latencies: 300 cycles

Table 8 shows the evaluated models. `Naive_Interval` is the same as Eq. 10 in Section 4.2.1. In addition to evaluating the relevance of modeling different components,

we compare our results with a Markov Chain that models the multithreaded performance in [40]. The Markov Chain model is discussed in Section 2.4.

Table 8: Evaluated models.

Evaluated Models	Descriptions
`Naive_Interval`	Optimistic overlap
`Markov_Chain`	Markov chain based model [40]
`MT`	Modeling multithreading (Section 4.4.1)
`MT_MSHR`	Modeling Multithreading + MSHR (Section 4.4.2.1)
`MT_MSHR_BAND` **(GPUMech)**	Modeling Multithreading + MSHR + DRAM Bandwidth (Section 4.4.2.2)

4.6.2 Model Accuracies of Different Scheduling Policies

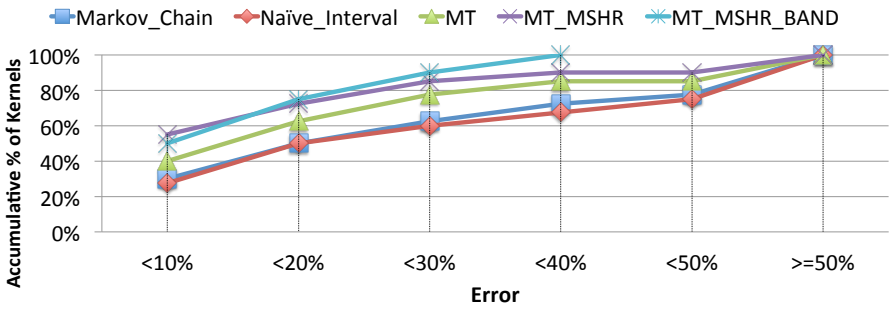


Figure 27: Model comparisons for round-robin policy

Figure 27 shows the comparisons between different models using the round-robin policy. `Markov_Chain` shows a slightly better result than the `Naive_Interval` since it models the multithreading effect more accurately. However, compared to `MT`, it still overestimates the performance. On the other hand, as `MT` does not account for the resource contention, some kernels with high memory divergence have high errors, such as *invert_mapping* from the *kmeans* benchmark that has a 330% error. `MT_MSHR` reduces the errors for most kernels down to less than 30%, except for the kernel *invert_mapping* from the *kmeans* benchmark. The reason why `MT_MSHR` is effective for most cases without modeling DRAM bandwidth is that the number of MSHR entries caps the number of read

requests, making the queuing delay of DRAM bandwidth bounded. However, since write requests do not allocate MSHR entries and *invert_mapping* has a large number of divergent write requests, the queuing delay of DRAM bandwidth is significant and has to be modeled. `MT_MSHR_BAND` (GPUMech) further reduces the error of *invert_mapping* from 180% to 35%. The remaining error is mainly caused by the inherit errors from modeling using queuing theory. Overall with our proposed mechanism GPUMech, 75% of the kernels have less than 20% errors, while only 50% of kernels have less than 20% errors with `Markov_Chain`. Furthermore, 32 warp case in Figure 29 (our baseline), shows that the average error of GPUMech is 13.2% while the `Markov_Chain` average error is 62.9%.

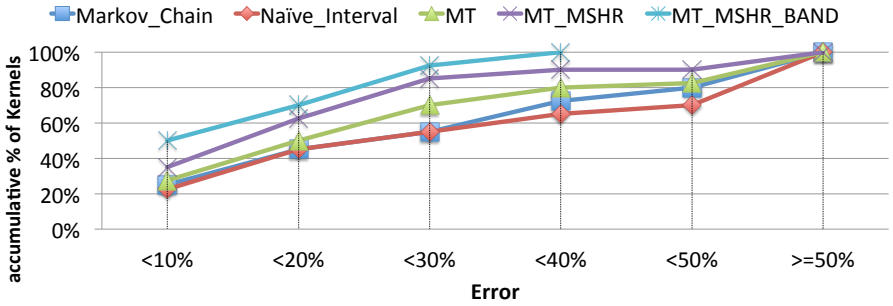


Figure 28: Model comparisons for greedy-then-oldest policy

Overall, `Markov_Chain` has two limitations, which are most likely the causes of high errors in our evaluation. First, the model assumes that threads are randomly interleaved without modeling any specific scheduling policy. Second, because the model assumes that a thread can issue no more than one memory request, it underestimates the queuing delays due to the resource contention in the memory system, especially for memory divergent kernels on GPUs.

Figure 28 shows the comparisons between different models using the greedy-then-oldest policy. The average error of GPUMech is 14.0% while the `Markov_Chain` error is 65.3%, showing a similar trend as modeling the round-robin policy. Overall, GPUMech models both scheduling policies with high accuracy.

4.6.3 Varying Hardware Configurations

In order to demonstrate the robustness of our model, we varied the number of warps per core, the number of MSHR entries and the DRAM bandwidth to see the accuracy impact of our models for multithreading, MSHR and DRAM bandwidth. The error in the Y-axis is the relative error *averaged over* all kernels. Since both round-robin and greedy-then-oldest policies show a similar trend, in the following figures, we report the results of modeling the round-robin policy only.

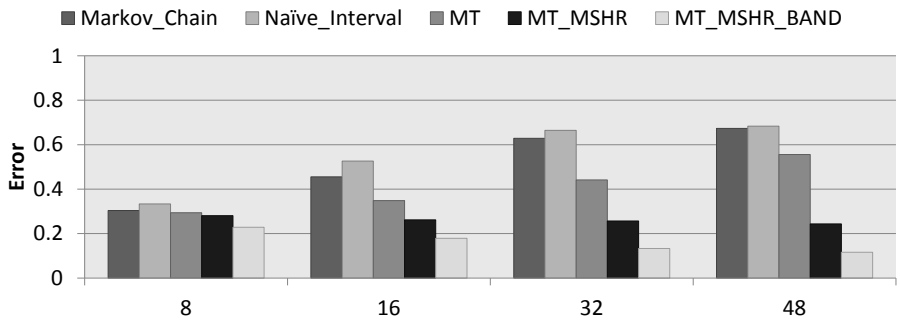


Figure 29: Errors with different number of warps per core

Figure 29 shows the errors with different number of warps. For the importance of resource contention modeling, except for `MT_MSHR` and `MT_MSHR_BAND`, the errors of other models increase with the number of warps per core since the delays caused by memory contentions get more severe with more warps. Although the MSHR is rarely congested when the number of warps is 8, some kernels still have significant queuing delays in DRAM bandwidth because of the write traffic, such as the two kernels from the *sad* benchmark. For the importance of multithreading modeling, when the number of warps is low, the prediction accuracies between `MT`, `Markov_Chain` and `Naive_Interval` are similar since the probability that instructions overlap with stall cycles is high. However, when the number of warps increases, the percentage of non-overlapped instructions increases. In this case, `MT` has a better accuracy than `Markov_Chain` and `Naive_Interval`. Overall, in GPUMech, the errors are higher in low number of warps per core, because it has more variations of multithreading. However, all other modeling techniques show higher errors

as the number of warps per core increases, which makes GPUMech more attractive.

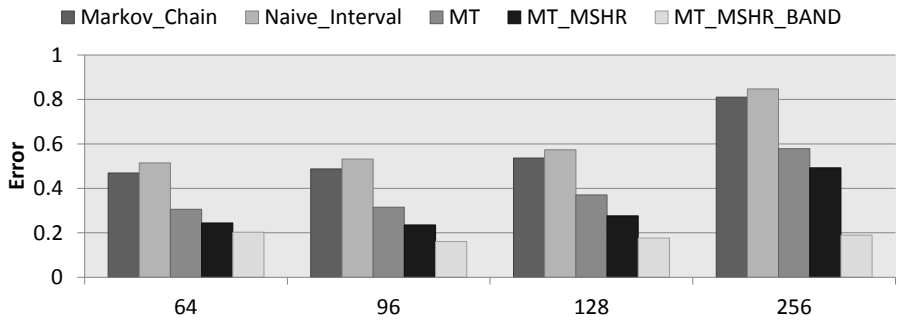


Figure 30: Errors with different number of MSHR entries

Figure 30 shows the errors with different number of MSHR entries. Increasing the number of MSHR entries increases the importance of modeling DRAM bandwidth. With more MSHR entries, the queuing delays in MSHR decrease, so the error differences between `MT` and `MT_MSHR` also decrease. However, more MSHR entries increase the queuing delays of DRAM since more on-the-fly memory requests need to be consumed. Only `MT_MSHR_BAND` captures these increases in DRAM congestion well since with all other modeling techniques, error increases as the number of MSHR entries increases.

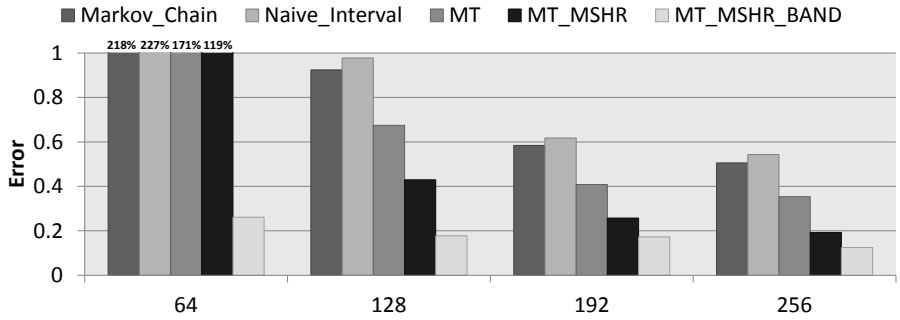


Figure 31: Errors with different DRAM bandwidth (the unit of X-axis is GB/s)

Figure 31 shows the errors with different DRAM bandwidth. DRAM bandwidth modeling is more important when the DRAM bandwidth is lower since lower DRAM bandwidth indicates higher DRAM queuing delays. The difference between modeling `MT_MSHR_BAND` and other models becomes smaller as the DRAM bandwidth increases. When the DRAM bandwidth is low, the errors are higher even with `MT_MSHR_BAND` indicating that it requires

a more accurate determination of DRAM queuing delays. For 64 GB/s, GPUMech shows 26.1% error, while for all other configurations the error was less than 17.8%.

4.6.4 Accuracy of Cache Simulation

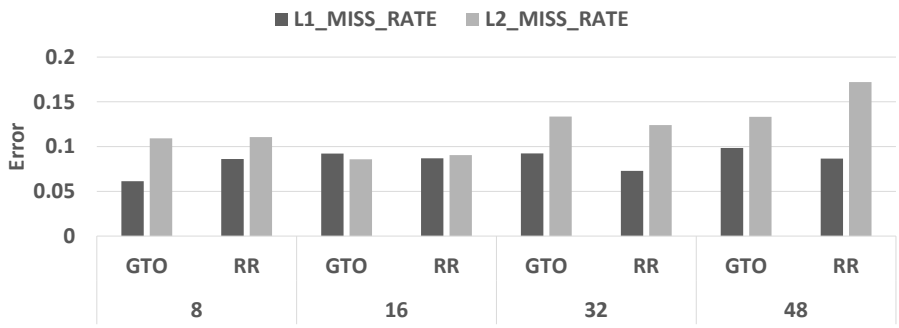


Figure 32: Error of L1/L2 cache miss rates compared against detailed simulations

Since the cache simulation is used for the input collector to get the cache miss rates per-PC, it is required to know the representativeness of the cache miss rates compared to those of detailed simulation. Figure 32 shows the errors of L1 and L2 cache miss rates of cache simulations compared to the detailed simulations. The L1/L2 cache miss rates below 5% are excluded since small differences can lead to high errors. For both RR and GTO policies, the memory instructions are fed into the cache simulator in the round-robin fashion. We made the following observations.

- Even though different scheduling polices may change the cache miss rates, the difference is negligible for our tested kernels since the errors between the polices are similar. The hit/miss rates are not affected much since each memory instruction has strong tendency on hit/miss rates depending on access types, e.g., streaming or indexed accesses.

- The major source of error is that detailed simulations have the "pending hits" which hit the MSHRs and is considered as miss while the cache simulation reports hits. To address the issue, random delays can be added to the miss latencies of L1 and L2 to capture more pending hits. The error of L1 miss rate is higher than that of L2

since a larger space (768 KB) and inter-core sharing of L2 make the prediction more challenging. On average, the errors of L1 miss rates are 8.3% and 8.6% while the errors of L2 miss rates are 12.4% and 11.5% for RR and GTO policies, respectively.

4.6.5 Discussions on Timing Overhead

In this section, we discuss the timing overhead of GPUMech compared to a detailed timing simulator.

GPUMech has three areas of overhead: *warp clustering*, *interval algorithm* and *cache simulation*. Since GPUOcelot is used for collecting traces for both detailed simulation and GPUMech, its overhead is excluded. First, the overhead of warp clustering could be significant depending on how many warps a kernel has. However, clustering only incurs a one-time cost in the per-input-basis. For the kernel with 100,000 warps, the clustering overhead is a few seconds. Second, the instruction algorithm is several hundred times faster than detailed simulations. In addition, the speedup can be further increased by running the interval algorithm of each warp in parallel, but we did not explore this option. Third, on average, the cache simulator is around 108x faster than our detailed simulator. Overall, GPUMech is 97x faster than detailed simulation.

To model a different hardware configuration, since the stall cycles may have changed, *cache simulation* and *the interval algorithm* of the representative warp need to be rerun. Because the representative warp is already selected, running interval algorithms on the remaining warps and warp clustering are not needed as these tasks are applied in the per-input-basis and remain unchanged across different hardware configurations. Overall, the speedup would be higher when exploring different hardware configurations since the profiling cost can be further reduced by excluding the cost of warp clustering and the running of the interval algorithms on the remaining warps.

4.7 Use Case

An advantage of GPUMech is the ability to construct CPI stacks which are used to visualize the performance bottlenecks and their relative impact. Table 9 shows the total categories in which cycles are spent.[2] We construct the CPI stack of a kernel as follows.

- We construct the CPI stack of the representative warp. To determine the CPI category (excluding `BASE`[3]), we check the reason for stalling. If the stall cycles are caused by a dependence on a compute instruction, we add the cycles to `DEP` category. If the stall cycles are caused by a memory instruction, we use the distribution of miss events to categorize the cycles. For example, if the stall cycles are 100 while the distribution of miss events is L2 hit: 10% and L2 miss: 90%, we then add 10 cycles to `L2` category while we add 90 cycles to `DRAM` category assuming no MSHR and DRAM queuing delays. By dividing each category with the number of instructions, we get the CPI stack of the representative warp. After the category of an interval is determined, we add the stall cycles of the interval to the category.

- Based on the performance improvement of multithreading, we shrink down each category of the CPI stack of the representative warp by the factor of $\frac{CPI_{multithreading}}{CPI_{repres\_warp}}$. By doing this, the relative importance of each category is reserved while modeling the performance under multithreading.

- To model the resource contention, we create two new categories: `MSHR` and `QUEUE`. The modeled queuing delays of MSHR and DRAM bandwidth are normalized by the number of instructions before being added to `MSHR` and `QUEUE` categories.

Table 9: Stall types of CPI stacks.

Stall types	Abbreviations
Instruction issue cycles	BASE
Compute Dependencies	DEP
L1 Hits	L1
L2 Hits	L2
DRAM access latency	DRAM
MSHR queuing delay	MSHR
DRAM queuing delay	QUEUE

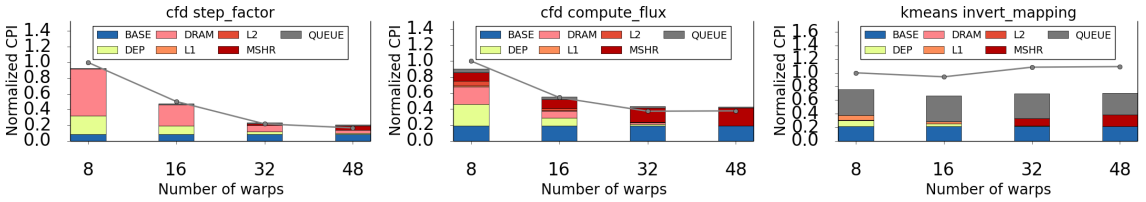


Figure 33: The CPI stacks of cfd_step_factor, cfd_compute_flux and kmeans_invert_mapping kernels.

4.7.1 Identify the Scaling Bottlenecks

In this application, we leverage the CPI stack to visualize the performance bottlenecks with varying the number of warps on a core. Increasing the number of warps may reduce the cycles spent in DEP because the computation takes fixed cycles while more warps can reduce the stall cycles caused by computation dependencies. However, the cycles spent in L1 and L2 depend on the miss rates where more warps may cause higher miss rates due to cache pollution. On the other hand, the cycles in MSHR and QUEUE may increase since more warps compete for those shared resources. GPUMech is able to visualize performance bottlenecks using the CPI stack and then find the configuration that has the best performance.

We select three kernels with distinct memory divergence degrees from Rodinia suite [69].

[2]Note that the DRAM access latency is the base DRAM access latency (300 cycles in our case) without queuing delays

[3]BASE category is the cycles used to issue an instruction, which is a constant depending on the system configuration.

cfd_step_factor kernel is a coalesced kernel with no divergent accesses. cfd_compute_flux has medium range of divergence since some memory instructions have up to 16 diverged requests. kmeans_invert_mapping has maximum memory divergence with up to 32 diverged requests per memory instruction.

Figure 33 illustrates the CPI stacks, shown in bars. The lines show the *oracle CPI* measured using detailed simulations. All CPIs are normalized by the oracle CPI of the configuration with 8 warps. As seen in the figure, GPUMech accurately predicts the scaling trend. In the following, we explain the performance limiting factors for each kernel.

- **cfd_step_factor**: Because all memory accesses are almost coalesced, the cycle overhead from `MSHR` and `QUEUE` are negligible in all configurations except for the 48 warps configuration, in which case larger MSHR queuing delays are incurred. Even though the major performance bottleneck is DRAM accesses, cfd_step_factor has a good scaling capability since the accesses are well spread into different intervals without congesting the memory system. In addition, we can see that cfd_step_factor has negligible or no cache locality since no cycles are seen in the `L1` or `L2` categories.

- **cfd_compute_flux**: In the configuration of 8 warps, `DRAM` and `DEP` have a similar amount of cycle overhead. The L2 cache is effective for the diverged accesses while L1 is not very useful since the working set is much larger than the L1 size. In terms of memory congestion, `MSHR` contributes to 10.8% of the cycle overhead while `QUEUE` contributes to 3.8% of the cycle overhead, thanks to the high degree of memory divergences. Since `MSHR` and `QUEUE` are not the majority of the cycle overhead, increasing the number of warps improves performance, as shown from the configurations of 8 to 16 warps. The predicted performance saturation point occurs in the configuration of 32 warps in which `MSHR` dominates the cycle overhead.

- **kmeans_invert_mapping**: In the configuration of 8 warps, since the load instructions have a high L1 hit rate (90.5%), the cycles spent in the L1 cache is significant. The

MSHR is negligible even though the divergence degree is high thanks to the high L1 hit rate preventing most accesses from occupying MSHR entries. On the other hand, it is a bit counter-intuitive to see that DRAM has negligible cycle overhead while QUEUE is high. The reason is that while the store instruction is not on the critical path and does not increase the cycles in DRAM category, the diverged write accesses still consume DRAM bandwidth and increase the queuing delays of the load instruction.

Because the performance bottlenecks could come from multiple sources, without CPI stacks, it is hard to tell what limits the performance of a given hardware configuration. By showing the relative importance of performance bottlenecks, the CPI stack is useful for software developers to apply the corresponding optimizations and for hardware developers to scale the required hardware resources, such as MSHR entries, DRAM bandwidth, to achieve the optimal performance. By leveraging the proposed model, not only can we find the performance saturation point, but also the details of performance bottlenecks.

4.8 Summary

In this work, we proposed GPUMech, which is the first interval analysis for GPU architectures. GPUMech models multithreading and resource contentions in MSHR and DRAM bandwidth due to memory divergence. To reduce the errors from control-divergent warps, we employ a clustering algorithm to identify representative warps. Overall, GPUMech is about 97x faster than a detailed timing simulator and on average, it only has 13.2% error for modeling round-robin scheduling policy and 14.0% error for modeling greedy-then-oldest policy. In addition, GPUMech generates CPI stacks, which helps the hardware/software developers to visualize performance bottlenecks.

The contributions of this chapter are as follows.

- We propose the first interval analysis technique for GPU architectures, which can be also applied to other mulitithreaded architectures as well.

- We improve the interval analysis technique by modeling multithreading (Section 4.4.1) and resource contention of memory system. (Section 4.4.2).
- By leveraging our proposed technique, we propose the first CPI-stack visualization tool for GPU architectures to provide insights into performance bottlenecks. (Section 4.7).

The sampling approach in Chapter 3 and the abstracted sampling approach in this chapter can be complimentary to each other to further reduce the simulation time. For example, after selecting the simulation points are selected using TBPoint, for each simulation point, GPUmech can be applied to explore the performance under different hardware configurations.

CHAPTER 5

SIMPROF: A SAMPLING FRAMEWORK FOR DATA ANALYTIC WORKLOADS

Chapter 3 and 4 introduce the approaches for reducing the simulation time of GPGPU kernels. This chapter focuses on the simulation time problem of another type of emerging workloads, data analytic workloads, which are prevalent in today's data center. In addition to massive thread-level parallelism, data analytic workloads also feature huge data volume and IO activity. This chapter presents a sampling framework, SimProf, which selects the simulation points through the execution behaviors on a real machine including call stacks and performance counters. It also includes an effective input-sensitivity test to reduce the number of simulation points when exploring multiple inputs.

5.1 Introduction

Data analytic workloads have growing importance as data volume increases rapidly. The enterprise can gain valuable insights using data analytics. For example, using the social media analysis, companies can adjust their pricing and promotion on the fly for optimal results. To process the huge data volume, data analytic workloads are typically built on top of a computing framework, e.g., Apache Hadoop [70] or Apache Spark [2] to scale out to multiple nodes. In addition, the frameworks handle how the workloads are scheduled onto multiple nodes and provides reliability to tolerate node failures.

In recent years, the computation of data analytic workloads has been moving toward in-memory computing, which keeps the data to be processed in the memory as much as possible for better performance. The major factor is the drop in DRAM prices every year and that the commodity hardware can support tera-bytes of main memory. The computing frameworks, such as Apache Spark, use the directed acyclic graph (DAG) execution

model to leverage the cheap and abundant main memory to achieve 100x faster performance than Hadoop [2], which requires frequent disk accesses. A recent study indicated that data analytic workloads become more CPU-bound than IO-bound [71]. For Spark SQL workloads, assuming a zero cost on disk IO only improves the job completion time by at most 19%. Therefore, it requires immediate attention to improve the performance of data analytic workloads from the architecture side.

Architectural simulation is the most common approach for architecture designs. However, because of a slow simulation speed, it is not possible to simulate entire workloads. Therefore, various approaches were proposed to sample the instructions to be simulated [3, 1]. For server workloads, so far mostly transaction-based workloads are assumed, e.g., SpecWeb [72], which have a large number of short and atomic transactions. To simulate this server workload, the common practice is to simulate a one time interval that covers a sufficient number of transactions, e.g., 10 seconds [28, 73, 74]. However, simulating a single interval is not suitable for data analytic workloads for the following reasons.

From the accuracy perspective, a single interval cannot represent the behavior of the entire data analytic workloads. The data analytic workloads typically have several stages, each of which has multiple tasks being executed. Furthermore, the task length of a data analytic workload can be much longer than transaction-based workloads because more complex queries are processed. The task length ranges from a few hundred milliseconds to tens of seconds.

From the efficiency perspective, even a 10-second interval demands a long simulation time e.g., 20 days for simulating 10 seconds of a 64-core workload [75]. Other optimization techniques, such as checkpointing and parallel simulations, provide additional simulation time reductions, but the space overhead and the simulation complexity become additional challenges to overcome.

In this chapter, we propose SimProf, which is the first sampling framework to generate simulation points for data analytic workloads based on phase analysis. It has the following

key features. First, it runs natively on a real machine for fast profiling speed. It uses the standard profiling interfaces, such as Java Virtual Machine Tools Interface (JVMTI) [76] and perf_event kernel API, which can be easily integrated into any functional simulator. Second, it identifies the phases using call stacks and applies statistical sampling methods to select the final simulation points, which can sample the phases that do not have homogeneous performances. Third, it detects input-insensitive phases, whose performance does not change by inputs. When simulating multiple inputs with different data volumes, the simulation time can be further reduced by skipping input-insensitive phases. Fourth, the simulation points contains method-level information for the architects to analyze program behaviors more easily. Finally, the validation is done against the performance of a real machine rather than a simulator to demonstrate the robustness of SimProf.

5.2 Background and Motivation

It is essential to understand the execution model of the computing framework, which determines the phase behaviors of data analytic workloads.

5.2.1 Apache Spark

Apache Spark is a computing framework for distributed computing. It leverages the in-memory computation to provide performance up to 100 times faster than Hadoop for some applications. Several libraries, such as Spark SQL [77] and GraphX [78] are built on top of the core functionality in Spark to support different types of workloads. In the following, we explain how a data analytic workload is executed and how the phases are formed.

Figure 34 shows how the benchmark `wordcount` is implemented and executed in Spark. Figure 34(a) shows the source code of `wordcount` written in Scala [79]. Line 1 is to read the files from Hadoop Distributed File System (HDFS) [70], which is a distributed file system, and to put all lines into `lines`, which is a Resilient Distributed Datasets (RDD). An RDD is an immutable collection of data elements. Multiple operations can be performed on an RDD, such as map and reduce operations. After applying an operation on

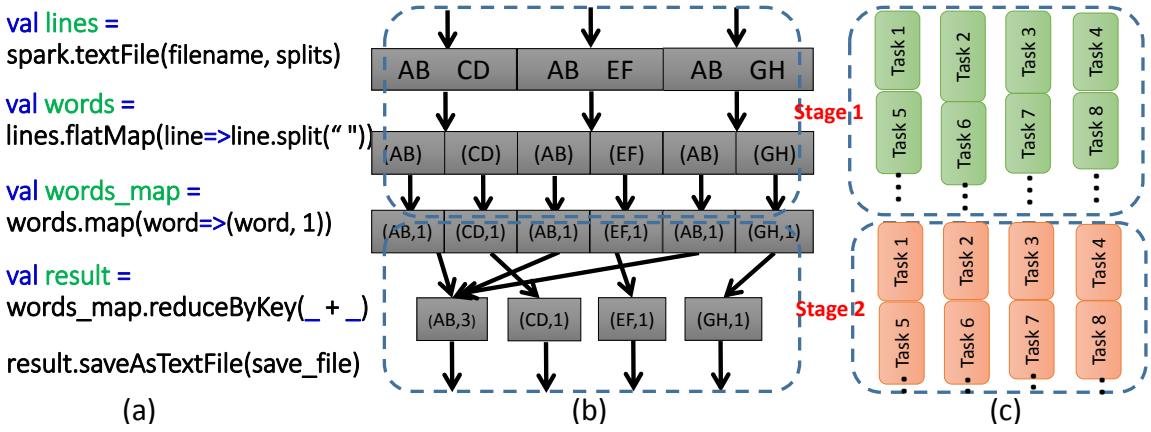


Figure 34: WordCount Example

every element in an RDD, the original RDD is transformed to a new RDD. For example, each line between lines 2 and 4 applies an operation and each operation results in a new RDD (`words`, `words_map` and `result`). Finally, the final RDD `result` is saved to a text file.

Figure 34(b) shows the execution flow. For the first three operations, the processing of a data element in an RDD can be done in parallel since it does not rely on the processing results of the other data elements. For the fourth operation (`reduceByKey`), the processing of a data element in `words_map` also involves the data elements with the same key. The fourth operation has to wait until the first three operations complete before it can proceed. Therefore, the first three operations form the first stage, while the other two operations form the second stage. In each stage, multiple tasks are spawned, as shown in Figure 34(c). Each task processes a "subset" of an RDD and applies multiple operations, e.g., three operations in stage 1. If stages have dependencies, as in this example, the tasks in different stages cannot be executed concurrently although the tasks within the same stage can be executed concurrently.

Figure 35 shows how the Spark workload is mapped into the distributed environment. First, a user submits the workload as a Spark job through the Spark driver to the cluster master. Then the cluster master analyzes the job and divides the operations into stages. For each stage, the master spawns tasks to multiple computing nodes, each of which runs a

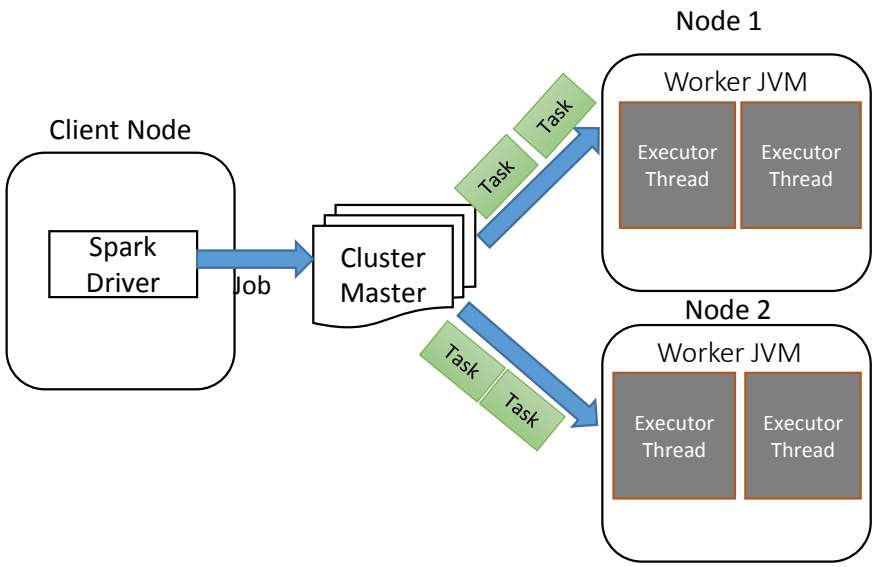


Figure 35: Spark Infrastructure [2]

Java virtual machine (JVM). Each task is processed by an executor thread within the JVM. Typically the number of executor threads is equal to the number of cores in the node to fully utilize the available computation power.

5.2.2 Phase Behaviors

We define "phase" as a set of sampling units that execute similar code. A sampling unit is a fixed number (e.g., 100 million instructions) of instruction intervals within a thread. Non-contiguous sampling units could belong to the same phase.[1]

We categorize the phase behaviors into intra-stage and inter-stage.

- **intra-stage**: multiple operations are executed within each task. Depending on the duration of each operation, several phases could be formed. One example is that the data processing and data IO can be in different phases. Initially, a task may be busy processing the data and generate the outputs to a memory buffer. When the memory buffer is close to full, it flushes the data to the disk to be used by the next stage. In addition, the framework operations used for data movement between nodes, e.g.,

[1] Essentially phase is the same as cluster in this chapter, but we use the term phase because it is commonly used in program profiling contexts.

data shuffling, also occur within the stage.

- **inter-stage**: The tasks in different stages may belong to different phases since they could execute different operations, e.g., map and reduce operations.

Phase analysis is a commonly used technique for selecting simulation points. The idea is to group the sampling units that have similar code executed into a phase. For each phase, a simulation point is selected to represent the behavior of the phase. However, the commonly-used tool of selecting simulation points, e.g., SimPoint [1], is not suitable for data analytic workloads for the following reasons.

First, most data analytic workloads are written in high-level languages, such as Java, Python or Scala, which use managed runtime to achieve platform independence. However, SimPoint collects the basic block counts as the code signature, while collecting the basic blocks for Java applications is not trivial and does not provide method-level information. The method-level information can provide useful insights for the architects to understand the architectural behaviors. Furthermore, collecting the basic block counts incurs high profiling overhead, which incurs 100% to 400% slowdown [80]. Finally, although porting the workloads to other languages, such as C++, is possible, it requires nontrivial porting effort and whether the behaviors are similar to the Java-based implementation is unknown.

Second, using the code signature solely to classify program behaviors is insufficient. We corroborate with the previous study [81] that the performance of server workloads could be dominated by other factors, such as the *data access behaviors*, while the similarly executed code may have heterogeneous performance because of different LLC miss rates. Thus, without being aware of the data-sensitive behavior within a phase, a single simulation point is insufficient to represent the performances of other points of the phase.

Third, the data diversity, which can be characterized in "4V," is significant for the data analytic workloads. Previous tools did not consider the input data impact on performance since only a few inputs exist for the conventional benchmark suites, such as SPEC CPU [82] and Parsec [8]. However, for the data analytic workloads, the data synthesizer is usually

used to synthesize data sets with different volumes based on the real-world seed inputs while retaining the characteristics of the inputs. Thus, it is important to understand the performance of which phases change by inputs. We will show that the number of simulation points can be trimmed down significantly by simulating only the input-sensitive phases.

Our goal is to sample only one executor thread based on its phase behaviors. However, since different computing frameworks may adopt different programming models, the design of the sampling framework should be general enough that it can be applied to any computing frameworks. For example, in Apache Hadoop [70], users define the map and reduce operations applied on the data while in Spark, users can define the types of data collections (RDD types) as well as the operations, which are not limited to map and reduce operations, e.g., union operation.

5.3 SimProf

Figure 36 is an overview of the SimProf framework. *Thread Profiling* profiles every executor thread within a JVM to retrieve the call stacks and the hardware counter events. *Phase Formation* vectorizes the call stacks and clusters them into phases. *Phase Sampling* leverages the statistical sampling approach, stratified random sampling, to select the simulation points based on phases. *Input-Sensitivity test* uses the mean and variance of the IPC of each phase to detect input-sensitive phases, which might have architectural behaviors sensitive to the inputs.

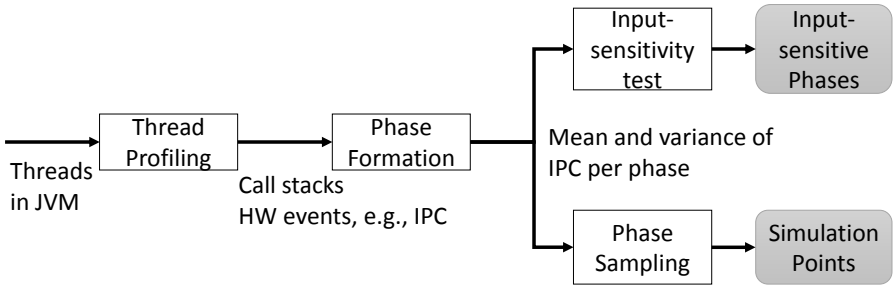


Figure 36: SimProf Overview

5.3.1 Thread Profiling

To detect the code executed in a sampling unit, the call stacks within the unit are collected. A call stack is a record of the active stack frames at a certain point in time during the program execution. In addition, the hardware counter events, such as IPCs and cache miss rates, are collected for validation and sampling purposes. To avoid a high profiling overhead, we take snapshots of the call stacks in a sampling unit periodically, as shown in Figure 37.

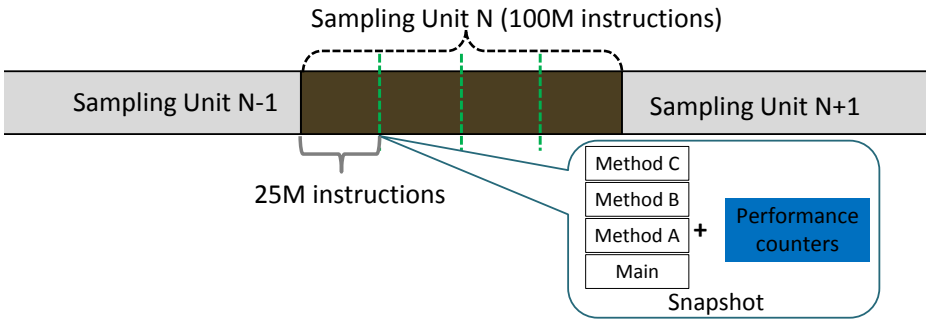


Figure 37: Snapshots in a sampling unit

Figure 38 shows the infrastructure for thread profiling. The call stack collector retrieves the call stacks of JVM through the JVM tool interface (JVMTI), which is the standard profiling interface available in all Java implementations. The hardware counter collector retrieves the hardware counter values through the `perf_event` interface provided by the Linux kernel. Both counters are controlled by the sampling manager, which controls the sampling rate, e.g., the frequency of a snapshot. It is also responsible for flushing the outputs of the collectors to the files since the collectors initially output to the memory for fast collection speed.

The profiling infrastructure is attached to each JVM, while SimProf profiles per-executor-thread basis in the JVM. Depending on the execution model of the profiling target computing framework, the lifetime of an executor thread can be different. In Spark, the lifetime of an executor thread is equal to the total time of a job. Profiling only one executor thread can cover different stages. However, in Hadoop, the lifetime of an executor thread is equal to

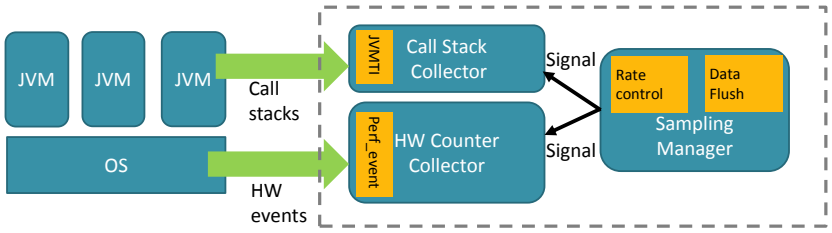


Figure 38: Thread profiler in SimProf

the lifetime of a task that it executes. So, the executor thread dies when its task is finished. In this case, the profiler merges profiled results from executor threads running on the same core to mimic a long-running executor thread in Spark.

Our design leverages the standard profiling interfaces so that it is applicable to any Java-based computing framework, not limited to Spark or Hadoop. The sampling unit size and the frequency of a snapshot can be tuned based on the users' needs. Empirically, we use a large sampling unit size, 100M instructions to avoid the simulation start-up effect, e.g., cold cache. The frequency of a snapshot affects the profiling overhead. The high frequency can slow down the application and skew the results of hardware counters, while low frequency may miss important call stacks executed in a sampling unit. We take snapshots every 10M instructions to balance accuracy and efficiency.

5.3.2 Phase Formation

Phase formation groups the sampling units into a phase if they have similar call stacks. It contains two steps. First, it converts the call stacks in one sampling unit into a feature vector. Second, it clusters the feature vectors into phases.

Figure 39 shows an example call stack that represents the IO routine commonly seen in Spark. From levels 1 to 4, it represents the starting methods of an executor thread. Then it is followed by the task routine, which indicates that a map task is currently executed. Finally, several IO-related methods, such as object serialization and disk writing, are called. Note that the figure only shows one call stack while one sampling unit typically contains multiple call stacks. All methods appearing in the call stacks in one sampling unit need to

be counted and converted into a feature vector. Each dimension of the vector represents a method while the value of the dimension is the frequency of the method appearing in the sampling unit. In addition, the number of dimensions in a feature vector is equal to the number of unique methods in the entire job execution so that all feature vectors have the same number of dimensions.

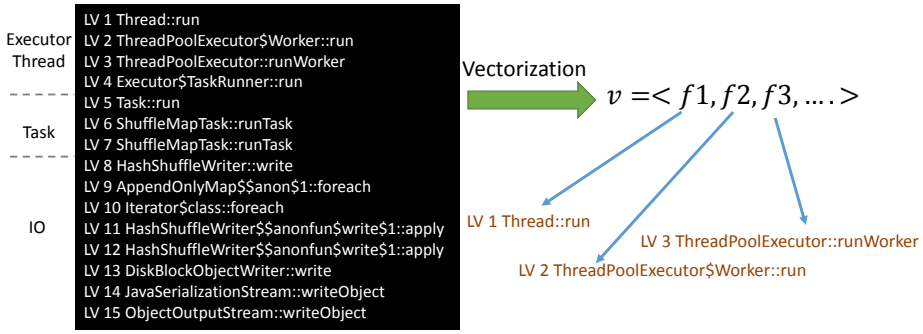


Figure 39: Convert call stacks into a feature vector

However, this type of feature vector has the following problems. First, a feature vector can easily have thousands of dimensions because a large number of methods are called, which often results in a high clustering time. Second, the high dimensions often have challenges in identifying the hot functions that have the most impact on performance.

To select the important methods (features), we use a linear regression to identify the methods that are highly correlated with performance. The univariate linear regression test [83] selects the top K methods that are highly correlated with the profiled IPCs. We set K as 100 to balance the clustering speed while still capturing important methods. For example, in Figure 39, the starting methods of the executor thread and the tasks are eliminated in the feature vector since those methods are common in most feature vectors and have no significant impact on performance.

We use the k-means clustering algorithm to group the sampling units into clusters (phases). To determine the number of phases k, we score the fitness of each k between 1 and 20 using the silhouette coefficient. Then, we choose the smallest k, which has at least 90% of the highest score among all k.

5.3.2.1 Phase Homogeneity Analysis

To understand how much the performance is similar in each phase, we calculate the coefficient of variation (CoV) of CPIs. The CoV is a good metric to indicate how homogeneous (i.e. how much the performance is uniform in a phase) the collection of sampling units is. A higher CoV means that the sampling units have a higher CPI variation.

Figure 40 show the population/weighted/maximum CoV of CPIs. We calculate the weighted CoV, which is the CoV of each phase weighted by the number of sampling units in the corresponding phase. The maximum CoV is the CoV of the phase that has the highest CoV. The population CoV is the CoV of all sampling units. The purpose is to demonstrate how well the phase formation performs. In an ideal case, the phase formation results in a low performance variation of each phase. In this case, the weighted CoV is low even if the population CoV is high. By contrast, if the phase formation is useless, it cannot separate high-performance phases from low-performance phases. In this case, the weighted CoV remains high when the population CoV is high. The figure shows that the weighted CoVs from all benchmarks are always lower than the population CoVs, so we can safely conclude that our phase formation separates phases with distinct performances. However, phase formation does not necessarily construct all phases to be homogeneous, as we see from the maximum CoV. We summarize the reasons for non-homogeneous phase behaviors as follows.

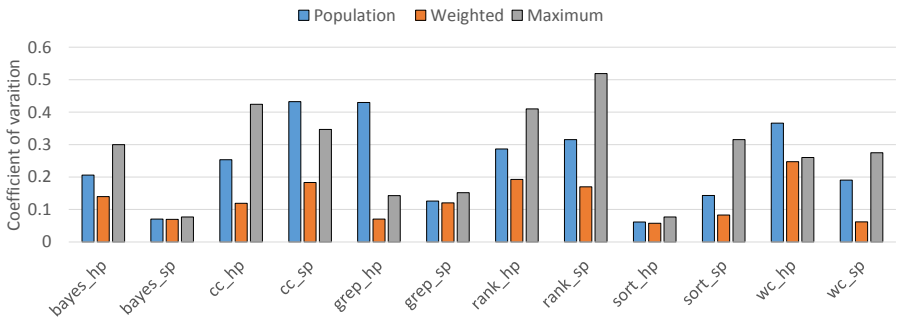


Figure 40: Coefficient of variation of CPIs

- **Data access pattern:** For key-value applications, if the reduce stage exists, it involves the sorting algorithm, which sorts the key-value pairs by the key since the reducer needs to process the data in the *per-key basis*. The common sorting algorithm used is quicksort, which sorts the keys recursively from small to large partitions. Although the sampling units execute the same sorting algorithm, some of them have a lower cache miss rate because of sorting the small partitions while others have a higher miss rate because of sorting the large partitions. For example, `wc_hp` spends a significant amount of time sorting keys (words) before the count of each word can be calculated. Thus, the non-homogeneous sorting phase results in a high weighted CoV.

  Another type of changing data access pattern is the reduce operation, which combines the values of the same key. For all key-value pairs, the values with the same key may not be next to each other, leading to random accesses. Thus, for sort and reduce operations, the LLC cache miss rates could affect the overall IPC.

- **OS scheduling:** we observe that sometimes the executor thread can be scheduled to other cores by OS. For the sampling units that involve the newly scheduled threads, higher D-cache misses could occur, leading to a higher CPI.

- **Phase Interleaving:** The phase behaviors of executor threads can be interleaved in a random order depending on how tasks are scheduled within threads. Even though the same phase behaviors are observed in two different sampling units, they may have diverse performances since they are interleaved with different phases of other executor threads.

- **Executed code difference:** to make a phase have homogeneous behaviors, the sampling units in a phase need to have nearly identical call stacks to ensure that they have similar behaviors. However, in some cases, since the large size of code is executed, a phase could have sampling units that have different code, leading to performance

differences.

Because non-homogeneity can occur within a phase, instead of assuming that all phases have homogeneous behaviors and selecting one sampling unit for each phase, we need a reliable way to select a set of simulation points within a phase to better represent the behaviors of the entire phase.

5.3.3 Phase Sampling

Phase sampling selects a number of sampling units within each phase as the final simulation points. To deal with the performance variation within a phase, we use the optimal allocation [84] to determine the sample size of each phase and use simple random sampling to select sampling units for each phase. This is a statistical sampling approach, known as stratified random sampling [84]. We refer to the "sample" as the set of selected sampling units (simulation points) while the "sample size" is the number of these units.

The idea of optimal allocation is to have a larger sample in the phase that has a higher IPC variance. Since the hardware counter events of every sampling unit, such as IPC, can be obtained from the frontend data collector (Section 5.3.1), the variance of each phase can also be measured.

Eq. 33 shows the sample size of a phase h determined by the optimal allocation. n is the overall sample size. For each phase h, N_h is the total number of sampling units, n_h is the sample size and σ_h is the standard deviation of phase h. The sample within each phase is randomly selected. We call the selected sampling units the final "simulation points". We use the sampling unit ID to represent each simulation point. On the other hand, the cluster center of each phase is also saved, which will be used for the input sensitivity test.

$$n_h = \frac{n \times (N_h \times \sigma_h)}{\sum_{i \in phases}(N_i \times \sigma_i)},\ h \in phases \tag{33}$$

The sampling error of the simulation points is represented as the confidence interval, which can be calculated as follows. Eq. 34 shows how the confidence interval is computed.

Eq. 35 shows the margin of error defined by the confidence level $(1-\alpha)$ and the standard error (SE). Users can specify the required confidence level, while the standard error depends on the sampling technique. We assume the distribution of the means (the average CPI of the simulation points of a phase ($\overline{CPI_h}$)) is normal because of the central limit theorem. In this case, α can be expressed in z-score form.

$$CI = sample\_mean \pm margin\_of\_error \tag{34}$$

$$margin\_of\_error = \alpha \times SE \tag{35}$$

Eq. 36 shows the standard error (SE) of stratified sampling. Eq. 37 shows s_h, which is the standard deviation of the phase. We use the collected CPIs of each sampling unit to get s_h.

$$SE = \frac{1}{N} * \sqrt{\sum_{h \in phases} [N_h^2 \times (1 - \frac{n_h}{N_h}) \times \frac{s_h^2}{n_h}]} \tag{36}$$

$$s_h = \sqrt{\frac{1}{n_h - 1} \sum_{i=1}^{n_h} (CPI_i - \overline{CPI_h})} \tag{37}$$

Based on the above analysis, users can choose the sample size based on the simulation budget and the target sampling errors. First, users choose the sample size n that fits in their simulation time budget. Second, SimProf picks the simulation points based on the sample size. Third, users simulate the simulation points and estimate the sampling error. If the error is higher than their requirement, they can increase the sample size and repeat the procedure until the sampling error is acceptable.

Since SimProf uses a large number of sampling units, the simulation time can still be significant; users can combine other sampling approaches, e.g., systematic sampling [3] to reduce the simulation time of each simulation point.

5.3.4 Input Sensitivity Test

Algorithm 5.1 shows the procedure for the input sensitivity test of reference inputs. Initially, one input is assigned as the training input and the rest of the inputs become the reference inputs. For each reference input, the unit classification classifies the sampling units of the reference input to phases. For each phase, we perform the input sensitivity test of the corresponding phase by comparing their performances and determine whether the performance of the phase changes by inputs.

```
# Assign one input as the base input, and the others are reference inputs
assign_inputs()
for inp in reference_inputs:
    # Classify sampling units into phases
    total_phases = phase_classification()

    for phase in total_phases:
        # Check if the phase is input sensitive
        if phase not in input_sensitive_phases and phase_sensitivity_test(phase) is True:
            input_sensitive_phases.add(phase)
```

Algorithm 5.1: Input sensitivity Test

5.3.4.1 Unit Classification

Unit classification classifies the sampling units from a reference input into phases using the cluster (phase) centers of the training input. Even though the program takes different inputs, similar methods are still executed resulting in the same set of phases. However, because of different length of phases and/or input-sensitive phases, the performance of the program varies by inputs.

The procedure for classifying a sampling unit is as follows.

- A sampling unit is vectorized into a feature vector based on the profiled call stacks. (Section 5.3.2)
- The feature vector is classified into the phase where the center of the phase has the

minimum distance to the feature vector.

Unit classification ensures that sampling units that execute similar program methods belong to the same phase between the training and the reference inputs. In this case, we can assume that each phase is executing similar code between both inputs so that we can use the input sensitivity test to see whether the performance of the phase would change by inputs.

5.3.4.2 Phase Sensitivity Test

The mean and standard deviation tests are used for the phase sensitivity test. For each phase, we collect the mean and standard deviation of the sampling units within the phase. Then, we compare mean and standard deviation between those of reference inputs and those of train inputs respectively. If any of them are larger than 10%, then the phase is considered *input sensitive*, as shown in Eq. 38. For a given phase, if all the reference inputs do not pass the input sensitivity test, then the phase is considered *input insensitive*. When exploring the micro-architectural behaviors of multiple inputs, one can skip the input-insensitive phases while focusing on simulating the input-sensitive phases.

$$if(|\frac{\mu_{train} - \mu_{ref_i}}{\mu_{train}}| > 10\% \; or \; |\frac{\sigma_{train} - \sigma_{ref_i}}{\sigma_{train}}| > 10\%); then \; Pass \tag{38}$$

Note that the input sensitivity test does not select the inputs for the test. As the number of inputs is infinite, it is impossible to select the representative inputs. However, one can use the knowledge to trim down the simulation space by categorizing the inputs and select one input from each category. For example, if an input is a graph, the categories can be based on the number of vertices and their connectivity. Selecting inputs is based on one's need and is out of the scope of this chapter.

After the input sensitivity test, we can easily trace back the methods that show input-sensitive/input-insensitive behaviors using the method information encoded in the phase centers. For example, we can analyze the phase centers and retrieve the method (feature)

that have the highest weight indicating that the method is most commonly seen in this phase.

5.4 Evaluation

5.4.1 Platform and Benchmarks

Our testbed is the machine with Intel(R) Core(TM) i7-4820K [85] and 40G DDR3-1333 MHz memory. We ran each benchmark on a single node since the executed code should be similar between different nodes. Analyzing the architectural behaviors of a single node is sufficient to cover the different phase behaviors.

Benchmark	Abbrev.	Type	Input Size	Frameworks
Sort	sort	Microbench	10G text	Hadoop, Spark
WordCount	wc			
Grep	grep			
NaiveBayes	bayes	Machine Learning		
Connected Components	cc	Graph Analytics	2^{24} nodes	
PageRank	rank			

Table 10: Evaluated benchmarks

Table 10 shows the evaluated benchmarks from BigDataBench [86]. Each workload has implementations on two frameworks: SPARK and Hadoop. We use "_[hp]" and "_[sp]" to represent the Hadoop and Spark frameworks respectively. For example, `bayes_hp` means the NaiveBayes algorithm implemented on top of Hadoop. The purpose of evaluating on both frames is to test the robustness of SimProf and analyze how phase behaviors look on different frameworks. Since Hadoop is disk-IO intensive when the default configuration is used, we apply common optimizations, such as increasing the memory buffer size of mappers and compressing the output of the mappers before writing to the disk, to improve the performance and make it closer to the real-world settings. The inputs are generated using the data synthesizer provided in BigDataBench.

We evaluated the following sampling approaches. Note that SRS and SimProf are probabilistic sampling techniques for which the range of the CPI error can be quantified as the confidence interval.

- **Single N-second simulation point (SECOND):** This is one of the most common approaches used for simulating data center workloads. The original use was for the transaction-based workloads, such as OLTP [87] and SpecWeb [72]. In those workloads, only a few type of transactions exist and each has a short running time. In that case, when N is sufficiently large, e.g. 10 seconds, it is able to cover the entire transaction behaviors.

- **Simple Random Sampling (SRS):** SRS randomly selects sampling units into the sample in which each sampling unit has an equal chance to be selected. The advantage is that SRS does not need to know the running code in each sampling unit so it can save profiling overhead.

- **Single point for each phase (CODE):** This is the SimPoint-like approach that uses only call stacks to cluster phases and selects the sampling unit closest to the center of the cluster as the simulation point.

- **SimProf:** It uses the stratified random sampling approach. The maximum number of phases is 20 and the threshold of the Silhouette coefficient is 90%.

5.4.2 Accuracy and Sample Size Results

To evaluate the sampling accuracy. We run each workload until its completion to collect the CPI of each sampling unit. By averaging the CPIs of all sampling units, we get the oracle CPI. The sampling errors are computed by comparing the differences between the predicted CPIs with sampling and the oracle CPIs.

Figure 41 shows the comparison of sampling errors of different sampling approaches. The sample size is 20 for SRS, CODE and SimProf while SECOND uses the 10-second

interval as the sample size. SRS sometimes has higher errors than SECOND since different sampling units have higher CPI variations. Although SECOND may not have high CPI errors, in most cases, the sample is not representative since it does not cover all the execution stages. For example, SECOND is not able to cover the reduce stage for all Hadoop workloads. CODE also has a higher error than SimProf since purely using the code signature does not reduce the CPI variation of each phase because of other factors that could affect the performance. The average error of SECOND, SRS, and CODE are 6.5%, 8.9%, and 4.0% respectively, whereas the average error of SimProf is only 1.6%.

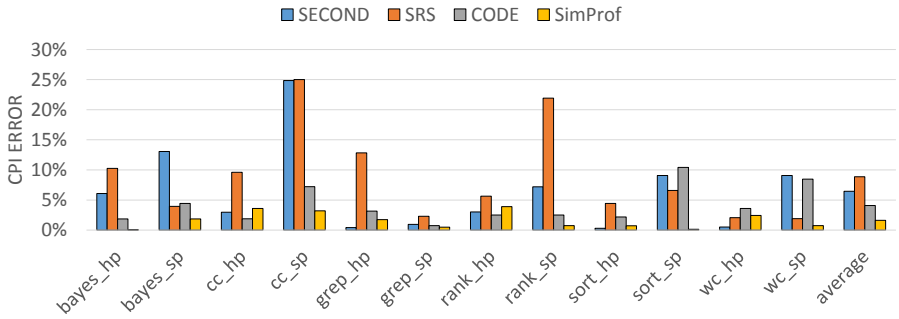


Figure 41: The sampling errors of CPI of different sampling approaches

Another advantage of SimProf is to have the bound of sampling errors thanks to the stratified random sampling. Figure 42 shows the required sample size of SimProf for the 99.7% confidence interval with sampling errors of 5% and 2%. Since SECOND uses the time duration as the sample size, the actual number of sampling units may differ by benchmarks. (Each sampling unit has 100M instructions.) In most benchmarks, the required sample size is less than SECOND while achieving much lower CPI errors except `cc_sp` and `rank_sp`. These two workloads have more phases and the CPI variations of these phases are also high. The average sample sizes of SimProf_0.05, SimProf_0.02 and SECOND are 85, 244 and 611 respectively. This demonstrates that SimProf not only selects more representative samples, which lead to a higher accuracy, but also has a smaller sample size that leads to a higher efficiency.

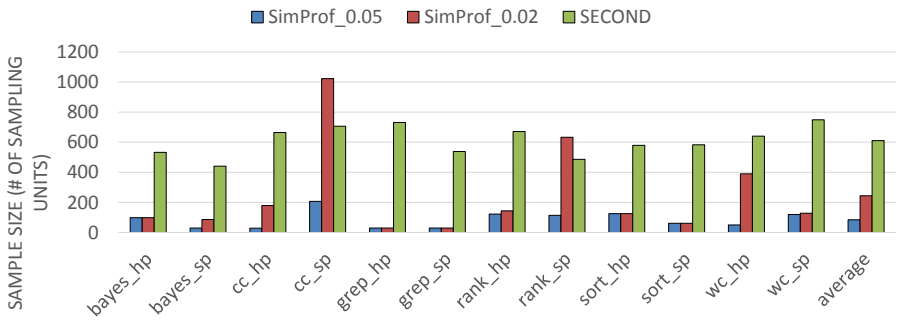


Figure 42: The comparison of the sample size (number of sampling units) between SimProf and SECOND.

5.4.3 Phase Analysis

Figure 43 shows the number of phases in each category of the workloads. It is interesting to see that the number of phases of Spark-based workloads has a much wide range (1 for grep and 9 for cc) than that of Hadoop-based workloads. Some benchmarks, such as `cc` and `rank`, have many more phases since they use more operations and data collection types for the optimization purpose, e.g., GraphX libraries, while in Hadoop implementations, only 1 to 2 map and reduce operations are defined, leading to fewer phases.

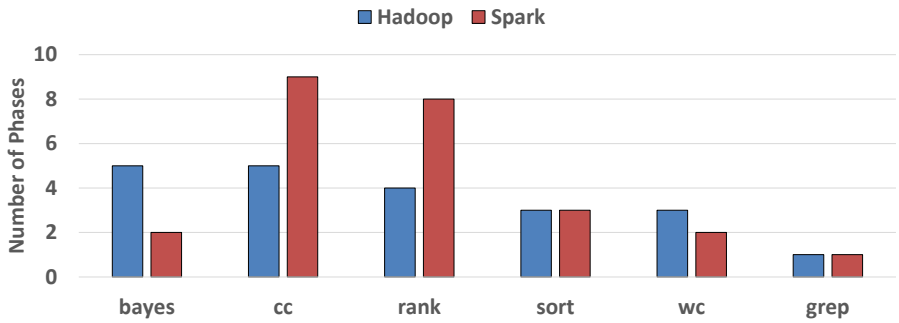


Figure 43: Number of phases

Since we use the workloads that operate on key-value pairs, we can categorize the phase types based on the dominant operations of the phase. We categorize the phase types into: (1) map, (2) reduce, (3) sort and (4) IO types. Sort indicates the key-sorting operation, such as quicksort while the IO includes the operations that read/write to the HDFS due to insufficient memory space. Since operations are often very tightly coupled, the boundary

between them is sometimes blurred. For example, a phase may have both map and IO operations active since each sampling unit in the phase contains both operations. In this case, the type of phase depends on the dominant operation.

Figure 44 shows the breakdown of different phase types while the weight depends on the number of sampling units in the type of phase. Sort operations appear in all Hadoop-based workloads, except `grep_hp` and `sort_hp`. The purpose of the sort operations is to reduce the number of mapper outputs for optimization purposes. Before the mapper outputs are flushed to the disk, they are sorted and merged by keys so that a reducer only receives one copy instead of N copies where N is equal to the number of mappers on the node. By default, Spark-based workloads do not enable this option so the sorting operations are not seen. In general, Hadoop-based workloads spent more time on IO operations instead of doing actual work than Spark-based workloads, which could be one of the reasons Hadoop-based workloads have a lower performance than Spark-based workloads.

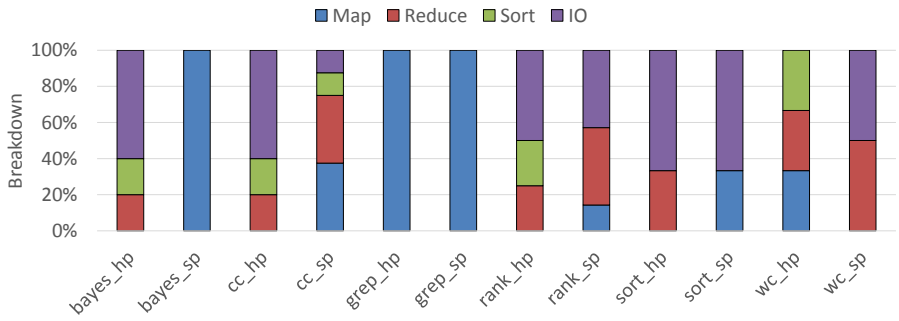


Figure 44: Phase type distribution

Figure 45 shows how the optimal allocation distributes the simulation points between phases. The sample size ratio of a phase represents the number of simulation points in the phase divided by the total number of simulation points. The figure also shows the CoV of CPI, and the weight of the phase. All three numbers range between 0 and 1. Phase 0 has the highest phase weight (29.1%) and the sample size ratio in which 35% of the simulation points belong to this phase. The high CPI variation is caused by the `aggregateUsingIndex` operation, defined in the GraphX library, which performs a

reduce operation. By contrast, although Phase 1 has the second-highest phase weight (18.9%), only 5% of the simulation points belong to the phase since the CPI variation of the phase is low. The phase is dominated by the `mapPartitionsWithIndex` operation, which sequentially converts the lines from an input file to the key-value pairs. The nature of sequential accesses leads to a low CPI variation. Thus, the optimal allocation allocates the simulation points based on both CPI variations and phase weights. Since SimProf captures both performance and call stacks, the information is used by the optimal allocation to determine the best partition of the simulation points.

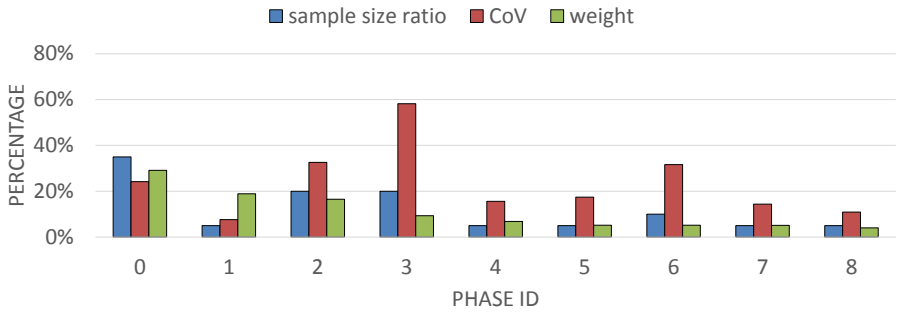


Figure 45: The sample size ratio of each phase of `cc_sp` distributed based on the optimal allocation. (The phases are sorted by the weight)

5.4.4 Input Sensitivity Analysis

Table 11 shows the graphs downloaded from the SNAP website [88] that we use to analyze how phase behaviors change by inputs. Since most of the original graphs are small and have a different number of nodes, we synthesize the Kronecker graphs [89] that have similar connectivity as the original graph. The resulting graphs have the number of nodes between 2^{20} and 2^{24}. One input is selected as the training input, while the rest are the reference inputs. To select the representative inputs, we try to select the ones that are likely to lead to distinct performances of the workloads to cover a wide range of inputs. For graph analytic workloads, we can select the graphs with diverse topology as the representative inputs. For text-based workloads, it is more challenging to identify the representative inputs since identification is more benchmark-dependent. For example, for WordCount, the inputs with

different frequencies of words should be used while for Sort, the inputs with different ordering between words should be used. Since the criteria of representative inputs for text-based workloads requires more analysis on the content of the documents, we evaluated the graph analytic workloads for now, leaving the text-based workloads for future work.

Table 11: Evaluated Inputs

Input Name	Input Type	
Google	Web graph	training input
Facebook	Social Network	reference inputs
Flickr	Online communities	
Wikipedia	Online encyclopedia	
DBLP	Computer science bibliography	
Stanford	Web graph	
Amazon	Product co-purchasing networks	
Road	Road Networks	

Figure 46 shows the percentage of simulation points that are in input-sensitive phases. This indicates the sample size for reference inputs. For training inputs, all simulation points need to be simulated, but for the reference inputs, the simulation points in the input-insensitive phases can be skipped. Using the proposed input sensitivity analysis, the sample size is reduced from 20% up to 45%. On average, 33.7% of the sample size can be reduced.

Figure 46: The input-sensitive sample size

Figure 47 shows the number of input-sensitive and insensitive phases. For most workloads, the number of input insensitive phases is at least 40% of the total number of phases. We found that the input-sensitive phases are likely to have some operations that have time-varying performance. For example, in `cc_sp`, the phase with the `aggregateUsingIndex`

operation has different performances at different execution stages. The performances of the phase change by inputs as well. In addition, a high CPI variation within a phase is not necessarily an input-sensitive phase since the variation could exhibit some pattern while the pattern does not change by inputs, e.g., the quicksort algorithm.

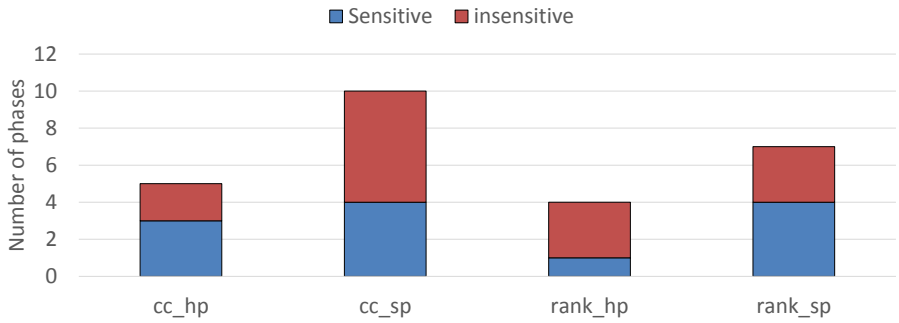


Figure 47: The number of input-sensitive and input-insensitive phases

5.4.5 Framework Comparisons

In this subsection, we compare the phase behaviors of the WordCount benchmark on Hadoop and Spark frameworks. In the following figures, the sampling units are sorted by phase IDs. Blue dots correspond to the left y-axis and represent the CPI of the sampling units while the red lines correspond to the right y-axis and represent the phase IDs of the sampling units.

Figure 48 shows the Spark implementation of WordCount. From the programmers' view, the reduce operation should occur at the second stage, as shown in Figure 34. However, by analyzing the call stacks, we found that the majority of the reduce operation occurs in the first stage. The phase with the majority of the sampling units is dominated by the `combineValuesByKey` operation of the `Aggregator` class, which calls not only the reduce operation but also the map and IO operations, e.g., the first three operations in Figure 34. Then the reduce operation is performed as hashing the key and inserting the key-value pairs into a large in-memory map. If the key already exists in the in-memory map, the newly inserted value is merged with the value in the map. This is the so-called *map-side reduce* optimization used to reduce the size of the mapper output. Since these

operations are tightly coupled, they belong to the same phase. Finally, the second phase reduces the output from the first stage and saves the output to the HDFS. Since the map-side reduce already takes care of the majority of the reducer work, the second phase only occupies 1% of the total sample size. In terms of CPI variations, we expect the first phase that has the reduce operation to have a high CPI variation; however, it shows fairly stable performance. This is likely due to multiple operations being merged together, e.g., map and IO, while the reduce operation is not the dominant operation of the phase. The second phase that contains the HDFS IO operations has a higher CPI variation because different data types are accessed.

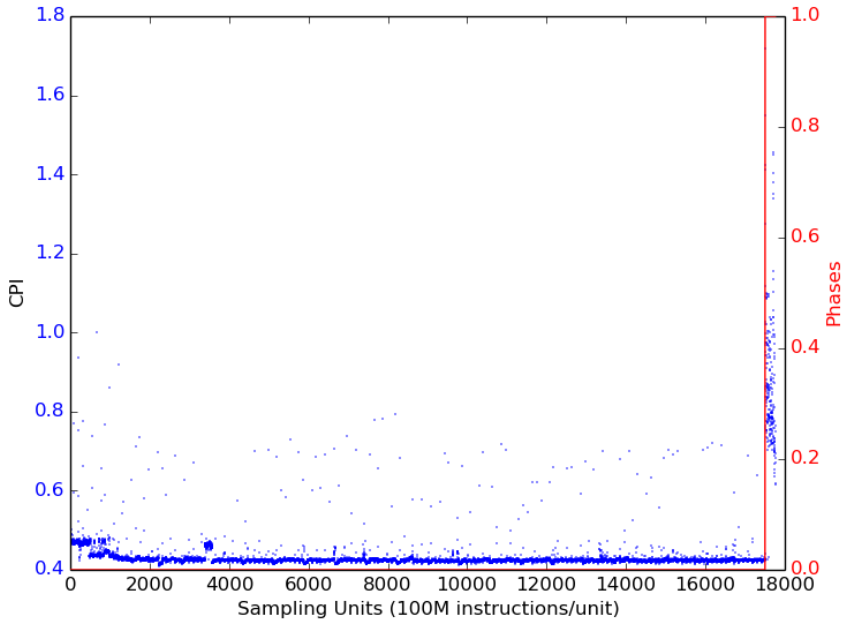


Figure 48: WordCount - Spark implementation

Figure 49 shows the Hadoop implementation of WordCount. The first phase is dominated by the map operation of the `TokenizerMapper` class, which generates the key-value pairs where the key is a word and writes them to the memory buffer. Since the map operations have a good cache locality, the phase has a high performance and a relatively low CPI variation. The second phase is dominated by the combine operation of the

`NewCombinerRunner` class, which performs the map-side reduce operation. Different from the Spark implementation, the combine operation is not coupled with other operations; therefore, the CPI variation is higher in this phase. The third phase is dominated by the sort operation, which uses the quicksort algorithm. The purpose is to reduce the number of mapper outputs sent to a reducer. Because of the recursive nature of the function, the CPI variation is high.

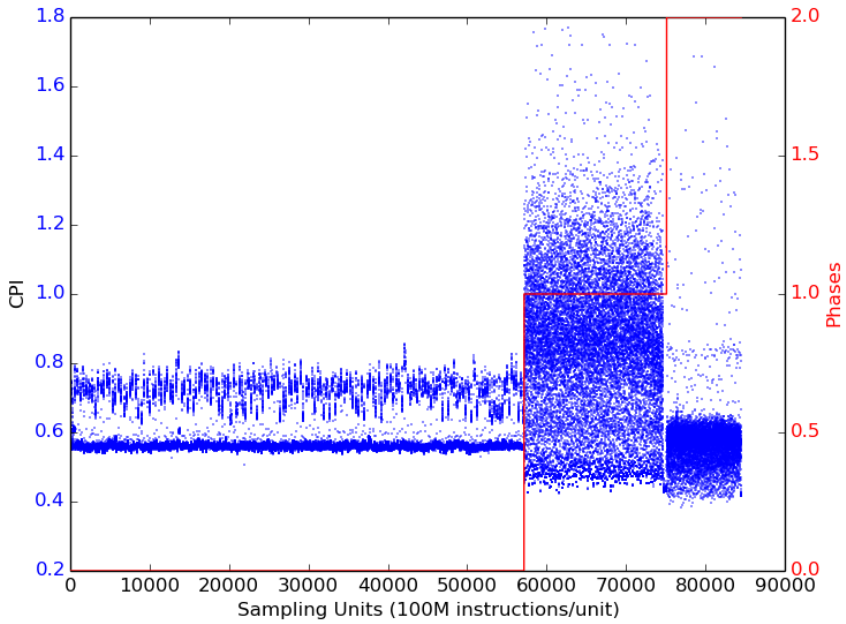


Figure 49: WordCount - Hadoop implementation

The architects can use SimProf to understand the operations composed of a phase and design optimization techniques to improve the performance of a phase. In addition, the CPI variations of some phases can be high since the cache miss rate fluctuates. The stratified random sampling approach, which selects multiple simulation points, in a phase better captures the phase behavior.

5.4.6 Architecture-level Analysis

Figure 50 shows the average CPIs of different types of phases. We can see that even with the same type of phases, different workloads can still lead to diverse CPIs. For map and

reduce type of phases, this is mostly due to different map and reduce methods being called. Depending on the methods, some phases can have higher CPI. For example, `grep_hp` has the lowest CPI since the pattern matching functions, which are compute-intensive and have higher ILP, are dominating. The sort type of phases has higher CPI than the other type of phases since the quicksort algorithm can incur high cache miss rates, except `bayes_hp`. In `bayes_hp`, besides the quicksort algorithm, some computation methods are also dominating, which increase the CPI of the phase. In summary, the map type of phases has lower CPI than the reduce type of phases when both phases coexist in the same benchmark. The CPI variance of both types of phases can be high depending on the called methods of the phases. The sort type of phases has the highest CPI in general while the IO type of phases has similar CPIs between workloads since the methods that read/write the HDFS are shared in common.

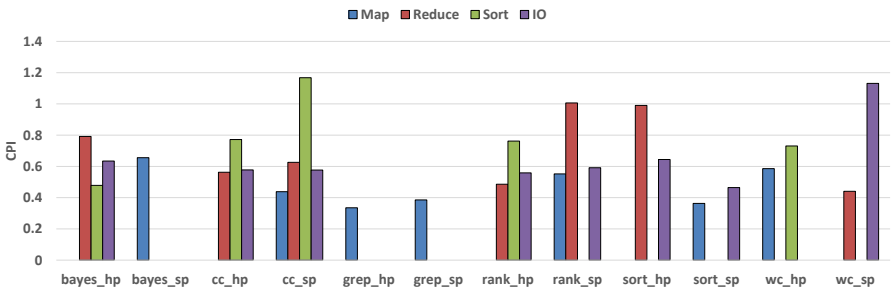


Figure 50: The CPI of each type of phases

Here is a quick summary of our findings.

- **Phase types:** the weights of different phase types of the evaluated benchmarks can be very different. Even for the same phase types, different benchmarks may exhibit different CPIs, e.g., map and reduce types of phases.

- **Number of Phases:** The number of phases of different benchmarks has higher variation in Spark than in Hadoop. For Spark, some complex operations, such as sortByKey, result in several phases for better performance, while some simple operations

can be merged into one phase. For Hadoop, the number of phases is more stable across different benchmarks following the MapReduce execution model.

- **Phase CPIs:** Reduce and sort types of phases are likely to have higher CPIs than other types. In general, both Spark and Hadoop do not have very low CPI due to low ILP. However, for `grep_hp` and `grep_sp`, the IPC can reach 3.0 (capped by 4.0) implying that the high ILP is still possible for the scale-out workloads if the phase is more compute-intensive than memory-intensive.

To understand the performance bottlenecks of different types of phases, we analyzed both frontend and backend events. For frontend events, we analyzed the ITLB and i-cache MPKI. For backend events, we analyzed the DTLB, L1D, L2 and L3 MPKIs. The detailed results are shown as follows.

5.4.6.1 Backend Bottleneck

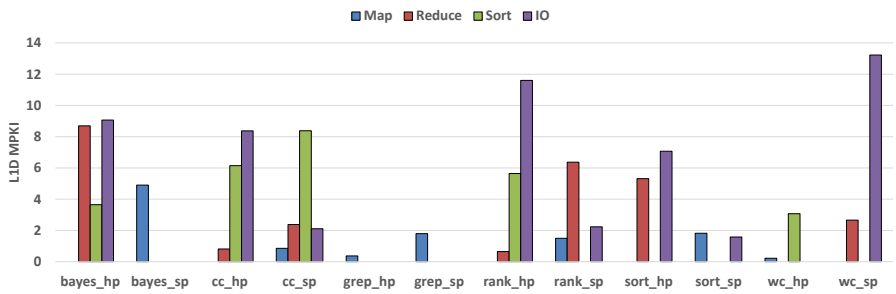


Figure 51: L1D MPKI of each type of phase

Figure 51 shows the L1D MPKI. We can see that the MPKI is less than 10 in all except the IO phases of `rank_hp`, indicating that L1D is effective in capturing the data locality. Another observation is that the Spark-based workloads have lower MPKI than that of Hadoop-based workloads, which could be due to the fact that the Spark execution model can merge multiple operations into the same phase so that the operations that process the same data can be executed together to have better locality. Based on the L1D MPKI, each type of phase does not have consensus behavior. Any type of operations can have low or high MPKIs.

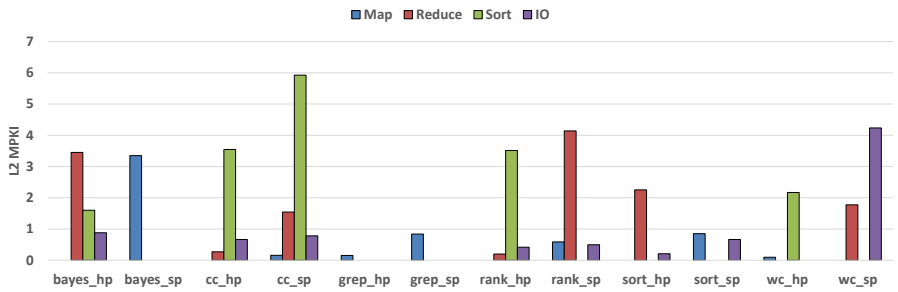


Figure 52: L2 MPKI of each type of phase

Figure 52 shows the L2 MPKI. The L2 MPKI of the IO type of phase is significantly reduced compared to the L1D MPKI, indicating that the working set size of the IO operations can be well fitted into the L2 cache. For the other types of phases, the L2 MPKI is moderately reduced.

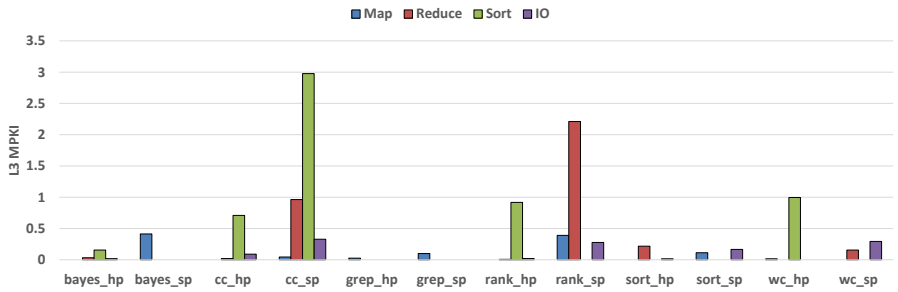


Figure 53: L3 MPKI of each type of phase

Figure 53 shows the L3 MPKI. We observed that (1) the sort type of phases and (2) some reduce types of phases have higher MPKIs than the other types of phases. This is caused by the huge working set size of those operations. The reduce type of phases have higher MPKI in Spark-based implementations than in Hadoop-based implementations. The reason is that the Hadoop-based implementations use the sort operations to make sure the reduce methods can merge the values of the same key from a sorted list without performing random accesses of a large memory partition. The random accesses are done by the sort operation rather than the reduce operation. By contrast, the Spark implementation employs the map-side reduce, which directly merges key-value pairs from the mapper output, leading to random accesses of a large memory partition.

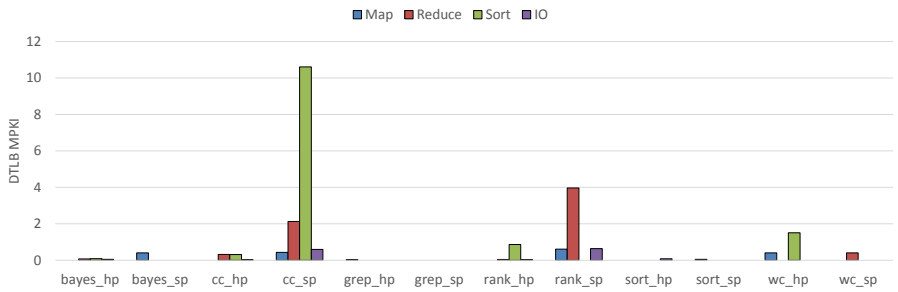


Figure 54: DTLB MPKI of each type of phases

Figure 54 shows the DTLB MPKI. The phases with higher L3 MPKI in `cc_sp` and `rank_sp` also exhibit a similar trend in DTLB MPKI, which corroborates our hypothesis that in Spark-based implementations, the reduce and sort phases have a large working set size being accessed. In addition, it is interesting to see that in the Hadoop-based implementations, the DTLB MPKIs are low since the memory buffer size used in Hadoop is lower than that of Spark, which is possibly the limitation of the Hadoop execution model.

Here is a quick summary of our findings.

- **Backend bottleneck:** The sort and reduce phases are likely to have higher L3 misses. The L3 misses are higher for graph workloads, connected components, and page rank. The reduce phases of Hadoop workloads do not have high L3 misses since the key-value pairs are sorted by the sort phase before being processed by the reduce methods leading to regular data accesses. Thus, the Hadoop workloads have higher L3 misses for the sort phase.

- **Performance correlation:** The CPI is correlated with L3 misses, indicating that most phases of the workloads are backend bound and the performance is limited by L3 misses.

5.4.6.2 Frontend Bottleneck

Figure 55 shows the L1I MPKI. We can see that Spark-based implementations have much lower L1I MPKI than the Hadoop-based implementations. However, even for Hadoop, most phases do not exhibit high L1I MPKI except for reduce and IO phases. By analyzing

the methods of those phases, we found that the high L1I MPKIs are caused by the IO operations of the HDFS. The IO operations also exist in reduce phases by copying the mapper outputs to the memory to be merged. Those operations involve the IO-specific routines that may expand the accessed code base. In Spark, the IO access routines are less frequent than Hadoop because of the high utilization of memory space. However, `rank_sp` and `wc_sp`, which have more frequent IO operations, also incur higher L1I MPKI than other benchmarks. We also found that the L1I MPKI is also negligible (less than 0.1), implying that the working set of instructions is not large.

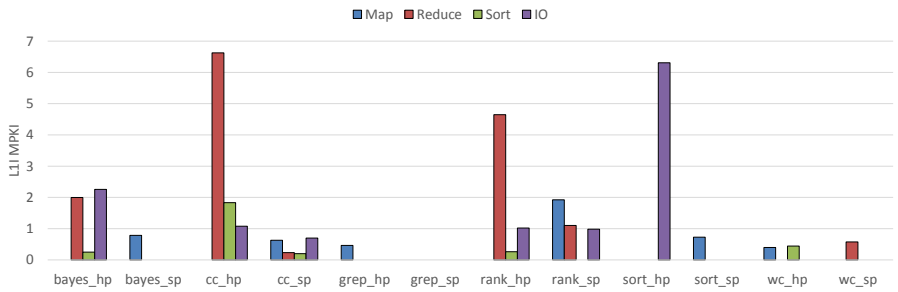


Figure 55: L1I MPKI of each type of phases

Here is a quick summary of our findings.

- Most phases in Hadoop and Spark workloads do not exhibit high L1I MPKI. Only the phases with IO operations have higher MPKIs, which may involve in IO-specific routines, e.g., HDFS access procedures, which may increase the access code sizes or the context switch of threads between different cores, leading to higher L1I misses.

5.5 Summary

SimProf is the first sampling framework for data analytic workloads based on modern computing frameworks, such as Apache Spark and Hadoop. It identifies phases using call stacks and applies statistical sampling methods to select the final simulation points, which can sample the phases that do not have homogeneous performances. It achieves a 1.6%

sampling error of CPI with only 20 simulation points, each of which has 100 million instructions. This sample size is less than 5% of that of a single 10-second interval. In addition, it detects input-insensitive phases, whose performance does not change by inputs. When simulating multiple inputs with different volumes, the number of simulation points can be further reduced by 33.7% on average as the results of skipping input-insensitive phases.

The contributions of this chapter are as follows.

- We propose the first sampling framework, SimProf, based on phase analysis for data analytic workloads, which are built on top of commonly used computing frameworks, e.g., Apache Hadoop and Spark.
- We propose to apply stratified random sampling, a statistical sampling approach to select the representative simulation points by taking into account the factors that cause performance variation within a phase.
- We propose an input sensitivity test to reduce simulation points when exploring multiple inputs by eliminating the points whose performance does not change by inputs.
- SimProf provides method-level information of each phase, which may help architects to understand the performance bottlenecks of data analytic workloads with managed runtime, e.g., Java.

For future work, we plan to study the phase behaviors of more data analytic workloads with various machine learning and graph algorithms. Using the generated simulation points, we would like to study how to improve the performance by accelerating the common operations that cause low-performance phases.

CHAPTER 6

CONCLUSION

Architectural simulation is one of the most important performance modeling techniques for architecture design. Compared to the native execution speed, the simulation speed has a 10,000x times slowdown. Therefore, techniques that reduce the simulation time are required. Prior techniques target conventional benchmark suites, which are single-threaded or have limited parallelism. However, the emerging workloads feature massive parallelism and huge data volume, which increase the simulation time further. In this dissertation, we present three simulation techniques to simulate those large-scale applications accurately and efficiently. We developed the techniques based on understanding the execution behaviors of the workloads. In addition, some mathematical theories are used to support and strengthen the techniques. Here is the summary of the techniques.

- Chapter 3 introduces the sampling techniques, inter-launch and intra-launch sampling, for GPGPU kernels. In inter-launch sampling, a new feature vector is proposed to capture the control and memory divergence, which are the unique characteristics of GPGPU kernels. In intra-launch sampling, the dynamic sampling approach is proposed to sample the instruction intervals that are non-deterministic due to warp scheduling effect.

- Chapter 4 presents a hybrid modeling technique that combines the functional simulation and analytical models to model the performance of a GPGPU kernel. The technique predicts the performance of a multithreaded GPU core using the performance of a single warp. This technique achieves low modeling error of performance compared to a detailed timing simulation while achieving two orders of magnitude speedup. In addition, it provides detailed insights into the performance bottlenecks of a GPGPU kernel.

- Chapter 5 presents a sampling framework for the emerging data analytic workloads. The framework identifies the phase behaviors of the workloads and samples each phase to get the simulation points. The framework uses method information from the call stacks to identify phases. Being aware that some phases do not have homogeneous performance, stratified random sampling is used to select samples based on the performance variation of each phase. In addition, the framework further reduces the simulation time by identifying the input-insensitive phases. The framework increases the accuracy and efficiency of simulating the data analytic workloads.

The emerging workloads become more complex and the executions not only involve CPUs but also other resources, e.g., network, disk. Furthermore, a workload can be distributed onto multiple compute nodes making the simulation more challenging. To faithfully model the performance of these workloads, it is required to develop different simulation techniques. This dissertation serves as a starting point to investigate such techniques to simulate these workloads accurately and efficiently.

REFERENCES

[1] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, (New York, NY, USA), pp. 45–57, ACM, 2002.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[3] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, (New York, NY, USA), pp. 84–97, ACM, 2003.

[4] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation.," in *IEEE 16th International Symposium on High Performance Computer Architecture (HPCA), 2010* (M. T. Jacob, C. R. Das, and P. Bose, eds.), pp. 1–12, IEEE Computer Society, 2010.

[5] E. K. Ardestani and J. Renau, "Esesc: A fast multicore simulator using time-based sampling," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '.

[6] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *ISPASS*, 2013.

[7] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, Sept. 2006.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," Tech. Rep. TR-811-08, Princeton University, 2008.

[9] T. Karkhanis and J. Smith, "A first-order superscalar processor model," in *ISCA '04*.

[10] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. Smith, "A mechanistic performance model for superscalar out-of-order processors," *TOCS '09*.

[11] T. M. Conte, M. A. Hirsch, and K. N. Menezes in *ICCD*, pp. 468–477, IEEE Computer Society.

[12] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder, "The strong correlation between code signatures and performance," in *ISPASS*, 2005.

[13] H. Patil, R. S. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation," in *MICRO*, 2004.

[14] C.-K. L. et al., "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[15] E. Perelman, G. Hamerly, and B. Calder, "Picking statistically valid and early simulation points.," in *IEEE PACT*, pp. 244–255, IEEE Computer Society, 2003.

[16] M. V. Biesbrouck, T. Sherwood, and B. Calder, "A co-phase matrix to guide simultaneous multithreading simulation," ISPASS '04, 2004.

[17] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder, "Motivation for variable length intervals and hierarchical phase behavior," in *In IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 135–146, 2005.

[18] L. Eeckhout, J. Sampson, and B. Calder, "Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation," in *In IISWC*, pp. 2–12, 2005.

[19] J. Lau, E. Perelman, and B. Calder, "Selecting software phase markers with code structure analysis," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, (Washington, DC, USA), pp. 135–146, IEEE Computer Society, 2006.

[20] T. E. Carlson, W. Heirman, K. Craeynest, and L. Eeckhout, "Barrierpoint: Sampled simulation of multi-threaded applications," in *ISPASS*, 2014.

[21] J. J. Yi, S. V. Kodakara, R. Sendag, L. D. J., and D. M. Hawkins, "Characterizing and comparing prevailing simulation techniques.," in *HPCA*, pp. 266–277, IEEE Computer Society, 2005.

[22] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "An evaluation of stratified sampling of microarchitecture simulations," in *In Workshop on Duplicating, Deconstructing, and Debunking, ISCA*, 2004.

[23] M. Oskin, F. T. Chong, and M. Farrens, "Hls combining statistical and symbolic simulation to guide microprocessor designs," ISCA '00, 2000.

[24] L. Eeckhout, R. H. B. Jr., B. Stougie, K. D. Bosschere, and L. K. John, "Control flow modeling in statistical simulation for accurate and efficient processor design studies.," in *ISCA*, pp. 350–363, IEEE Computer Society, 2004.

[25] S. Nussbaum and J. E. Smith, "Modeling superscalar processors via statistical simulation," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, (Washington, DC, USA), pp. 15–24, IEEE Computer Society, 2001.

[26] L. Eeckhout, K. D. Bosschere, and H. Neefs, "Performance analysis through synthetic trace generation.," in *ISPASS*, pp. 1–6, 2000.

[27] L. Eeckhout, *Computer Architecture Performance Evaluation Methods*. Morgan & Claypool Publishers, 1st ed., 2010.

[28] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "Simflex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, pp. 18–31, July 2006.

[29] A.-T. Nguyen, P. Bose, K. Ekanadham, A. K. Nanda, and M. M. Michael, "Accuracy and speedup of parallel trace-driven architectural simulation.," in *IPPS*, pp. 39–44, IEEE Computer Society, 1997.

[30] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, "The wisconsin wind tunnel: Virtual prototyping of parallel computers.," in *SIGMETRICS*, pp. 48–60, 1993.

[31] M. Chidester and A. George, "Parallel simulation of chip-multiprocessor architectures," *ACM Trans. Model. Comput. Simul.*, vol. 12, pp. 176–200, July 2002.

[32] J. Chen, M. Annavaram, and M. Dubois, "Slacksim: A platform for parallel simulations of cmps on cmps," *SIGARCH Comput. Archit. News*, vol. 37, pp. 20–29, July 2009.

[33] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 475–486, ACM, 2013.

[34] 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 *PLDI*, 2005.

[35] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *HPCA '10*.

[36] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. Smith, "A performance counter architecture for computing accurate cpi components," in *ASPLOS '06*.

[37] S. Eyerman and L. Eeckhout, "Per-thread cycle accounting in smt processors," in *ASPLOS '09*.

[38] X. Chen and T. Aamodt, "Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs," *TACO '11*.

[39] M. Breughe, S. Eyerman, and L. Eeckhout, "A mechanistic performance model for superscalar in-order processors.," in *ISPASS '12*.

[40] X. E. Chen and T. M. Aamodt, "A first-order fine-grained multithreaded throughput model," in *HPCA*, 2009.

[41] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. W. Hwu, "An adaptive performance modeling tool for gpu architectures," in *PPoPP*, 2010.

[42] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *HPCA*, 2011.

[43] J. W. Sim, A. Dasgupta, H. Kim, and R. Vuduc, ""gpuperf: A performance analysis framework for identifying performance benefits in gpgpu applications"," in *PPoPP*, 2012.

[44] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *ISCA*, 2009.

[45] S. S. Baghsorkhi, I. Gelado, M. Delahaye, and W.-m. W. Hwu, "Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors," PPoPP '12, 2012.

[46] T. Tang, X. Yang, and Y. Lin, "Cache miss analysis for gpu programs based on stack distance profile," in *ICDCS '11*.

[47] C. Nugteren, G. van den Braak, H. Corporaal, and H. Bal, "A detailed GPU cache model based on reuse distance theory," in *HPCA '14*.

[48] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere, "Workload design: Selecting representative program-input pairs," in *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, (Washington, DC, USA), pp. 83–94, IEEE Computer Society, 2002.

[49] M. B. Breughe and L. Eeckhout, "Selecting representative benchmark inputs for exploring microprocessor design spaces," *ACM Trans. Archit. Code Optim.*, vol. 10, pp. 37:1–37:24, Dec. 2013.

[50] W. C. Hsu, H. Chen, P. C. Yew, and H. Chen, "On the predictability of program behavior using different input data sets," in *Interaction between Compilers and Computer Architectures, 2002. Proceedings. Sixth Annual Workshop on*, pp. 45–53, 2002.

[51] J.-C. Huang, L. Nai, H. Kim, and H.-H. Lee, "Tbpoint: Reducing simulation time for large-scale gpgpu kernels," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 437–446, May 2014.

[52] "Introducing TITAN." http://www.olcf.ornl.gov/titan/.

[53] "Swiss national supercomputing centre." http://www.cscs.ch/.

[54] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *IISWC*, 2012.

[55] "Macsim." http://code.google.com/p/macsim/.

[56] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems," in *PACT*, 2010.

[57] J.-C. Huang, J. H. Lee, H. Kim, and H.-H. S. Lee, "Gpumech: Gpu performance modeling technique based on interval analysis," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, (Washington, DC, USA), pp. 268–279, IEEE Computer Society, 2014.

[58] J. B. Andrews and C. D. Polychronopoulos, "An analytical approach to performance/-cost modeling of parallel computers," *J. Parallel Distrib. Comput.*, vol. 12, no. 4, 1991.

[59] G. Bernacchia and M. C. Papaefthymiou, "Analytical macromodeling for high-level power estimation," in *ICCAD '99: Proc. of the 1999 IEEE/ACM Int'l. conference on Computer-aided design*, 1999.

[60] J. A. Butts and G. S. Sohi, "A static power model for architects," *Microarchitecture*, vol. 0, pp. 191–201, 2000.

[61] W. Jia, K. Shaw, and M. Martonosi, "Stargazer: Automated regression-based gpu design space exploration," in *ISPASS '12*.

[62] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44 –54, oct. 2009.

[63] T. S. Karkhanis and J. E. Smith, "Automated design of application specific superscalar processors: an analytical approach," in *ISCA '07: Proc. of the 34th annual Int'l. Symp. on Computer Architecture*, (New York, NY, USA), ACM, 2007.

[64] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1370–1380, 2008.

[65] T. Rogers, M. O'Connor, and T. Aamodt, "Cache-conscious wavefront scheduling," in *MICRO '12*.

[66] J. Miller, H. Kasture, G. Kurian, C. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores.," in *HPCA '10*.

[67] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, *MacSim: A CPU-GPU Heterogeneous Simulation Framework User Guide*. Georgia Institute of Technology, 2012.

[68] The IMPACT Research Group, UIUC, "Parboil benchmark suite." http://impact.crhc.illinois.edu/parboil.php.

[69] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC'09*, 2009.

[70] T. White, "Hadoop: The definitive guide," 2009.

[71] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 293–307, USENIX Association, 2015.

[72] R. Hariharan, N. Sun, and S. Microsystems, "Workload characterization of specweb2005," 2006.

[73] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-out processors," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, (Washington, DC, USA), pp. 500–511, IEEE Computer Society, 2012.

[74] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.

[75] E. PARSA, *Rigorous and Practical Server Design Evaluation*. Georgia Institute of Technology, 2015.

[76] Oracle, "JVM TI Reference." WWW page, 2007.

[77] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 1383–1394, ACM, 2015.

[78] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (Berkeley, CA, USA), pp. 599–613, USENIX Association, 2014.

[79] M. Odersky and M. Zenger, "Scalable component abstractions," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pp. 41–57, 2005.

[80] M. Kambadur, K. Tang, and M. A. Kim, "Harmony: collection and analysis of parallel block vectors," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, 2012.

[81] M. Annavaram, R. Rakvic, M. Polito, J.-Y. Bouguet, R. A. Hankins, and B. Davies, "The fuzzy correlation between code and performance predictability," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, (Washington, DC, USA), pp. 93–104, IEEE Computer Society, 2004.

[82] Standard Performance Evaluation Corporation, "Spec CPU 2006." http://www.spec.org/cpu2006.

[83] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[84] S. Wikipedia and L. Books, *Sampling (Statistics): Census, Sample, Stratified Sampling, Sampling Bias, Statistical Unit, Opinion Poll, Statistical Survey, Margin of Erro*. General Books LLC, 2011.

[85] INTEL, "Intel Core i7 Processors." http://www.intel.com/products/processor/corei7.

[86] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: a big data benchmark suite from internet services," HPCA, 2014.

[87] C. A. Curino, D. E. Difallah, A. Pavlo, and P. Cudre-Mauroux, "Benchmarking OLTP/Web databases in the cloud: The OLTP-bench framework," in *Proceedings of the Fourth International Workshop on Cloud Data Management*, CloudDB '12, pp. 17–20, 2012.

[88] J. Leskovec and R. Sosič, "SNAP: A general purpose network analysis and graph mining library in C++." http://snap.stanford.edu/, June 2014.

[89] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *J. Mach. Learn. Res.*, vol. 11, pp. 985–1042, Mar. 2010.