Accelerating the Evaluation of Large Workloads on Post-Dennard Systems with Sampling

Alen Kandathumthodukayil Sabu

National University of Singapore

2024

Accelerating the Evaluation of Large Workloads on Post-Dennard Systems with Sampling

Alen Kandathumthodukayil Sabu
(M.E., BITS-Pilani)

A Thesis Submitted for
the degree of Doctor of Philosophy

DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING

National University of Singapore

2024

Supervisor:
Assistant Professor Trevor Erik Carlson

Examiners:
Professor Li-Shiuan Peh
Associate Professor Weng-Fai Wong

Declaration

I hereby declare that this thesis is my original work and that I have written it in its entirety. I have duly acknowledged all sources of information which have been used in the thesis.

This thesis has not been submitted for any degree in any university previously.

[illegible]

Alen Kandathumthodukayil Sabu

December 05, 2024

Dedicated to my family, who made me possible.

I am silver and exact. I have no preconceptions.
Whatever I see I swallow immediately
Just as it is, unmisted by love or dislike.
I am not cruel, only truthful
The eye of a little god, four-cornered.
Most of the time I meditate on the opposite wall.
It is pink, with speckles. I have looked at it so long
I think it is part of my heart. But it flickers.
Faces and darkness separate us over and over.

Sylvia Plath

Acknowledgments

I never imagined that the clunky, white machine with a black-and-white CRT monitor running *Windows 95* that I first encountered in primary school would evolve into the powerful computers we have today. In school, our access to computers was very limited. The Internet, accessed through slow landline dial-up connections, often required tens of seconds to load a single webpage. We could access broadband Internet at local computer cafes (an earlier form of Internet-as-a-service?) offering slightly faster connections. Those were the days when floppy disks and cassette tapes were being replaced by CDs. Computers evolved rapidly and became increasingly affordable. As a millennial, I was fortunate enough to witness firsthand the technological shifts of that era. My growing interest in mathematics and programming during my senior year of high school led me to pursue an undergraduate degree in computer science right after the Great Recession.

I often wonder what might have been had I chosen a different path, perhaps in mathematics or literature. Irregardless, pursuing computer science was indeed a great choice, and the journey has been quite an adventure, especially over the past six years. From sleepless nights, rejections, and handling depression to moments of accomplishment, I can't say it hasn't been a fun ride. While pursuing knowledge and going through tough times, this phase of my life was also about self-reflection and exploration.

Looking back on the times that inspired me to pursue academic research, I am grateful for the influence of my family. My mom, dad, and sister have always been my biggest supporters. Their trust and encouragement have been the bedrock upon which I have built my aspirations. I am indebted to my grandfathers, whose passion for knowledge, scientific temper, and sense of righteousness have had a significant impact on my life. My extended family was helpful throughout, especially my amazing cousins. They always welcomed me with open arms and open hearts during my travels. I can confidently attest that they are truly the best. I am indebted to several inspiring teachers who have shaped my academic journey. In particular, my senior high school physics teacher, Jerin Jose, introduced me to research thinking, and later, Biju Raveendran motivated me to pursue systems research for my master's thesis. I am grateful to the many inspiring individuals I interacted with at BITS Pilani (Goa), especially Sreejith Vidhyadharan, Bharat Deshpande, and Anguraj Baskar, among others. My stint at NetApp exposed me to the field of systems performance modeling and measurement. The freedom to

explore recent publications and work on improving performance models was a unique experience that ignited my passion for research in the area.

I want to express my heartfelt gratitude to my advisor, Trevor E. Carlson, for his guidance, support, and encouragement throughout my PhD. His mentorship was instrumental in shaping me into the researcher I am today. Despite our occasional disagreements, he was consistently patient and understanding right from our first meeting in Bengaluru. He was particularly supportive during tough times, such as the COVID-19 pandemic, when it was hard to be productive. His emphasis on research ethics (particularly when conducting experiments) and kindness toward fellow researchers (since we often received harsh comments instead of constructive criticism from conference reviewers) have significantly influenced my approach to academic research. As a well-recognized figure in computer architecture conferences, Trevor often introduced me to other researchers as an expert in sampling and simulation methodologies, which significantly benefited me during my job search.

I was fortunate to have Harish Patil and Wim Heirman as my collaborators throughout my PhD. Their insights and expertise were invaluable to my research. Given the extensive interaction we have had, Harish was a co-advisor during my PhD. He constantly encouraged me and helped me throughout. I also had the privilege of working under his mentorship during my internship at Intel. At a time when few companies were hiring, Harish went the extra mile to secure me this opportunity. The six months I spent working with him were both enjoyable and intellectually stimulating.

I am grateful for the thought-provoking and inspiring conversations with numerous computer architecture researchers and practitioners. I would like to acknowledge the helpful interactions with Lizy John, Lieven Eeckhout, Jason Lowe-Power, Magnus Sjalander, Timothy Pinkston, Yifan Sun, and Matt Sinclair, which helped refine my research direction and ensure its relevance. Although limited, I benefited from insightful conversations with industry experts like Gilles Pokam, Gabriel Loh, Alexander Isaev, Karthik Sankaranarayanan, Sudhavana Gurumurthi, Jason Clemens, and Joseph Greathouse. Their perspectives were instrumental in broadening my research in this field. I would like to extend my gratitude to my thesis examiners, Li-Shiuan Peh and Weng-Fai Wong, and the department representative, Ambuj Varshney, for their kind and constructive feedback. Their careful reviews significantly improved the quality of my thesis.

I am thankful to the entire CompArch group (Trevor's research group at NUS) for fostering a supportive and collaborative environment. Even though many of us have not worked together directly, the camaraderie and enthusiasm within the group have been a constant source of inspiration. I would also like to thank Stephanie Hepner for her assistance in proofreading my papers. The CompArch group was relatively small when I joined in 2018. Jinho Lee was the sole PhD student in the group at the time. We experienced the highs and lows of academic life together, offering mutual support as we navigated the challenges. Neethu Mallya helped me get started in the Singaporean academic environment. She was helpful in several aspects, including discussions about potential research directions. Andreas Diavastos was always available for both research and personal conversations, providing invaluable support. His comments on my papers were of great help. The group began to grow with individuals from diverse backgrounds.

Soon, Ali Hajiabadi, Burin Amornpaissanon, and Newton Singh joined, followed by Ahmed Shalabi and Razvan Nitu. We shared many memorable experiences, including Friday evening basketball games and group outings. Coffee chats with Ali and Andreas were the perfect respite after a day of computer architecture. However, the COVID-19 pandemic disrupted these social gatherings. Post-pandemic, we resumed group activities like hiking, biking, and kayaking along with new members, such as Arash Pashrashid, Yun Chen, Yaswanth Tavva, Lingfeng Pei, Miao Yu, Tingting Xiang, Wei Siew Liew, Changxi Liu, and Yihao Fu. The camaraderie within the group fostered stimulating conversations and cultural exchanges. I had the pleasure of collaborating with Changxi Liu on several projects. Our brainstorming sessions over Indian food were always productive. I also collaborated with Qingxuan (Ray) Kang and Akanksha Chaudhari on a project. In the latter part of my PhD, I had the opportunity to work with Jikun Zhang on some interesting projects. These experiences, enriched by friendships and intellectual collaboration, have left me with core memories and a profound appreciation for the people who shaped my journey.

I would also like to thank several individuals outside of our research group who made my six years at NUS memorable. Nitya Lakshmanan has been my go-to friend at NUS, and interactions with her often helped me navigate the challenges of grad school while keeping my sanity intact. Soundarya Ramesh was incredibly patient and kind during our deep conversations over shared meals. The workplace was a fun environment, thanks to Ayush Mishra, Raj Joshi, Mohit Upadhyay, and Rohan Juneja. Singapore's warm weather ushered lasting friendships with Ashwin Ram, Varsha Suresh, Georgina Roca, Ashwin Kumar, Febin Issac, and Ashok Narendranath. Towards the final stages of my PhD, I met some awesome undergrads like Inaz Begum, Joann Stanley, and Bebin Joseph, who introduced me to more interesting people. The list would be incomplete without mentioning Aung Nyein Kyaw (*aka* "bro") for graciously brewing both hot and cold beverages for us over the years. Among those I met at conferences who greatly appreciated various aspects of my research include Charles Hong, Zhantong Qiu, Nandeeka Nayak, and Hyokeun Lee. Zhantong collaborated with me on a project, and her excitement about building simulators is a perpetual mystery to me. I would like to express my heartfelt appreciation to my long-time friends, who have been a constant source of encouragement. In particular, I would like to thank Ijaz Muhamed, Michelle Varghese, Krishnanand SJ, Rahul Udayakumar, Tobin Mathew, Gokul Rajan, Vipin TM, Majo Oommen, Baptist Joseph, Indrajeet Khandekar, Prashant Singh, Anjali Ajayan, and Anand Sasidharan. Their presence in my life has made this journey memorable and enjoyable. I would also like to express my sincerest gratitude and appreciation to the staff at SoC, UHC, UCS, and the cafeterias for their valuable service throughout my PhD. Finally, I would like to acknowledge the generous grants from Intel Corporation and the travel support provided by IEEE TCCA and ACM SIGARCH during my doctoral research.

Contents

Abstract

As the traditional Moore's Law-driven performance gains have plateaued with the end of Dennard scaling, computer architects adopted novel design strategies to further improve performance. This marked a radical shift in the design of next-generation computing systems, including multi-core processors, accelerators, and heterogeneous systems. Evaluating the performance of complex, realistic workloads running on these systems poses unique challenges, particularly due to the long simulation times. Sampling serves as a promising solution by intelligently selecting the representative subsets of a workload for performance evaluation. In this thesis, we explore novel methodologies to evaluate the performance of post-Dennard systems in a fast and efficient way using sampling.

To address these challenges, we first propose LoopPoint – a sampled simulation methodology that applies to general-purpose multi-threaded workloads. LoopPoint uses application loops to demarcate regions that represent the amount of work done. We demonstrate that LoopPoint reduces the simulation time of large multi-threaded workloads from a few years to a few hours. In a follow-up work, Viper, we make use of the hierarchical structure of program execution to select regions of finer granularity suitable for RTL-level simulations. We show that naive adaptations of SimPoint or LoopPoint may not result in an optimal sample, as the application periodicity and phases vary among workloads.

Modern architectures often incorporate complex dynamic optimization techniques to improve system performance gains at runtime. However, prior sampled simulation methodologies are incapable of handling the dynamic nature of software and hardware. On this front, we propose Pac-Sim, which can be used to evaluate dynamically optimized software and hardware. Pac-Sim performs online analysis and relies on a real-time predictor to determine detailed simulation regions. This allows Pac-Sim to accurately evaluate dynamically scheduled applications, accounting for any runtime performance variability.

The increasing computational demand posed by high-performance computing and artificial intelligence workloads is driving the shift toward heterogeneous architectures. Simulation of future heterogeneous systems is essential in understanding the interactions between compute components, but full-program simulations are prohibitively time-consuming and resource-intensive. We propose XPU-Point to select representative regions of heterogeneous CPU-GPU workloads to enable fast, accurate sampled simulations. XPU-Point significantly speeds up the simulation of HPC and AI workloads without compromising accuracy.

To summarize, we show that simulation solutions alone are insufficient because of the significant slowdown observed, and sampling works as an efficient technique to render the simulation of large workloads tractable. We evaluate a variety of multi-core and heterogeneous workloads to develop methodologies that accelerate the performance evaluation and design space exploration of novel architectures.

List of Publications

1. **Alen Sabu**, Harish Patil, Wim Heirman, Trevor E. Carlson, "*LoopPoint: Checkpoint-driven Sampled Simulation for Multi-threaded Applications*," in IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2022.
2. **Alen Sabu**†, Changxi Liu†, Trevor E. Carlson, "*Viper: Utilizing Hierarchical Program Structure to Accelerate Multi-core Simulation*," in IEEE Access, 2024.
3. Changxi Liu†, **Alen Sabu**†, Akanksha Chaudhari, Qingxuan Kang, Trevor E. Carlson, "*Pac-Sim: Simulation of Multi-threaded Workloads using Intelligent, Live Sampling*," in ACM Transactions on Architecture and Code Optimization (TACO), 2024.
4. **Alen Sabu**, Harish Patil, Wim Heirman, Changxi Liu, Trevor E. Carlson, "*XPU-Point: Simulator-Agnostic Sample Selection Methodology for Heterogeneous CPU-GPU Applications*," in International Conference on Parallel Architectures and Compilation Techniques (PACT), 2025.

Works in Progress

1. **Alen Sabu**, Zhantong Qiu, Harish Patil, Changxi Liu, Wim Heirman, Jason Lowe-Power, Trevor E. Carlson, "*Accelerated Simulation of Parallel Workloads using Loop-Bounded Checkpoints*."

Other Relevant Publications

1. Harish Patil, Alexander Isaev, Wim Heirman, **Alen Sabu**, Ali Hajiabadi, Trevor E. Carlson, "*ELFies: Executable Region Checkpoints for Performance Analysis and Simulation*," in International Symposium on Code Generation and Optimization (CGO), 2021.

Non-Refereed / Non-Proceedings

1. **Alen Sabu**, Harish Patil, Wim Heirman, Trevor E. Carlson, "*ROIperf: Rapid Validation and Iterative Tuning of Workload Sampling Methodologies*." in Workshop on Computer Architecture Modeling and Simulation (CAMS), 2023.
2. **Alen Sabu**, Harish Patil, Wim Heirman, Alexander Isaev, Trevor E. Carlson, "*Approaching a High-Performance, General-Purpose Multi-Threaded Sampling Methodology*." in Young Architect Workshop (YArch), 2020.

† Joint first authors

List of Figures

List of Tables

Chapter 1

Introduction

உறங்கு வதுபோலுஞ் சாக்காடு உறங்கி விழிப்பது போலும் பிறப்பு.[1]

— குறள்: 339

1.1 The Context

Central processing units (CPUs or processors) have long been the cornerstone of computing, responsible for executing instructions and managing system resources. To ensure efficient resource allocation and meet the ever-growing computational demands, accurately estimating the performance of processors is essential. Computer architects typically rely on microarchitectural simulations to assess system performance metrics and compare design choices. The processor designs undergo a comprehensive evaluation of power consumption, performance capabilities, area requirements, and their trade-offs prior to fabrication.

With the end of Moore's law [9], computer architects have turned to alternative approaches to enhance computational capabilities. One prominent strategy involves a shift towards increasing the core count [10, 11] and embracing heterogeneity in archi-

[1](transl.) *Death is sinking into slumbers deep; Birth again is waking out of sleep. — Kural: 339*

tectures [12], complemented by the introduction of several software- and system-level optimizations aimed at improving performance and power efficiencies [13]. As processors/systems continue to evolve in complexity and power, accurately assessing their performance characteristics becomes increasingly intricate. Understanding the workload for the analysis and performance prediction of future systems is an extremely difficult task. Workloads may have extremely long run times and are fairly sophisticated with OS, library, and hardware requirements.

Microarchitecture simulators like gem5 [6] and Sniper [14] are heavily used to estimate the performance of real-world workloads on a new processor design. The purpose of these simulations is to evaluate the performance of a proposed architecture, identify potential bottlenecks, and improve the efficiency of the hardware design before it is implemented in physical hardware. However, simulators are orders of magnitude (typically, $10,000\times$ or more [6]) slower as compared to native execution. This challenge is further exacerbated by the increasing complexity of modern architectures, which necessitates the development of efficient performance evaluation techniques. The focus of this thesis is to address this critical gap by proposing novel workload sampling methodologies that enable fast and accurate performance evaluation of future systems.

1.2 Challenges Involved

For several decades, Moore's law, coupled with Dennard scaling [15], fueled exponential performance gains in single-core processors. This trend is reflected in the significant performance gains observed for SPECint and SPECfp benchmarks as shown in Figure 1.1. However, as Dennard scaling reached its physical limits, the industry shifted its focus to multi-core and heterogeneous architectures. This effectively extended the performance gains predicted by Moore's Law but also necessitated the development of entirely new techniques and infrastructures to accurately evaluate system performance

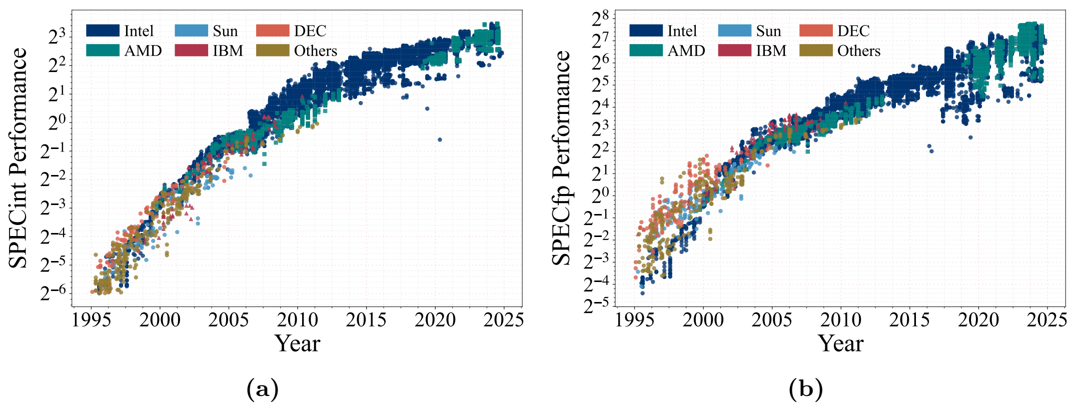


Figure 1.1: The normalized single-core performance scores of (a) integer and (b) floating-point SPEC CPU benchmarks on various processors over the last 30 years. Performance is measured using scores derived for each processor from the following SPEC CPU benchmarks: SPECint1995, SPECfp1995, SPECint2000, SPECfp2000, SPECint2006, SPECfp2006, SPECint2017, and SPECfp2017. The data is collected from spec.org [1].

in the post-Dennard era.

The widening performance disparity between microarchitecture simulators and the systems they model necessitates exploring alternative simulation techniques. Cycle-accurate full-system simulation, while invaluable for design verification and performance analysis, becomes increasingly time-consuming for multi-core architectures and heterogeneous CPU-GPU architectures. GPUs, characterized by their numerous execution units with a large number of threads, can lead to significant slowdowns when simulated on traditional CPUs [7, 16]. For example, the detailed simulation of SPEC CPU2017 benchmarks may take months to years, whereas that for the heterogeneous CPU-GPU applications in the SPEChpc 2021 benchmark suite may take decades, as shown in Figure 1.2. Sampled simulation is considered a sophisticated solution to making these extremely long simulation times tractable. This technique employs program analysis to determine representative regions of an application for detailed simulation. Sampled simulation methodologies exploit the well-established correlation between executed code and program performance,

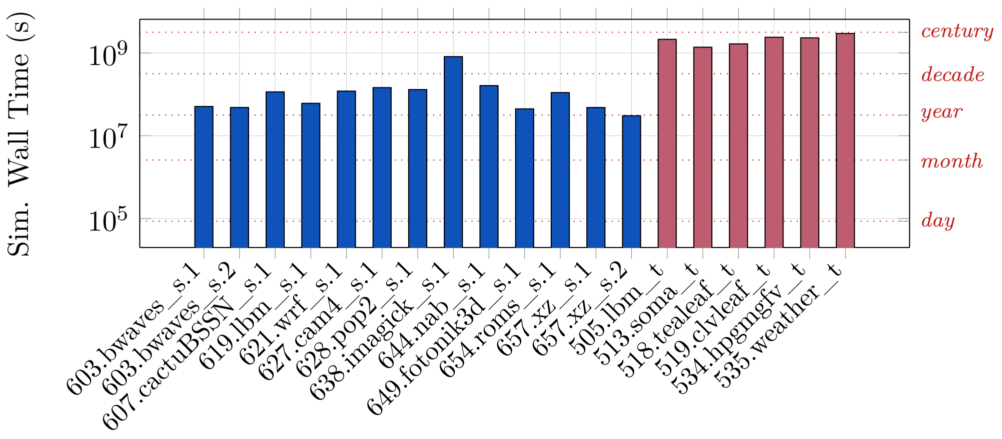


Figure 1.2: The estimated wall-clock times (in seconds) for the full simulation of multi-threaded (eight OpenMP threads) SPEC CPU2017 benchmarks and SPEChpc 2021 Tiny benchmarks (rank=1) using *Reference* inputs. The benchmarks were compiled with the Intel oneAPI toolchain. We assume the simulation speed of gem5 (CPU portion) and AccelSim (GPU portion) to estimate the wall times based on the instruction counts of the benchmarks.

as shown in prior research [17, 18].

While there are a number of solutions proposed for sampling single-threaded [2, 19, 20, 21, 22, 23, 24, 25, 26, 27], multi-program [28], and multi-process [29, 30] applications to accelerate simulation, these techniques are not deemed extensible for multi-threaded and heterogeneous workloads. Multi-threaded applications tend to synchronize the threads at certain points during execution and shared memory accesses, presenting a unique challenge [31]. This challenge is particularly evident in heterogeneous systems, where diverse compute units are closely integrated. In such cases, representing the amount of work done by the threads or compute units in terms of instructions per cycle (IPC), as shown to work for single-core performance, may lead to inaccurate measurements. It is also challenging to accurately capture or represent the execution pattern of the compute units, as the exact timing of each compute core can vary greatly.

Existing techniques for the sampled simulation of multi-threaded applications either do not provide significant speedups to be practical (Time-Based Sampling techniques [32, 33] can show less than 10× speedup as compared to fully-detailed simulation) or apply only to particular synchronization types (BarrierPoint [34] for barrier-based workloads). A solution is needed that both supports generic multi-threaded applications, irrespective of the synchronization primitives used, as well as allows for fast evaluation.

Most of the prior research on sampled simulation assumes the system to be static. However, modern hardware improves its performance and power efficiency by changing the hardware configuration, like the frequency and voltage of cores, according to a number of parameters such as the technology used, the workload running, etc. Techniques such as dynamic voltage and frequency scaling (DVFS) [35, 36, 37], dynamic cache reconfiguration [38, 39, 40], TurboBoost [41], etc., have been developed to adjust the hardware state in response to executed instructions and active processes. Additionally, dynamic scheduling techniques [42] have been developed for multi-threaded applications. To quickly estimate the performance of multi-threaded applications running on next-generation dynamic hardware and software, a sampled simulation methodology is needed that can dynamically adapt to changes in the system at runtime while accurately determining relevant performance metrics.

The profound increase in the demand for high-performance computing (HPC) resources in recent years has driven the widespread adoption of heterogeneous architectures, such as CPU-GPU systems [12]. However, evaluating the performance of these systems poses a significant challenge due to the lengthy simulations involved. While some efforts have addressed these challenges for specific workload classes [43, 44], they are often rigid with respect to region selection and can limit the overall simulation speedup when regions are large. Existing sampled simulation techniques for GPU kernels [16, 45, 46] may not represent an accurate performance estimate of the entire system in such cases. This highlights the need for techniques specifically designed for heterogeneous applications.

1.3 Simulation of Multi-core Systems

We aim to solve the challenges related to multi-threaded applications and propose a novel sampled simulation technique, which is both agnostic to the type of synchronization primitives used and scales by the similarity exhibited by the application. We proposed *LoopPoint* [8], a generic multi-threaded sampled simulation methodology that utilizes application loops to represent the amount of work done by the threads. LoopPoint combines several vital features, including (a) repeatable, up-front application analysis, (b) a novel clustering approach to take into account run-time parallelism, and (c) the use of loop-based simulation markers to divide the work into measurable chunks, even in the presence of spin-loops. LoopPoint chooses representative regions within a multi-threaded application that serve as checkpoints, allowing parallel simulation. These checkpoints can reproduce the performance of the original application and can significantly reduce simulation runtime compared to prior works.

LoopPoint considers loop-based regions demarcated by loop entries, allowing for repeatable regions. By monitoring the amount of work as represented by loops and not instructions or barriers, we can isolate multi-threaded application representatives and understand the amount of global work completed. LoopPoint enables synchronization-agnostic application sampling for multi-threaded workloads while still scaling the amount of work based on the representative nature of the application. The methodology has been adapted to widely used microarchitecture simulators like gem5, Sniper, etc., as well as in the industry. We released the representative checkpoints (as x86 executables or *ELFies* [47]) of a subset of SPEC CPU2017 benchmarks for the public to use.

While sampling techniques like BarrierPoint and LoopPoint improve the efficiency of microarchitectural simulations, the granularity of the identified regions may not be suitable for achieving comparable speedups at the RTL level. Recent works [48] attempted to adapt prior solutions like SimPoint [20] for RTL-level simulations on Verilator [49] using smaller region sizes aiming to improve simulation efficiency, which, however, resulted in accuracy that is typically not acceptable. The result is that it is currently infeasible to evaluate the performance of large workloads on the RTL level. While FPGA simulation infrastructures, such as Firesim [50], offer a faster alternative for simulation, FPGAs are specialized devices with inherent limitations in terms of memory capacity and processing units. Therefore, it is often not possible to fit large, realistic processor models on FPGAs.

This highlights the need for developing specialized workload sampling methodologies that can be flexibly applied to both microarchitecture-level and RTL-level simulations. These methodologies should support finer region granularities that align with the dynamic phase behavior exhibited by the application. Previously proposed workload sampling methodologies typically rely on fixed-length intervals for analysis, which can often be out of sync with the periodicity of program execution. Since an application's phase behavior [17, 51, 52] is strongly correlated to the code it executes, it can exhibit a hierarchy of phase behaviors that can be observed at various interval lengths, rendering conventional sampling techniques inadequate. By tailoring the sampling approach to capture the specific characteristics and phases of the workload, more accurate and efficient sampled simulations can be performed at both the microarchitecture and RTL levels. We proposed *Viper* to determine the simulation regions more systematically, which resulted in shorter simulation regions better suited for RTL simulations. Utilizing the innate program structures instead of fixed-length intervals allows for flexible region sizes that are more likely to be aligned with the application periodicity, thereby reducing the possibility of aliasing.

High-performance, multi-core processors are the key to accelerating workloads in several application domains. To continue to scale performance at the limit of Moore's Law and Dennard scaling, software and hardware designers have turned to dynamic solutions that adapt to the needs of applications in a transparent, automatic way. In such cases, profile-driven sampling methodologies may result in different performances for each execution. With this level of dynamism, it is essential to simulate next-generation multi-core processors in a way that can both respond to system changes and accurately determine system performance metrics. Currently, no sampled simulation platform can achieve these goals of dynamic, fast, and accurate simulation of multi-threaded workloads.

We proposed *Pac-Sim*, which is designed for fast and efficient simulation of multi-threaded applications without the need for any up-front application analysis and allows for the simulation of dynamically scheduled multi-threaded applications even in the presence of runtime hardware events – this was not possible with previously proposed sampled simulation methodologies. Pac-Sim includes an online sampling and decision-making phase based on predictions that rely on previously executed code, thereby completely eliminating the need for offline profiling. We incorporate application analysis to guide sampled simulations, similar to SimPoint-like [20] methodologies but without the need for upfront pre-processing, as seen in SMARTS-like [2] methodologies. Pac-Sim makes intelligent simulation decisions through online learning and implements lightweight online profiling, clustering, and warmup techniques for optimal performance. Moreover, the proposed methodology can accommodate hardware state changes, software features, and other factors that affect simulation results.

1.4 Simulation of Heterogeneous Systems

The prevalence of CPU-GPU architectures in heterogeneous computing arises from their ability to address the evolving demands of modern workloads, coupled with their well-established programming models. GPUs have emerged as the most widely used general-purpose accelerators in modern data centers [53] and supercomputers [54] that accelerate massively parallel big data analysis [55, 56] and machine learning [57, 58] workloads. While previous works have investigated characterizing workloads that consist of CPU components [2, 20, 30, 32, 34] and GPU [45, 46, 59, 60] components independently, as well as their comparative analyses [61], hybrid solutions that support analysis and workload reduction for multiple types of heterogeneous workloads, from CPUs, GPUs, and even custom hardware accelerators (like FPGAs), have not yet been investigated. Given the importance of these workloads, from HPC systems to data center use, simulation of heterogeneous workloads is key to understanding the interactions between compute components and how their interactions can affect overall runtime performance.

We proposed *XPU-Point*, a unified sampling methodology for heterogeneous workloads that can accurately (a) understand the workloads running on heterogeneous systems to (b) build a representative sample for the fast and accurate performance analysis of the workloads. We also (c) estimate the accuracy of the proposed sampling methodology. XPU-Point proposes a comprehensive methodology for the sampled performance evaluation across a broad spectrum of real-world workloads, from scientific simulations to artificial intelligence on heterogeneous CPU-GPU architectures. This enables computer architects and performance researchers to quickly estimate the performance of long-running, heterogeneous workloads using sampled simulation, which was not possible before. While the primary focus of XPU-Point is to enable sampled microarchitecture simulation, its methodology can be adapted to broader performance analysis and characterization of several classes of heterogeneous workloads.

1.5 Validation of Selected Sample

Workload sampling can significantly speed up the simulation performance, assuming the regions of interest (ROIs) or the representative sample found can be proven to accurately represent the behavior of the full workload. One standard way to validate the representativeness [62, 63, 64] of the ROIs is to measure the sampling error, which is the difference in the performance of the full workload and the extrapolated performance using the ROIs. The performance is typically obtained through detailed simulations [65]. However, the simulation of long-running workloads is infeasible, taking months to years.

We propose *ROIperf*, a framework that validates the ROIs selected using workload sampling methodologies. ROIperf leverages the native hardware performance counters [66] by evaluating both the full workload and its representative regions on real hardware systems. This approach ensures the validation of ROIs through the performance measurement on real hardware instead of simulation. The methodology is particularly beneficial for long-running programs for which the prevailing simulation-based validation techniques are infeasible. While this technique does not allow for the performance estimation of future hardware (where timing simulation is needed), this path enables one to evaluate if the selected ROIs are representative and, therefore, can be used to determine the overall performance characteristics of the workload accurately. We demonstrate the efficacy of ROIperf by evaluating various sample selection methodologies across a wide range of workloads. ROIperf achieves a significant speedup in validating regions selected for simulation.

Table 1.1: Table summarizes the methodologies proposed in this thesis. We categorize the methodologies into two main groups: *Sample Selection* and *Validation*. The table also identifies the *Analysis Type* used by each methodology. Notably, some methodologies require an upfront analysis or profiling phase to extract application-specific characteristics. Additionally, the table indicates the primary applicability of the methodology.

	Methodology	**Analysis Type**	**Primary Applicability**
Sample Selection	LoopPoint	○	Statically scheduled multi-threaded applications
	Viper	○	Multi-threaded applications on RTL-level simulators
	Pac-Sim	●	Dynamically scheduled multi-threaded applications
	XPU-Point	○	Heterogeneous CPU-GPU applications
Validation	ROIperf	◐	Single-threaded and multi-threaded applications

● Online Profiling ◐ Offline Analysis ○ Offline Profiling

1.6 Thesis structure

The rest of this thesis is structured as follows. The background and prior work on performance evaluation techniques, sampling, and simulation are reviewed in Chapter 2. The next five chapters present the primary contributions (summarized in Table 1.1) of the thesis. We present the motivation, methodology, and results of the *LoopPoint* sampled simulation methodology in Chapter 3. We introduce *Viper* methodology, which enables faster RTL-level simulations, in Chapter 4 and present our live sampled simulation methodology, *Pac-Sim* in Chapter 5. In Chapter 6, we present the *XPU-Point* methodology for the accurate sample selection in heterogeneous CPU-GPU workloads. We introduce a novel technique *ROIperf* to validate the regions of interest identified by workload sampling methodologies in Chapter 7. Finally, we conclude the thesis in Chapter 8 with a summary of thesis contributions and present the future directions.

Chapter 2

Related Work

The purpose of computing is insight, not numbers.

— Richard Hamming

In this chapter, we will review the related work for contributions to this thesis.

2.1 Workloads and Analyses

The SPEC CPU benchmarks suite is the de facto benchmark for evaluating the performance of processor designs. It includes a wide variety of applications, such as 3D rendering, biomedical imaging, and electronic design automation. These benchmarks are designed to intensively exercise different aspects of a processor, including the front-end (instruction fetch, branch prediction), the back-end (retirement, function units), and the memory subsystem (cache hierarchy, prefetching). These benchmarks have large dynamic instruction counts in the order of trillions of instructions, which severely affects the simulation time of detailed performance simulators. Nair et al. [67] studied the phase behavior of SPEC CPU2006 and SPEC CPU2000 benchmarks and identified simulation points using SimPoint. Similarly, the single-threaded version of SPEC CPU2017 has been studied [68, 69]. Several workload characterization techniques are proposed to categorize benchmarks into subgroups that exhibit similar behaviors. Hoste et al. [70] demonstrated that workloads may behave similarly on one microarchitecture but drastically different on others, motivating the need to profile benchmarks in microarchitecture-independent ways. They proposed characteristics such as the register-dependent distance, branch predictability, instruction mix, and data stream working-set size to capture the intrinsic software behaviors. Shao et al. [71] further introduced ISA-Independent characterization, pointing out that specialized architectures are unconstrained by the conventional instruction set semantics that are presumed by the microarchitecture-independent characterization. They showed that their techniques are useful to rule out ISA-dependent characteristics such as the register spilling effect and provide insights for hardware specialization. Alameldeen et al. [72, 73] demonstrated the problems of non-determinism with multi-threaded workloads. They showed that small timing variations in the software or the operating system can affect the process scheduling and execution path of the program, resulting in false conclusions during design explorations. Alameldeen et al. [31] also demonstrated that IPC can be a poor performance indicator, leading to inaccurate estimation of speed-ups in terms of the actual run time of the programs as they may not reflect useful work done. For example, spin-lock loops can contribute to misleading IPC increase, and enhancements to instruction sets can decrease IPC even as performance improves.

2.2 Characterizing Program Execution

A basic block is a sequence of instructions that has single entry and exit points with no branches or jumps within the sequence. A basic block vector (BBV) is a data structure that represents a set of basic blocks, storing counts for each executed basic block, and forms a fingerprint of a region's execution. It provides a compact representation of the program's control flow. Typically, BBVs are collected at regular intervals during the program execution. Each of these BBVs represents a region of an application that correlates to region performance [18]. BBVs provide information about how the program execution behavior changes over time.

LRU stack distance is the number of distinct cache accesses between consecutive accesses of the same data item [74]. LRU stack distance vectors (LDVs) are data structures that are used to keep track of the LRU stack distances. LDVs consist of integers associated with each cache line, representing the number of cache lines accessed between the current cache line and its most recent access. Shen et al. [75] showed that LDVs can be used to characterize program behavior. While BBVs focus on analyzing control flow patterns, LDVs provide insights into memory access patterns and cache behavior. By combining BBVs and LDVs, a more comprehensive understanding of program behavior can be achieved [34].

2.3 Sampling Single-threaded Workloads

SimPoint [20] uses basic block vectors (BBVs) as unique signatures to represent instruction streams with fixed or variable length intervals based on the fact that code sections that perform similar workloads should traverse similar sequences of basic blocks. The BBVs are then clustered using the k-means clustering [76] algorithm to identify the number of phases within the application. A representative region is selected from each cluster that is assigned a weight proportional to the number of regions that belong to the cluster. The SimPoint methodology was extended to support x86 applications in PinPoints methodology [24, 77] using Pin for the BBV generation. However, SimPoint did not account for performance differences between similar instruction streams due to micro-architecture and hardware differences, such as the cache states and clock frequency changes, let alone thread interactions in multi-threaded workloads. SMARTS [2] proposed a systematic sampling framework that simulated programs by alternating among fast-forward, warm-up, and detailed simulation phases and obtaining IPC samples for each detailed simulation. The program IPC can then be estimated with high confidence using statistical methods. However, it can only be used to estimate the overall IPC of the program, but not the IPC trace throughout the program execution. Another work on software phase markers [26] uses loops to determine simulation regions but is limited in that they only provide support for single-threaded applications using phase markers denoting phase changes. LiveSim [27] is another simulator that uses statistical sampling with confidence levels to estimate IPC. They extended the framework by using in-memory checkpoints at sample regions to enable interactive simulations. pFSA [78] uses hardware virtualization to spawn processes that fast-forward to regions of interest (ROIs) at near-native speed and perform detailed simulations in parallel. They also proposed a novel cache warm-up technique based on estimating the error induced by insufficient cache warming.

2.4 Sampling Multi-threaded Workloads

Ekman et al. [79] propose a methodology to reduce the number of simulation points using a matched-pair comparison method to estimate the full application performance. SimFlex [30] extends SMARTS methodology to support multiprocessor applications with an increased sample length. SMARTS and SimFlex use random sampling, and therefore, the samples are not necessarily representative. Perelman et al. in [52] extend the Simpoint methodology to use for phase analysis of multi-threaded workloads. COTSon [80] targeted the full software stack and complete hardware models to ensure both high performance and accuracy. Time-based sampling methodologies [32, 33] introduced a generic simulation framework for multi-threaded applications based on the progressed time rather than instruction count. However, this bounded the total simulation time to the length of the program execution, not the structures of the program. Both SimFlex and Time-based Sampling did not exploit knowledge from the software, such as barriers, tasks, and loops, which allowed us to break down programs into representative regions in an informed way. BarrierPoint [34] and TaskPoint [43] leveraged the structures in multi-threaded programs by using barrier synchronization primitives and tasks in the task programming paradigm as the units of work, respectively. This allowed automatic identification of regularities in the software because of the intentions of these software primitives. However, these methods rely on particular programming paradigms, which limits their generalities.

2.5 Sampling GPU Workloads

GTPin [81] is an ahead-of-time (AOT) instrumentation tool for workloads that run on Intel GPUs. In AOT instrumentation, the binary is modified to insert monitoring or profiling code before the actual execution. Leveraging GTPin, Kambadur et al. [59] proposed a solution to sample workloads running on Intel GPUs. They utilize kernel names, arguments, and basic block entries to select representative regions of the GPU programs at a kernel-level granularity. Yu et al.[82, 83] propose a SimPoint-like strategy to detect representative loops that can be used to extrapolate kernel performance. TBPoint [45] uses BBVs and other kernel-specific features to identify representative kernels, whereas Principal Kernel Analysis (PKA) [46] monitors the IPC difference between sampling units to determine the regions to fast-forward. Both TBPoint and PKA enable the sampled simulation of GPU workloads at both the inter- and intra-kernel levels. Sieve [60] extends on prior works to show that using the kernel name and instruction count allows for better sample selection. Photon [16] utilizes GPU Basic Block Vectors (BBVs) for inter-kernel and intra-kernel workload sampling, resulting in significant improvement in sampling accuracy compared to previous approaches.

2.6 Analytical Modeling

Eyerman et al.[84] proposed a model to divide the dynamic instruction stream into long-latency miss events, such as branch mispredictions and cache misses that limit the scope of Out-of-Order behaviors. Each interval of the instruction stream can then be characterized by an analytical latency model based on interval length and latency type, and the overall performance can be reconstructed from these discrete intervals. RPPM [85] used multiple performance indicators such as the number of instructions, branch entropy, and long-latency loads to project single-threaded performance. It also takes into account synchronization overheads by identifying critical paths to project multi-threaded performance. However, the fixed mathematical formulation prohibited the opportunity to estimate the performance of future hardware that has not been seen before, which requires different analytical formulations. Statstack [86] is proposed to estimate the cache miss ratio of a fully associative cache with LRU replacement policy. It uses reuse distance samples (RDS) - the number of memory references between successive same cache line accesses - to estimate the stack distance distribution. This allows the cache miss ratio to be accurately estimated for a wide range of cache sizes and programs. Linear branch entropy [87] is proposed to model the branch miss rate of any branch predictor by finding a linear relation between the branch entropy and the sampled branch miss rate per configuration of a branch predictor.

2.7 Warmup Techniques

There are primarily three kinds of warmup techniques: statistical warming, checkpoint-based warming, and functional warming. Statistical warming techniques reconstruct the cache state by collecting all the memory access information. For example, Memory Reference Reuse Latencies (MRRLs) [88] records the number of instructions between consecutive references to each unique memory location. Similar to MRRLs, Memory Time-stamp Record (MTR) [5] and Boundary Line Reuse Latency (BLRL) [89] also choose to record memory access information but using different methods. MTR records the snapshot of memory reference patterns, while BLRL considers reuse latencies across the boundary line of the pre-sampled and the sampled regions. Unlike prior works, DeLorean [90] collects only a selected number of key reuse distances to speed up the statistical warming. CoolSim [91], on the other hand, uses virtualized fast-forwarding to speed up the performance of collecting memory reuse information. Memory Hierarchy State [92] is a checkpoint-based warmup technique that saves the state of all the major microarchitecture components into a touched memory image (TMI) that decreases the cost to load and store this data.

2.8 Simulation Infrastructures

Gem5 [6] is a cycle-accurate simulator that models CPU pipelines and cache protocols in fine granularity. However, running large multi-core benchmarks is slow due to their detailed models. Sniper [14] and ZSim [93] are fast multi-core simulators that use binary instrumentation to speed up functional simulations. Sniper pipes program execution traces from the binary instrumentation to its modeling backend for detailed modeling of various micro-architecture components. This gives Sniper high modularity and flexibility for employing different modeling strategies at different granularities. ZSim proposed a two-phase parallelization technique to speed up simulation and used user-level virtualization to enable fast simulation for thousands of cores. While ZSim can provide high simulation throughput, instruction schedules need to be computed up-front, and apart from memory hierarchy changes, it does not provide the capability to adjust these schedules at run time. RTL-level simulators are used to simulate and verify the behavior of digital circuits described at the Register Transfer Level (RTL). Among the most widely used RTL simulators are Verilator [49] and VCS [94].

There are several GPU simulators and heterogeneous CPU-GPU simulators available. GPU simulators are extremely slow [7] as compared to the real execution, as they run on the CPU, which typically has fewer cores than the GPU being simulated. Trace-driven GPU simulators, such as MacSim [95], execute functionally generated traces with a timing model to generate the performance results. Execution-driven GPU simulators, such as Multi2Sim [96], gem5-gpu [97], MGPUSim [98], gem5 APU [99, 100] and GPGPUSim [101], directly execute the binary for performance simulation. Simulators like Accel-Sim [7] and NVArchSim [102] support both execution- and trace-driven simulation modes. Among these simulators, Multi2Sim, gem5-gpu, MacSim, and gem5 APU support the simulation of heterogeneous CPU-GPU workloads.

2.9 Synthetic Workload Generation

Synthetic workload generation techniques involve creating lightweight workload clones that mimic the behavior of real-world applications. These synthetic clones are typically used in studies related to performance evaluation and benchmarking. MAMPO [103] is a multithreaded synthetic power virus generation framework targeting multicore systems. It uses a genetic algorithm to search for the best power virus for a given multicore system configuration. SynchroTrace [104] is a trace-based multi-threaded simulation methodology that accurately replays synchronization- and dependency-aware traces for chip multiprocessor systems. SynchroTrace achieves this by recording synchronization events and dependencies in the traces, allowing for the replay on different hardware platforms. GPGPU-Minibench [83] captures the execution behavior of existing GPGPU workloads in a profile, which includes a divergence flow statistics graph (DFSG) to characterize the dynamic control flow behavior of a GPGPU kernel. G-MAP [105] statistically models the GPU memory access stream locality by considering the regularity in code-localized memory access patterns and the parallelism in the execution model to create miniaturized memory proxies. Mystique [106] is yet another technique that generates benchmarks from production AI models by leveraging PyTorch execution traces. Ditto [107] focuses on synthesizing workloads for data centers mimicking traditional CPU performance behaviors, like branch mispredictions, cache miss rates, and IPC. However, these techniques may not be applicable to all workload studies involving cache compression or prefetching.

2.10 Checkpointing Techniques

Checkpointing is a technique used to save the state of a system at specific points in time. Architectural checkpoints preserve the software-visible state, including the register files of cores, memory, and I/O device states. Widely used emulators like QEMU [108] and Simics [109] are used to maintain these states. Microarchitectural checkpoints capture the states of components such as pipelines, caches, branch predictors, and TLBs.

ELFies [47] are user-level executable checkpoints for regions of interest. Extracted from deterministic replays of the full-program recording used for profiling, ELFies inherently capture the initial state of the ROI as observed during profiling. This significantly reduces the impact of non-repeatability, as it only affects the native execution of the ELFie itself, not the entire program execution leading up to and within the ROI. However, imprecise reconstruction of the operating system state during ELFie creation can lead to system call failures or unpredictable execution behavior. These discrepancies can introduce deviations from the original execution and potentially cause application failures.

MINJIE [48] is an open-source platform that integrates a set of tools for pre-silicon validation, verification, etc. MINJIE provides an instruction set interpreter/emulator called NEMU, which is used for checkpoint generation. The checkpoints are restored later to simulate them in parallel on Verilator [49].

Chapter 3

LoopPoint: Checkpoint-driven Sampled Simulation for Multi-threaded Applications

Truth... is much too complicated to allow for anything but approximations.

— John von Neumann

In this chapter, we introduce a novel sampling technique for multi-threaded applications called LoopPoint, which is both agnostic to the type of synchronization primitives used and scales by the similarity exhibited by the application. LoopPoint combines several vital features, including (a) repeatable, up-front application analysis, (b) a novel clustering approach to take into account run-time parallelism, and (c) the use of loop-based simulation markers to divide the work into measurable chunks, even in the presence of spin-loops.

3.1 Introduction

Sampling is a well-known application workload reduction technique that traces its roots back decades. From the earliest works [2, 20], researchers have been able to identify regularity in single-threaded applications and exploit that to sample large applications into

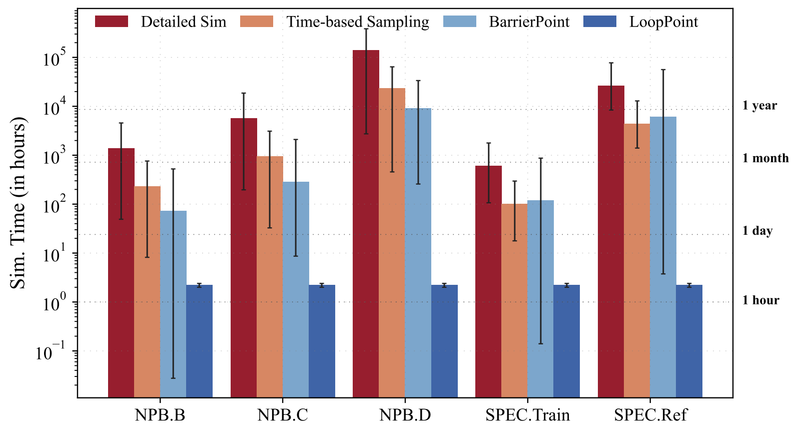


Figure 3.1: Approximate time to evaluate the performance of multi-threaded benchmarks with different methodologies. *The average result and error bars represent the estimated simulation time for all benchmarks in the corresponding suite and input sets, assuming infinite simulation resources (the longest simulation region determines the overall simulation time). Benchmarks were configured with 8-threads and passive OpenMP wait policy, assuming a total simulation speed of 100 KIPS.*

smaller application representatives. Because of the repeated execution of regions with similar behavior, these techniques have been shown to accurately predict the original workload behavior, and significantly reduce the simulation time needed [2, 20].

Apart from sampling, researchers have developed a number of complementary techniques to reduce the overall amount of work required to simulate applications in detail, including input size reduction [110] and benchmark synthesis [111]. While each technique presents its benefits and challenges, sampling has emerged as a straightforward way to maintain the original application characteristics and accurately extrapolate performance while reducing the overall simulation burden.

With the increasing number of cores in modern processors, multi-threaded applications can exploit a large amount of compute through task and loop parallelism. Simulating these large, multi-threaded applications is extremely difficult, even on modern simulators. Ultra-fast FPGA-based simulators [50] require detailed implementations and are capacity-limited, preventing the simulation of large processors and large parallel systems, and fast software-based simulators [14, 93] still require a significant amount of time to run an entire large, parallel workload to completion. Multi-threaded applications are inherently difficult to analyze [73] as the threads can go to sleep at any time, threads interfere with one another, and complex behavior emerges from regular application parameters like misalignment of threads to cores and unequal cache distribution.

Some of the earliest multi-threaded sampling solutions prove effective when the threads themselves do not synchronize but can still interact with the memory hierarchy [30]. Any amount of synchronization requires thread progress to be measured in time to track the amount of progress or parallelism in the application. The move towards a time-based sampling methodology has led to the development of sampling techniques for synchronizing multi-threaded applications. These techniques [32, 33] describe one of the first generic sampling solutions for multi-threaded applications. However, the overall simulation speed is still bound to the total application length, which dominates the simulation time of this methodology. Later proposals, in the form of application and synchronization-specific methodologies [34, 43, 44], exceeded the performance of time-based sampling and allowed for the simulation complexity to be bound to application diversity, not application length. Unfortunately, these methodologies are tied to specific application characteristics (the use of barriers [34] or tasks [43, 44]), and therefore do not represent a general sampling solution that covers all application types. In fact, as Figure 3.1 demonstrates, both time-based sampling, and BarrierPoint (when inter-barrier regions exist to simulate), **approach a simulation time of one year to simulate the sample** when considering large, multi-threaded applications. Clearly, current methodologies are insufficient for simulating the largest, most realistic benchmarks like the multi-threaded SPEC CPU2017 with the `ref` input set.

In this work, we aim to overcome the limitations of these prior works to enable synchronization-agnostic application sampling for multi-threaded workloads while still scaling the amount of work based on the representative nature of the application. To accomplish this goal, we present the *LoopPoint* methodology that reduces an application to a few representative regions, called *looppoints*, by taking into account several key factors like understanding (1) *where to simulate* which requires (1a) an accurate analysis methodology that can provide for reproducible analysis, and (1b) using a precise clustering mechanism that partitions the regions to reduce the workload into its representative components. In addition, our methodology presents (2) *how to simulate* the regions to allow the application to take advantage of the underlying hardware, while not constraining execution to a deterministic path [72] that might not exhibit true application behavior.

We make the following contributions in this work:

1. A representative simulation region selection methodology called *LoopPoint* suitable for the performance projection of multi-threaded programs (more details on supported workloads in Section 3.3.11) based on using loop iterations as the unit of work.

2. A technique to enable multi-threaded sampled simulation by filtering out spin-loops during region identification, selecting repeatable loop boundaries of a practical region size, and accurately extrapolating performance characteristics.

3. The development of a process to record a constrained application checkpoint for accurate analysis and subsequently simulate the workload's unconstrained behavior during simulation.

4. A comprehensive evaluation of the LoopPoint methodology to demonstrate the potential for speedup while maintaining accuracy using the OpenMP-based multi-threaded subset of SPEC CPU2017 benchmark suite and NAS Parallel Benchmarks (NPB).

In the following sections of this work, we first provide an overview the LoopPoint methodology, results, and evaluation. In Section 3.2, we detail each of the components needed for a fast, accurate, and generic multi-threaded sampled simulation. In Section 3.3, we describe the LoopPoint methodology. We then detail the experimental infrastructure and setup in Section 3.4, evaluate the LoopPoint methodology in Section 3.5. Finally, we compare to related work (Section 3.6) and conclude the chapter (Section 3.7).

3.2 Fast and Generic Multi-threaded Simulation Requirements

Time-based sampling methodologies [32, 33] present the first workable solution to sample generic multi-threaded applications. However, the speed-ups achieved (up to 5.8×) using these methodologies are limited by the need to visit the entire application. To achieve high speed-up while maintaining accuracy during multi-threaded workload sampling, we need to consider the inherent application regularity and the amount of parallelism present in the workload at any particular time. We need to define a unit-of-work that is suitable to exploit the application regularity and, at the same time, is applicable across a variety application and synchronization types. The key is the ability to (1) recognize representative regions in a generic way across multi-threaded workload types, and to (2) classify these regions considering application parallelism. To this end, we present a new application sampling methodology called *LoopPoint* that (a) uses loop iterations as the main unit of work, (b) utilizes constrained *pinballs* [77] (user-level checkpoints that allow for reproducible analysis), (c) employs heuristics to remove synchronization during analysis, but use them during simulation, and (d) performs unconstrained simulation of the selected simulation regions allowing for fast and accurate workload evaluation. Figure 3.2 shows the overall methodology.

Sampling methodologies that rely on instruction counting can perform poorly when dealing with multi-threaded applications [31]. We demonstrate this by performing a

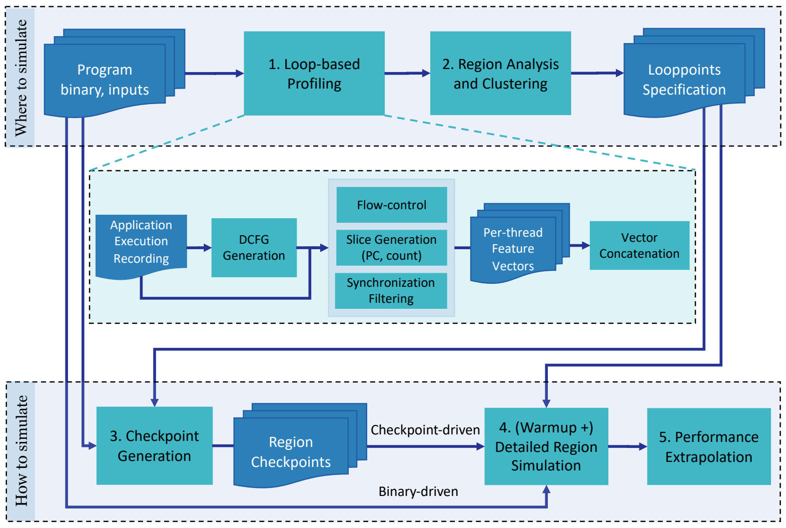


Figure 3.2: LoopPoint-based region selection and simulation for multi-threaded workloads. The workload is captured for analysis and region selection based on loop information. The representative regions are simulated using a checkpoint-driven method as well as by binary-driven unconstrained way allowing for extrapolation of performance and other metrics of interest.

naive adaptation of Simpoint [20] for multi-threaded applications of SPEC CPU2017 that use eight threads. With this methodology, the average error in predicting the runtime of the applications using active wait policy is 25% and as high as 68.44%, whereas errors for the passive wait policy are as high as 20%.

Previous works like the BarrierPoint [34] methodology use inter-barrier regions as the unit of work, whereas the TaskPoint [43] methodology applies only to task-based applications that use task instances as the unit of work. Unfortunately, BarrierPoint, when used to sample large applications with a small number of barriers, can yield negligible simulation speed-ups. This can be common, especially while sampling realistic workloads for which the length of inter-barrier regions is a bottleneck. BarrierPoint, therefore, is not practical for such workloads. Figure 3.1 shows how the instruction count (and therefore simulation time) of an inter-barrier region grows with larger input sets of SPEC CPU2017 and NAS Parallel Benchmarks (NPB) [112] with eight threads. BarrierPoint works well for NPB with the A input size [34], but as the input sizes grow, for classes C, D and E, inter-barrier regions become so large that it becomes impractical to use BarrierPoint for those input sets. The same is the case with SPEC CPU2017 using `ref` inputs.

Instead, LoopPoint uses loop iterations as the unit of work with the goal to apply to generic multi-threaded programs. The idea of using loop iterations as slices for single-threaded programs was proposed in [26]. With loop entries as slice boundaries, the simulation regions can then be specified using a *(PC, count)* pair for the starting and ending loop entry for each simulation region. By monitoring the amount of work, as represented by loops, and not instructions or barriers, we can isolate multi-threaded application representatives and understand the amount of global work completed. For multi-threaded programs, one additional constraint is that the loop entries that are chosen to start and end slices should be those doing meaningful work. Automatically separating loops doing real work from synchronization can be a daunting task. However, we can use application knowledge or synchronization mechanism details to filter out synchronization loops. For example, the Intel OpenMP run-time uses functions in the `libiomp5.so` library for synchronization; hence loops from that library should not be counted towards *work done* while profiling the application. Alternatively, if the synchronization routines are known before-hand, the code from such routines can likewise be avoided.

Where to simulate. As detailed cycle-accurate simulation can be time-consuming, architects and researchers often use sampling to decide where to simulate by choosing small portions or regions of long-running program executions for simulation. Sampling requires (a) choosing the regions so that they are representative of the whole program behavior and (b) projecting the whole-program performance based on the simulation results of the selected regions. SimPoint [20] is a popular simulation region selection approach. It works by dividing the program execution into smaller slices and collecting an execution signature for each slice. *K-means* clustering is used to determine phases from slice signatures. One representative per cluster is then chosen with the weight corresponding to the cluster size. Since these representatives are designed to be micro-architecture independent, the signature collected for each slice needs to be dependent only on the program execution and not based on any micro-architecture dependent metric. Typical signatures used include the *BBV* (Basic Block Vector) which contains execution counts of various *basic blocks* (single-entry/exit code blocks). How to slice a program's execution into regions is an important decision. For single-threaded programs, using a fixed instruction count called the *slice size* has been shown to work well [20]. In our work, we keep slices of approximately similar sizes demarcated by loop entries. The region selection is based on the replay of a previously recorded whole-program execution as a pinball. According to the micro-architecture of the recording machine, the synchronization seen there can be different from the synchronization seen during unconstrained simulation. We, therefore, augment our region selection methodology to make a selection only on the real computation or work done. The heuristics described earlier to avoid synchronization loop entries as region boundaries can also be used to filter out (to execute but not count) synchronization code during profiling for region selection.

How to simulate. A critical decision that the simulator developers need to make is how to simulate, i.e., how to connect the application in consideration to a simulator. The most commonly used methods are (1) *binary-driven* where a program binary is executed during simulation feeding instructions to the simulator, (2) *checkpoint-driven* where a snapshot of selected region memory/register state and a list of injection events are used to drive the simulator, and (3) *trace-driven* where an instruction-by-instruction recorded state is fed to a timing-only simulator. The choice of how to simulate depends on several factors, such as ease of deployment, cost of generation, and flexibility of the evaluation. For this work, we use both binary-driven and checkpoint-driven simulations for our evaluation, although the implementation itself is generic and supports any of these simulation methods. Checkpoints are easier to share among multiple users than program binaries whose execution might require complex setup and input availability. We propose to capture regions selected by LoopPoint as pinball [3] checkpoints so they can be used to drive PinPlay-based simulators.

By default, PinPlay supports *constrained* replay of pinballs where the shared memory accesses among threads are repeated in the order captured during recording. Simulation based on such constrained replay will repeat the thread ordering based on the micro-architecture of the machine on which the pinballs were generated. However, we ideally want the target, simulated micro-architecture to decide the thread behavior during simulation. To achieve that, we also use binary-driven simulation of the regions selected by LoopPoint using stable (PC, count)-based boundaries defining those regions. Therefore, the simulation proceeds as though the region was executed natively on the simulated micro-architecture. Another technique to achieve unconstrained simulation using pinballs is to convert them to executable checkpoints, called ELFies [47].

3.3 The LoopPoint Methodology

In this section, we explain the different parts of the proposed methodology, LoopPoint. We start with an upfront analysis of the application to determine its behavior and to identify loops, as shown in Figure 3.2. This is a one-time step and we use the information collected here for clustering regions to choose representatives. The representative regions are then simulated with sufficient warmup. The simulation results enable us to reconstruct the overall application performance.

3.3.1 Selecting a Unit of Work

Multi-threaded applications may use different execution paths with different runs, and therefore the use of IPC to evaluate the performance of multi-threaded workloads is infeasible [31]. LoopPoint proposes a strategy that identifies regions of interest in terms of work done by each thread. We define the unit of work as the actual amount of compute done within a slice of an application. For an unmodified application with the same input set, the unit of work chosen needs to remain the same for each application execution regardless of the properties of the underlying hardware, although the number of instructions executed may vary each time. The generality of the chosen unit of work is crucial for application sampling as this determines the amount of simulation speedup achieved. We would want the chosen unit of work to be large in number within the program, to be one that repeats itself, and to remain unchanged over multiple executions.

We consider the number of loop iterations as the unit of work done. Program loops are ubiquitous across application domains and the number of iterations of any particular loop doing real computation as opposed to synchronization can remain constant over multiple executions for an unmodified application and for a fixed input size. In a multi-threaded environment, we consider loop execution, ignoring spin-loops (one form of active synchronization), to compute the amount of work done. Spin-loops contribute to the IPC of the application and consume CPU cycles, however, they do not contribute to the meaningful work done by the particular thread (waiting cannot be considered work completed). This is the key to LoopPoint methodology we present here.

3.3.2 Understanding Parallelism

One of the fundamental requirements of a multi-threaded sampling methodology is the ability to understand how the parallelism of an application changes, over time, and to use that information to drive the representative selection process. In fact, understanding parallelism in a generic way is one of the main insights of this work. To accomplish this, we continue to use worker loop instructions as the key metric for work completed.

Program phase behavior is an important aspect to consider while sampling applications. A phase is a set of slices in a program's execution that shows similar behavior, regardless of where they appear within the execution. The locations in source code whose executions correlate to a phase change in the application are called software phase markers [26]. The software phase markers can accurately identify the phase changes that occur in an application execution irrespective of the underlying microarchitecture. These are execution points that can act as simulation region boundaries that are invariant across multiple application executions. We identify source-level program loops as possible checkpoints which form the basic building blocks of a program.

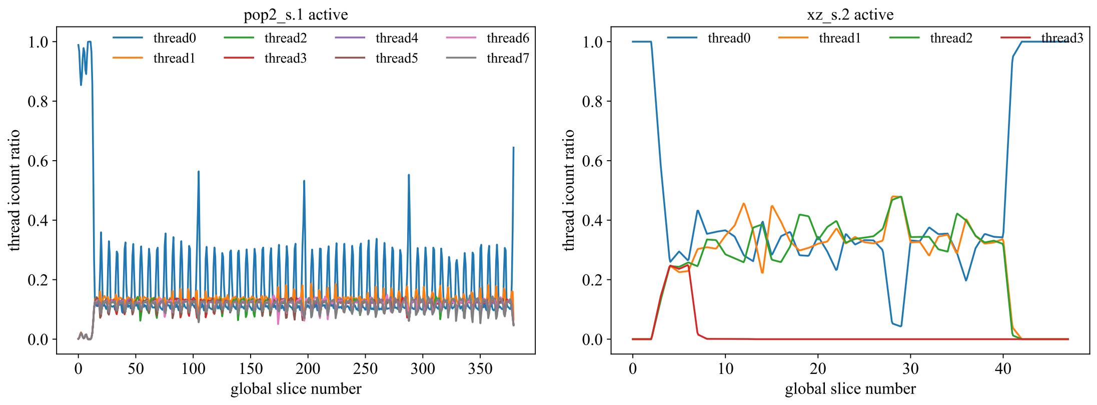
Capturing BBVs is an essential way to understand the fingerprint of an application execution region. We consider the slice-size to be approximately $N \times 100$ million global (all-threads) instructions that align with loop boundaries for a N-threaded application. For example, we collect BBVs in intervals of approximately 800 million instructions for an 8-threaded application. We ignore the instructions executed in spin-loops or any other synchronization code while collecting the BBVs. The end of a region specified by a BBV is the next loop entry once the instruction count target is achieved. Although this can be implemented in several ways (as described in [26]), we do not currently differentiate between inner and outer loop markers and do not restrict specific threads to indicate loop boundaries. The loop entries that serve as region markers need to be worker loops and not spin-loops. We assume that the spin-loops are found only in the synchronization library (for example, OpenMP), and therefore, we end a region only at a loop entry that is present in the main image of the application. The per-region BBVs of each thread are concatenated into a longer, global BBV that represents a multi-threaded region. This guides the clustering phase when there are regions that exhibit non-homogeneous thread behavior. Figure 3.3 shows the ratio of the number of instructions executed by each thread as the application progresses. The application `657.xz_s.2`, as an example, clearly exhibits a non-homogeneous thread behavior.

There are a number of reasons to maintain sufficiently large per-thread slices (approximately 100 million instructions). If a smaller slice-size is chosen, a large number of simulation points may be required, and such regions are highly sensitive to warmup and aliasing issues [32]. At the same time, we also need to make sure that there are enough intervals in the application for the clustering algorithm to work efficiently [23]. Prior analyses [20] on single-threaded applications showed that fixed size (of 100 million instructions) intervals of execution can be used to identify phase behavior. Using varying length intervals [25] corresponding to the application periodicity can help mark the phases more accurately. In LoopPoint, we use approximately similar interval lengths, however, the methodology can also be used with varying length intervals.

While we profile an application for BBVs or any feature vectors, we make sure that all threads in the application make the same amount of forward progress during analysis. This is to stabilize the collected profile for any thread imbalance that is caused by external events on the host processor (and is unrelated to the analysis environment). We call this method to enforce equal progress between threads flow-control.

3.3.3 Marking Region Boundaries

Every region in an application has its boundaries at a loop entry. The regions need to be represented so that it is repeatable across multiple executions of the application. In the case of single-threaded applications, instruction count can be used to define regions reliably. However, for multi-threaded applications, this does not hold. We describe the start and end of each region as an ordered-pair (PC, count), where the *PC* is the address of the corresponding region boundary marker instruction and the *count* is the



(a) **(b)**

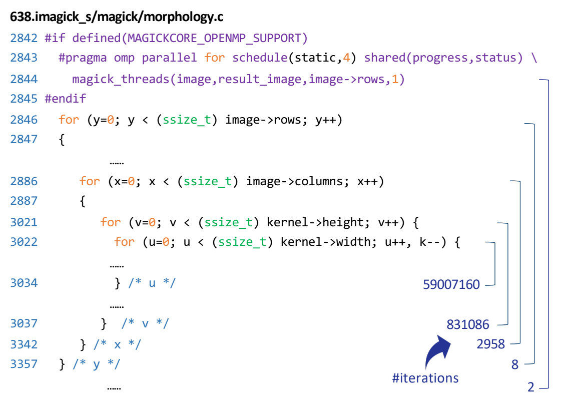
Figure 3.3: The above graphs show the variation in the share of the per-thread instruction count on a per-slice (with a slice size of 800M global instructions) basis as the application progresses. If we consider a multi-threaded region, the basic-block share is different for all threads. This is subtly captured by concatenating the per-thread execution fingerprints.

execution count of the marker at the start and end of the region. The value of count for a particular region size is invariant across multiple executions, which represents the unit of work done. Hence, these markers remain valid simulation points even in the presence of spin-loops.

3.3.4 Identifying Loops using DCFG

Loops are often found in typical applications, and the number of loop iterations can remain constant for an unmodified application for a particular input over multiple executions. This is the key to our generic methodology which is explained below in detail.

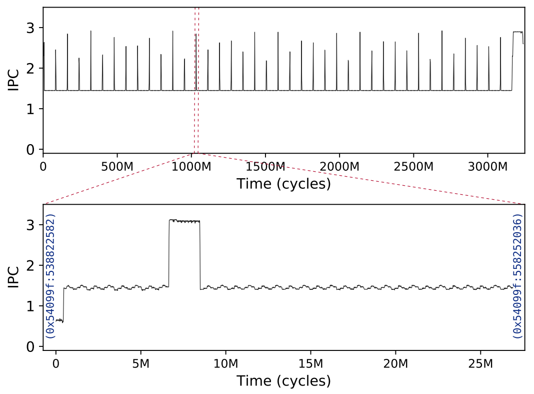
We employ a Dynamic Control-Flow Graph (DCFG) to identify the regions that represent loops. A DCFG is similar to a classical control-flow graph with a primary difference: Each edge of a DCFG is augmented with a trip count to indicate the number of times the edge was traversed. The source code locations whose executions correlate with a phase change are called software phase markers [26]. The software phase markers identify the



638.imagick_s/magick/morphology.c

```
2842 #if defined(MAGICKCORE_OPENMP_SUPPORT)
2843   #pragma omp parallel for schedule(static,4) shared(progress,status) \
2844     magick_threads(image,result_image,image->rows,1)
2845 #endif
2846   for (y=0; y < (ssize_t) image->rows; y++)
2847   {
            ......
2886     for (x=0; x < (ssize_t) image->columns; x++)
2887     {
3021       for (v=0; v < (ssize_t) kernel->height; v++) {
3022         for (u=0; u < (ssize_t) kernel->width; u++, k--) {
            ......
3034         } /* u */
            ......
3037       }  /* v */
3342     } /* x */
3357   } /* y */
           ......
```

(a)



(b)

Figure 3.4: An example of a representative region identified by LoopPoint. (3.4a) The numbers represent iterations of the corresponding loops that form the 8-threaded region. The start point and end point of the chosen region are at line 3022, the entry point of loop *u*. (3.4b) The top graph shows the variation of IPC over time for the full application run, while the bottom graph shows that of the chosen region. The (PC, count) boundaries are marked inside the IPC graph of the region.

phase changes that occur in an application execution irrespective of the underlying microarchitecture. These phase markers need to repeat in number and order across multiple program executions so that they can meaningfully act as simulation region boundaries. We choose headers of loops that are in the main image of the program, assuming that the synchronization loops are in the libraries. The number of iterations of synchronization loops may vary across different program executions. The DCFG of the whole program execution is instrumented for loop header instructions to identify a subset of loops from the main image. Loop header instructions are instrumented to emit Basic Block Vectors (BBVs) after *slice-size* number of instructions. Figure 3.4 shows a region identified using DCFG. The region is contained in the `638.imagick_s.1` application with train inputs and eight threads.

3.3.5 Clustering Representative Regions

Once an application is profiled, and region boundaries marked, we will have a collection of variable-length regions. These BBVs (with spin-loops filtered) represent the state of the application and also allow one to understand the amount of work accomplished by each thread. For example, in regions where a single thread is active, the thread will no longer interfere with memory requests from other threads, potentially leading to faster single-thread execution. However, a fully populated system with N threads would continue to interfere, potentially slowing overall progress. The amount of time the application executes becomes the combination of the amount of work executed in one quantum, together with the runtime attributed to that quantum. These quanta can then be clustered in order to identify similar work, and therefore identify similar runtime behavior. Although BBVs are used in this work, other feature vector information [34] can be concatenated on a per-thread basis and can be used in this methodology.

The BBVs are projected down to 100 dimensions by random linear projection to bring down the computing requirements for the clustering algorithm. We use the K-means clustering technique [113] along with a BIC goodness criteria [114] to select clustering in a method similar to previous work [20]. The K-means algorithm requires the selection of the maximum number of clusters that we can expect, $maxK$, for which we use $maxK = 50$.

Because we use BBV data that represents both parallelism and work executed, we can now cluster the regions and use the resulting clusters for workload extrapolation. We choose the BBV that is closest to the centroid of each cluster to be the representative of the cluster. We generate the region that represents each cluster from the original application based on the region boundaries and call them *looppoints*.

3.3.6 Warmup

For high performance, we will want to simulate each looppoint separately, in parallel, given enough resources. For accurate results, the microarchitectural state needs to be warmed up at the start of the simulation of each region. There are several techniques [5, 90, 115] proposed to warmup cache state. For binary-driven simulation, we warm up each region from the start of the application to minimize warmup error. Likewise, for checkpoint-driven constrained simulation, we use a sufficiently large warmup region preceding the simulation region. Determining the appropriate amount of warmup required for each representative region falls outside the scope of this work.

3.3.7 Runtime Extrapolation

Once the representatives are simulated, we can estimate the overall application execution time through the use of weight-based extrapolation. In this methodology, we use the percentage of work that this region represents, based on the instruction count of the entire collection of representatives that have been clustered together relative to the total amount of work done in the original application (quantum multiplier), to extrapolate the final runtime performance. The instructions that contribute to spin-loops are not considered here. The final step of this methodology uses the simulation results of these identified representatives, along with the multiplier, to reconstruct the overall workload runtime.

Our runtime extrapolation uses the below mentioned formula considering N looppoints identified as rep_1 to rep_N:

$$total\ runtime = \sum_{i=rep_1}^{rep_N} runtime_i \times multiplier_i \tag{3.1}$$

The *multiplier* of a looppoint is the ratio of the sum of the filtered instruction counts from all of the regions that are represented by the looppoint to the filtered instruction count of that looppoint.

$$multiplier_j = \frac{\sum_{i=0}^{m} inscount_i}{inscount_j} \tag{3.2}$$

where m is the number of regions that are represented by the j^{th} looppoint.

We evaluate our region selection methodology by comparing the extrapolated runtime based on region simulation with the actual runtime based on the whole-application simulation to compute the prediction error. We demonstrate runtime extrapolation using the above formula, but this methodology can be used for any event of interest, such as cache and branch miss counts, for example.

3.3.8 Reproducible Application Execution for Accurate Analysis

The execution path of a multi-threaded application can vary from run to run due to several factors. One requirement to use this methodology is the ability to analyze a multi-threaded application in a repeatable way. Traditional execution environments do not support this type of execution to allow for reliable, reproducible execution. We leverage Intel's Pin [116] and Pinplay [77] tools to generate reproducible, constrained, multi-threaded execution snapshots, called pinballs, to allow for repeatable analysis. Pinballs are more advanced than a trace file in that they contain a snapshot of the execution state of an application (registers and memory). By replaying the Pinball, we can analyze the properties of an application to collect the microarchitecture-independent execution signatures of the application.

3.3.9 Putting it All Together

Together, the combination of reproducible replay of applications, along with the identification and clustering of workload characteristics, allows us to build an end-to-end methodology to identify workload representatives for performance extrapolation. Previous works [34] have shown that extrapolation in this manner does apply to runtime, as well as other metrics of interest. The insights with respect to the identification of application parallelism, as well as the constrained, reproducible execution of the workloads, allow us to analyze, cluster, and extrapolate multi-threaded workloads across a number of synchronization types.

3.3.10 Speed-up Potential

One of the most significant benefits of a checkpoint-based methodology is the ability to substantially reduce the amount of work that needs to be simulated to estimate the entire application performance. Simulator performance relates directly to the required length and number of regions to simulate. In addition, checkpoints can be simulated in parallel, with enough resources available, speeding time-to-results significantly.

3.3.11 Workload Applicability

LoopPoint targets statically scheduled multi-threaded workloads regardless of the synchronization mechanisms used in order to simulate them in a faster way that was not possible before. The methodology is particularly effective for loop-intensive applications. For other workload types or scenarios involving aggressive loop optimizations, the heuristic can be adapted to utilize function calls or database transactions as the unit of work. Dynamically scheduled multi-threaded applications would require a different type of methodology for sampling due to their non-deterministic nature. This is because such applications can interact with other threads in ways that were not seen in the initial execution of the application, potentially leading to incorrect extrapolations.

Checkpoint-based methodologies, such as BarrierPoint, necessitate prior application analysis to determine workload phases. However, these phases are input-dependent and are reusable only when the application and corresponding libraries exhibit consistent behavior. We address the problem of workload imbalance among the threads (a heterogeneous workload) by keeping per-thread information intact while clustering the individual regions. Like other checkpoint-based methodologies, we also assume that the hardware configuration is known up-front. This configuration is free from any runtime-dependent configuration changes or unexpected events that trigger a configuration change while the application is running. An example of a dynamic event is thermal throttling resulting in a dynamic voltage and frequency scaling (DVFS) event, which can affect the application performance and is runtime- and hardware-dependent. Due to the microarchitecture-dependent behavior of vectorized code (SIMD instructions), the application profile and identified clusters may vary across different microarchitectures. To address this, SIMD instructions may need to be treated as continuous instruction sets within the BBV.

3.4 Experimental Setup

In this section, we describe the setup on which we conducted our experiments to evaluate our generic multi-threaded sampling methodology.

3.4.1 Simulation Infrastructure

In this work, we use Sniper multicore simulation infrastructure [14] (version 7.4) with modifications to support PC-based simulation region specification. We configured Sniper to model a multicore out-of-order processor resembling the Intel Gainestown microarchitecture using an 8 or 16-core processor model to simulate 8 or 16-threaded (respectively) applications. The simulated system characteristics that we use are detailed in Table 3.1.

Table 3.1: The primary characteristics of the simulated system.

Component	Features
Processor	8 & 16 cores, Gainestown-like microarch.
Core	2.66 GHz, 128 entry ROB
Branch predictor	Pentium M
L1-I cache	32K, 4-way, LRU
L1-D cache	32K, 8-way, LRU
L2 cache	256K, 8-way, LRU
L3 cache	8M, 16-way, LRU

Table 3.2: SPEC CPU2017 speed application attributes. F=Fortran, KLOC=thousand lines of code. From [1]

Application	Lang.	KLOC	Application Area
603.bwaves	F	1	Explosion modeling
607.cactuBSSN	F, C++	257	Physics: relativity
619.lbm	C	1	Fluid dynamics
621.wrf	F, C	991	Weather forecasting
627.cam4	F, C	407	Atmosphere modeling
628.pop2	F, C	338	Wide-scale ocean modeling
638.imagick	C	259	Image manipulation
644.nab	C	24	Molecular dynamics
649.fotonik3d	F	14	Comp. Electromagnetics
654.roms	F	210	Regional ocean modeling

3.4.2 Workloads

In order to evaluate the proposed methodology, we consider the SPEC CPU2017 [117] benchmark suite. SPEC CPU2017 is available in two different versions depending on the evaluation purpose: `rate` and `speed` [118]. The `rate` version is used to estimate the throughput of the underlying system whereas the `speed` version is used to estimate the runtime of the benchmark on the system. Unlike prior versions of SPEC benchmarks, CPU2017 includes a set of synchronizing multi-threaded programs that share memory consisting of OpenMP-compatible multi-threaded applications. We use the `speed` version of SPEC CPU2017 with `train` inputs and eight threads (See Table 3.2 for application descriptions) for our evaluation. The `train` input set is used so as to keep the full program simulation time to a reasonable length. As the detailed simulation of the full

SPEC CPU2017 applications with `ref` inputs is not practical, computing the sampling error is also not feasible. Therefore, we utilize the `ref` inputs to estimate the potential speedup of the methodology in the chapter. The benchmarks we use include OpenMP directives, with a summary of the primitives used described in (Table 3.3).

Table 3.3: SPEC CPU2017 speed synchronization primitives used. sta4=static for, dyn4=dynamic for, bar=barrier, ma=master, si=single, red=reduction, at=atomic, lck=lock.

Application	sta4	dyn4	bar	ma	si	red	at	lck
603.bwaves	Y					Y	Y	
607.cactuBSSN	Y	Y	Y			Y	Y	
619.lbm	Y							
621.wrf		Y		Y				
627.cam4	Y	Y	Y	Y				
628.pop2	Y		Y	Y				
638.imagick	Y		Y	Y	Y			Y
644.nab		Y	Y			Y	Y	
649.fotonik3d	Y							
654.roms	Y							

All SPEC CPU2017 workloads except `657.xz_s` runs are 8-threaded. `657.xz_s.2` runs with 4-threads whereas `657.xz_s.1` runs as a single-threaded application.

All the benchmarks in the SPEC CPU2017 benchmark suite are compiled using the Intel compiler toolchain (Intel Parallel Studio XE, version 2019 Update 2) with optimizations enabled (-O2) and debug information available for binary to source-level mapping, and built for the 64-bit x86 instruction-set architecture.

We also use NAS Parallel Benchmarks (NPB) [119, 120] version 3.3 with OpenMP based parallelization [121] that use class `C` inputs. We evaluate all benchmarks in the suite with both 8 and 16 threads, but do not evaluate the npb-dc (data cube) benchmark because of the large amount of data generated by that application. These benchmarks are compiled using GCC 5.5 for applications in C and GFortran for Fortran applications with -O3 optimizations for the x86-64 architecture.

We consider both `active` and `passive` wait policies for thread synchronization of the SPEC CPU2017 OpenMP applications. We use the `passive` OpenMP wait policy to configure NPB benchmarks. In passive wait policy, the threads do not spin while waiting for other threads. Meanwhile in the case of active wait policy, the threads remain active and they consume processor cycles while waiting by executing spin-loops. The use of (PC, count) region specification can accurately represent a region over multiple runs even in the presence of spin-loops, which is not possible if the region specification is based on global or per-thread instruction counts.

For each benchmark, we record the execution path of the whole application and keep it as a pinball so that it can be replayed in both constrained and unconstrained mode later on. We have developed Pintools [116] to generate BBVs of the regions which are fed to Simpoint for clustering the regions to identify the representative regions. We also have employed Pintools to restrict the forward progress of all the threads in a well balanced way thereby avoiding the chances of recording a skewed trace because of CPU load imbalances. The representative regions identified are simulated in parallel. We evaluate the runtime accuracy of the chosen representatives by simulating in constrained and unconstrained modes.

3.4.3 Constrained Execution Infrastructure

We use Intel's PinPlay [77] infrastructure that provides tools to record and replay arbitrary regions of a program execution. The recorder captures the execution of an application in a set of files collectively called a pinball [3] which can later be replayed on any machine since pinballs are portable. A pinball consists of a memory file (`.text`), the architecture register values at the beginning of the execution region in per-thread register files (`.reg`), a set of memory and register values in per-thread injection files (`.sel`), and a subset of shared-memory dependencies among various threads in per-thread dependency files (`.race`). A pinball once captured is self-contained, which means that both the application binary and inputs are not needed during replay of the pinball.

The replayer loads the initial memory and register state and starts executing the restored program region like a regularly loaded binary. System calls are skipped and their side-effects are injected. Shared-memory access in all threads are monitored and the threads are artificially delayed as needed to enforce the access order as recorded in the pinball. Finally, the replay is ended gracefully when the exit condition is met. Since system calls are skipped during replay, a pinball can be replayed across different operating systems.

3.4.4 DCFG and Basic Blocks

The Dynamic Control Flow Graph (DCFG) is created by executing the program via a pin-tool enabled with the DCFG library [122, 123]. Internally, the pin-tool *hooks* the control-flow instructions and records a count of each of the resulting edges throughout the execution of the workload on a per-thread basis. At the end of the execution, *fall-through* edges are created to ensure non-overlapping basic blocks. These basic blocks are guaranteed to have only one entry and one exit point and not overlap with each other. In this way, they differ from the basic block structures in Pin, which do not have these guarantees. The resulting basic blocks and the edges that connect them thus create a connected graph. From this graph, routine boundaries are identified based on call edges and heuristics to handle non-standard routines that are sometimes found in non-compiled code. Inside the sub-graph of each routine, the immediate dominators of each node are found. Loops are then identified using the immediate dominator relationships. The graph, including the identified routines and loops are recorded.

3.4.5 Unconstrained Replay

PinPlay's replayer enforces determinism among the threads by injecting recorded system call side-effects and enforcing the recorded shared memory access thread order. We use this mode when analyzing the workload (collecting BBVs and DCFGs to be used in the clustering phase), to ensure different steps of the profiling methodology have a consistent view of the program's execution flow (as recorded during the initial whole-program recording). However, during performance simulation, we want the timing model to control thread progress and synchronization, not PinPlay as this can introduce artificial thread stalls[1].

3.4.6 Synchronization Handling

OpenMP active runs, enabled by setting the environment variable OMP_WAIT_POLICY to ACTIVE [125], have threads busy-waiting at user-level (as opposed to using futex() in the passive runs). We replay a pinball that was recorded earlier for reproducible analysis for the generation of BBVs. If we directly use the recording, we encounter the busy-waiting code that was originally executed by the application. However, the busy-waiting code can differ if the application is executed another time with different conditions. While busy-waiting consumes processor cycles, they do not contribute to the *real* work done by the program. Therefore, we ignore busy-waiting during BBV profiling, yet include it during simulation. Identifying busy-waiting code automatically [126] can be a challenge and is yet another research problem. In our methodology, we ignore the entire code from the relevant synchronization library (`libiomp5.so` in our case). Note that this idea can easily be extended to other compilers and threading libraries. For example, in the case of applications using pthread synchronization, we can ignore the code from the `libpthread` library. The filtered instruction count is up to 40% (for `657.xz_s.2`) fewer than the original instruction count for the active runs.

[1]See [124] for a methodology that uses constrained replay during multi-threaded performance simulation, and which can, in limited cases, work around the artificial stalls.

3.5 Evaluation

In this section, we present the evaluation results of LoopPoint methodology. We analyse the effect of various model parameters that make up the methodology. We also evaluate the accuracy and the speedup achieved using LoopPoint.

3.5.1 Accuracy

We show the accuracy of LoopPoint methodology by comparing the predicted runtime and the actual runtime of the application. The predicted runtime is calculated by considering the performance of all the representative regions as mentioned in Section 3.3.7. The representative regions are augmented with a warmup region so that the microarchitectural state is warmed when the detailed region starts simulating. The prediction error of our methodology is the percentage difference in the simulation performance of the whole application and the extrapolated performance making use of the performance of all the representative regions identified for the application.

3.5.1.1 Constrained and unconstrained simulations

The LoopPoint methodology is tested for applications using the `active` and `passive` wait policies, and the simulation results are given here. Synchronizing multi-threaded applications with `active` wait policy uses spinloops to synchronize the threads. Sampling such an application can be considered a difficult problem to solve. We ignore the instructions that contribute to spin-loops during BBV generation and clustering phases as described in Section 3.4.6.

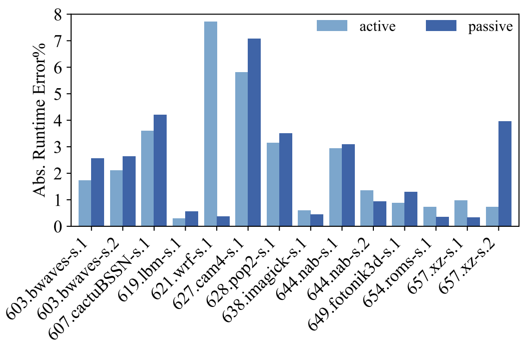
We perform binary-driven unconstrained simulations of the whole application as well as the representatives to measure the performance. In order to mark the region boundaries using (PC, count) correctly, we need to keep spin-loops away from being the region boundaries, as mentioned earlier. We limit the region boundaries to be from the application code and not from any of the library code. Here, we make an assumption that the synchronization code can only be present in the libraries.

The region checkpoints are generated as pinballs which can be used for constrained simulation. We assume a large enough warmup region added to the representative region while generating the pinball checkpoint. However, using constrained simulation introduces artificial thread delays and is therefore not reliable for performance extrapolation. There are several ways to simulate these pinball checkpoints in an unconstrained way. One such method is to convert them to ELF binaries, called ELFies, as discussed in a prior work [47]. In this chapter, however, we are not evaluating ELFies. Instead, we consider the region boundaries specified as (PC, count) to perform unconstrained simulation using the application binaries by providing perfect warmup before the start of detailed simulation. One caveat that we want to mention is that not all region boundaries specified using (PC, count) can provide stable regions. For instance, applications can have certain code blocks that are selectively executed with respect to the underlying microarchitecture. Such code blocks or PCs cannot serve as stable (PC, count) region boundaries. We assume that the users can choose the appropriate stable regions and that, while straightforward to accomplish in an automated way, we leave that analysis to future work.

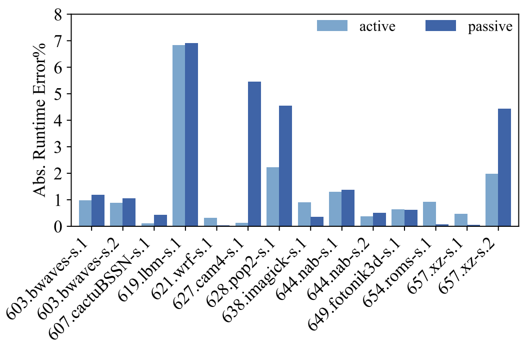
Results when simulating constrained simulation can be misleading and can lead to high errors. For example, we observe a runtime error for `657.xz_s.2` of up to 19.6% while simulating in a constrained environment. One of the reasons that using constrained simulation infrastructure can result in high error rates is that the simulation itself does not properly mimic the real application run. Instead, the application tries to replicate the behavior that was recorded previously on a specific machine. For instance, constrained execution forces spin-loops to be replayed, even though this would not occur in a real execution. This introduces high error for applications, like `657.xz_s.2`, that have fewer synchronization points compared to other applications in the SPEC CPU2017 benchmark suite and, therefore, can see high variability from run to run.

The runtime prediction results (Figure 3.5a) using the unconstrained simulation of active applications yield an average absolute error of just 2.33%, whereas that of passive applications is 2.23%. These error rates are comparable to previous sampling methodologies [34].

The looppoints identified are representative of the application across microarchitectural configurations. Our up-front analysis is solely based on architecture-level details, not microarchitectural settings or simulation details. Figure 3.5b shows the error in predicting the runtime of the same applications while simulated for an inorder core instead of the out-of-order Gainestown-like core while keeping all other simulation parameters the default as in Table 3.1. The graph clearly shows that looppoints can be portable across microarchitectures.



(a) Gainestown core



(b) Inorder core

Figure 3.5: The runtime prediction errors of SPEC CPU2017 applications (train inputs) using active and passive wait policies that use eight threads for unconstrained simulation. The y-axis represents the percent error in predicting the runtime of each of the applications along the x-axis.

3.5.1.2 Varying the number of threads

We show that LoopPoint supports varying the number of application threads. Figure 3.6 shows the error rates while predicting the runtime of the NPB benchmarks. The applications are evaluated using eight and 16 threads. Note that the applications using a different number of threads need to be profiled separately, as discussed in Section 3.3. We observe that the average absolute error obtained is 2.87% for 8-threaded applications, while for the 16-threaded applications, it is as low as 1.78%.

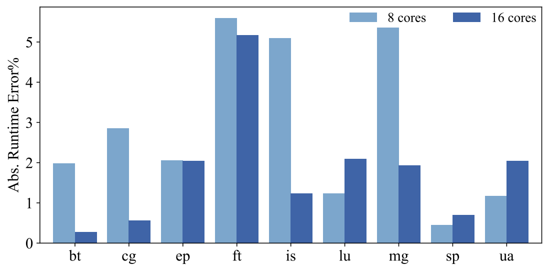
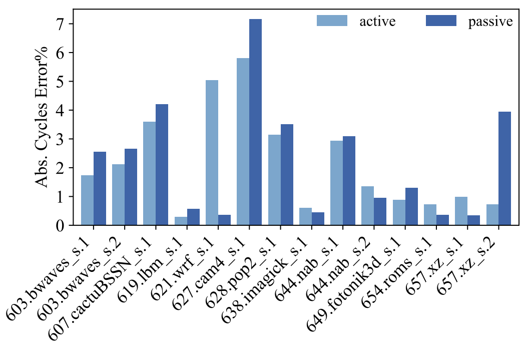


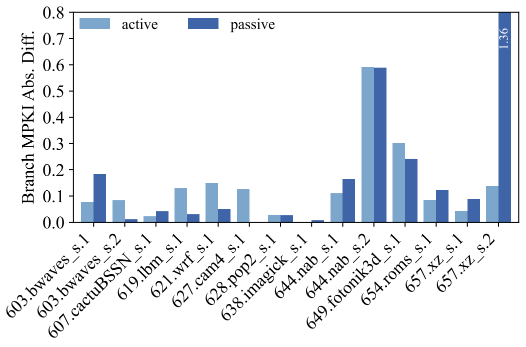
Figure 3.6: The runtime prediction results of the NPB benchmarks that use 8 and 16 threads. The applications use a passive wait policy and class C inputs. The y-axis represents the error percentage in predicting the runtime of each of the applications on the x-axis.

3.5.1.3 Comparison of other metrics

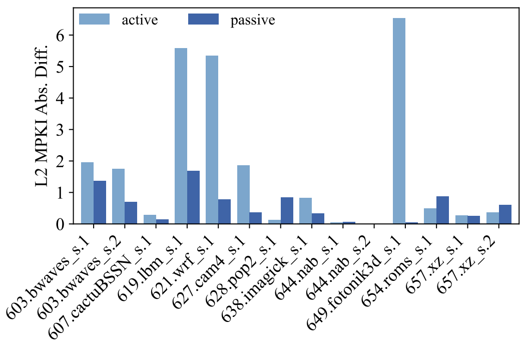
Figure 3.7 shows the performance prediction of several metrics while simulated on an unconstrained environment for applications using `active` and `passive` wait policies. LoopPoint can determine microarchitectural metrics like the number of cycles (Figure 3.7a), branch miss rate or MPKI (Figure 3.7b), the miss rates or MPKI of different components in the memory hierarchy (Figure 3.7c), etc. In Figure 3.7b and Figure 3.7c, we show the absolute differences in the metrics predicted, rather than the percentage error in prediction, because those metrics have small absolute values and a small difference can result in a high percentage error. Previous research [24, 34] has presented differences in a similar manner.



(a) Number of Cycles



(b) Branch MPKI



(c) L2 MPKI

Figure 3.7: The prediction errors of various metrics for SPEC CPU2017 benchmarks using LoopPoint. The benchmarks use active and passive wait policies with train inputs and eight threads and are simulated in realistic unconstrained mode.

3.5.2 Speedup

We consider speedup in two different ways: theoretical speedup and actual speedup. Theoretical speedup is the reduction in the number of instructions (ignoring the instructions that contribute to spinloops) to be simulated in detail when using the LoopPoint methodology. We also define the actual speedup as the reduction in the simulated runtime using LoopPoint with respect to the simulated runtime of the whole application.

Serial speedup is the speedup achieved when all the representatives are simulated back-to-back. It is the overall reduction in work given the serial execution of both the full and reduced workload. Parallel speedup assumes sufficient parallel resources and evaluates the speedup given the execution of all regions in parallel.

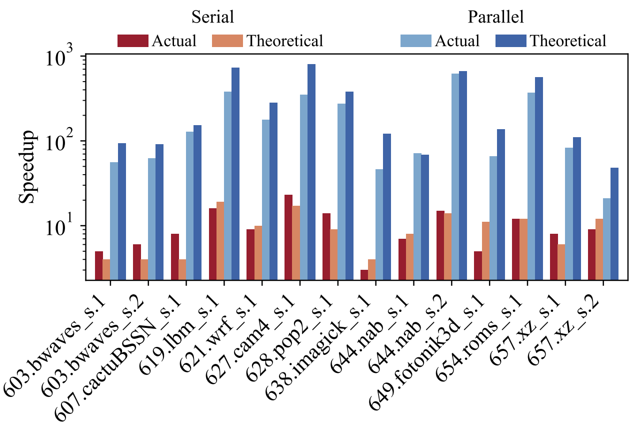


Figure 3.8: A comparison of theoretical and actual speedups achieved by LoopPoint. The workload used is SPEC CPU2017 applications (active wait policy) using train inputs.

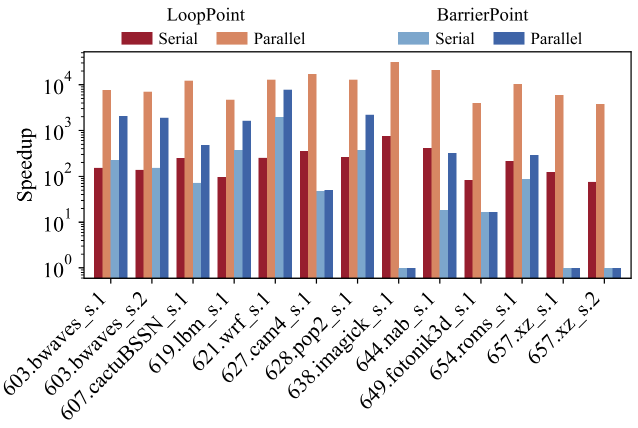


Figure 3.9: LoopPoint and BarrierPoint theoretical speedup for SPEC CPU2017 applications (passive wait policy) using ref inputs.

In Figures 3.8 and 3.9, we see both the serial and parallel speedups for these applications. We obtain a maximum speedup of 801× for the applications with `train` inputs and 31,253× for the applications with `ref` inputs. The average serial speedup for applications using `train` inputs and `ref` inputs are respectively 9× and 244× whereas the average parallel speedup for the applications are 303× and 11,587× respectively for `train` and `ref` inputs. This implies that a significant reduction of simulation resources is now possible using the LoopPoint methodology, where simulations that would take months to complete can now be finished in hours.

In Figure 3.9, we compare the theoretical simulation speedup using LoopPoint and BarrierPoint for the benchmarks using `ref` inputs. Note that we do not plot the actual speedup values using the `ref` inputs. We first validate our methodology with `train` inputs, and by extension, we analyze and simulate `ref` input representatives to estimate the performance of the larger application with confidence. Unfortunately, it is not possible to validate the error rates for applications with `ref` inputs because the full runs take too long to simulate (a few months to years, as shown in Figure 3.1).

We observe that LoopPoint consistently achieves good speedup whereas BarrerPoint lags behind for a number of applications. LoopPoint is able to reduce the application into representative regions that can finish the simulation in a reasonable time. Additionally, with the BarrierPoint methodology, there is no guarantee on the size of a representative region. For example, the 8-threaded `638.imagick_s.1` benchmark has a very large inter-barrier region (93.06 B instructions) that is comparable to the size of the entire application (93.35 B instructions), defeating the purpose of sampling. However, there are a few applications for which BarrierPoint outperforms LoopPoint. Those applications have a large number of barriers, and the inter-barrier regions are typically smaller than the LoopPoint regions. BarrierPoint is unsuitable to evaluate both of the `657.xz_s` applications as they do not contain barriers at all. Overall, a hybrid approach can be chosen to speed up smaller applications, but LoopPoint provides the first methodology to allow for generic sampling of applications that results both in a high simulation speedup and accuracy.

We also show the speedup achieved using NPB applications in Figure 3.10. LoopPoint achieves good speedups while the applications are evaluated for eight threads as well as

16 threads. The maximum parallel speedup achieved while evaluating the 8-threaded applications is 2,503× with an average of 1,031×, whereas, for the 16-threaded applications, the maximum speedup achieved is 1,498× and an average of 606×. Do note that NPB applications are less complex and more repetitive in nature than SPEC CPU2017 applications. Therefore, the error rates are lower, and the speedups achieved are larger when compared to the `train` inputs of the SPEC CPU2017 suite.

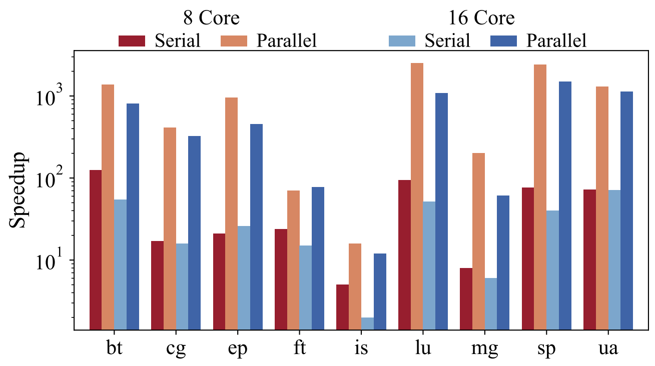


Figure 3.10: A comparison of actual speedups achieved by LoopPoint when the applications use 8 and 16 cores. Speedups are listed for the NPB suite using the C input set and a passive wait policy.

3.6 Related Work

Before architects build new hardware designs, it is extremely useful to predict the hardware design's power, performance, and area (cost). Existing circuit-design tools are able to simulate complex, modern applications on large, multi-core systems, but at the cost of significant simulation time that can be intractable (requiring months to years of simulation time for the SPEC CPU2017 benchmarks).

While there have been many attempts to solve this problem, previous works were unable to provide a combination of three things for multi-threaded workloads: (1) choosing accurate representatives without detailed simulation, (2) demonstrating simulation speedup based on application representatives, not on overall application runtime and (3) allowing the simulation of hardware designs that might not yet have analytical models. Our proposal addresses all these concerns through the determination of application parallelism, clustering, and the extrapolating of the results based on this information.

Sampled Simulation Methodologies. Sampled simulation methodologies applicable to single-threaded applications [2, 20, 26, 27] and multi-threaded applications [30, 32, 33, 34, 43, 52, 79] are discussed in Chapter 2.

Analytical Modeling. There has recently been some progress on the development of a completely analytical model for single-threaded [127] [128] and multi-threaded [85] workloads. One major drawback of analytical models is the inability to estimate the performance of next-generation hardware designs. New processor, cache, and memory techniques without analytical models will not be able to use these methodologies.

Constrained Simulation. Multi-threaded checkpoints were used [124] for constrained simulation. Their goal was to estimate the relative performance analysis of regions-of-interest across multiple micro-architectures. They describe a mechanism for speedup computation in the presence of artificial stalls added by the constrained replay of checkpoints during simulation. There could be cases where the speedup computation is inconclusive. We support unconstrained simulation as well as constrained simulation and also provide an absolute performance extrapolation methodology. For relative, cross-micro-architectural performance analysis, unconstrained simulation is desirable as it need not have to deal with artificial stalls.

Handling Busy-waiting. The problem of busy-waiting is mentioned in [77] although in the context of multi-process programs using Message Passing Interface (MPI). The work focuses on simulating a specific single-threaded process from multiple processes in an MPI program and uses the selective logging feature of PinPlay to exclude the busy waiting code from consideration, both in the profiling and simulation phases.

Workload Characterization. There are different works that study the time-varying runtime behavior of standard benchmarks. Wu et al. [68] study the phase behavior of SPEC CPU2017 workloads. Moreover, the work identifies the single-threaded simulation points using SimPoint methodology and correlates them with the phase behavior. Nair et al. [129] study the phase behavior of SPEC CPU2006 and SPEC CPU2000 using SimPoint methodology. The work demonstrates that SimPoint yields similar CPI prediction results for both application suites, suggesting similar phase behavior.

3.7 Conclusion

The need to understand larger, more complex multi-core processors continues to increase. This becomes even more critical as the multi-core processors (and the serial code) tend to be the bottleneck in highly parallel applications. General-purpose applications are found on embedded devices, mobile phones, and back-end data center servers. While platforms may differ in their demands, accurately understanding the applications remains crucial.

Simulation solutions alone are insufficient because of the significant slowdown (10,000× or more [6]) seen when simulating applications with industrial-quality simulators. Simulation solutions today require alternatives like sampling to reduce the workloads to realistic simulation times. However, current sampling solutions either target single-threaded workloads or are only applicable to specific workload types.

In this work, we present a generic multi-threaded sampling methodology, one that considers the inherent parallelism of the application and allows for the automatic reduction of workloads to sizes that are on the order of the representatives of the workloads themselves. We demonstrate how our classification methodology automatically partitions the workload into representatives and allows us to predict the performance of the workloads at hand with high accuracy.

Chapter 4

Viper: Utilizing Hierarchical Program Structure to Accelerate Multi-core Simulation

You shall know the truth and the truth shall make you mad.

— Aldous Huxley

Workload sampling techniques typically rely on fixed-length intervals for analysis, which can often be out of sync with the periodicity of program execution. Since an applications phase behavior is strongly correlated to the code it executes, it can exhibit a hierarchy of phase behaviors that can be observed at various interval lengths, rendering conventional sampling techniques inadequate. We propose Viper, which leverages the hierarchical structure of program execution in order to achieve better sampling accuracy and smaller regions, which enables faster RTL simulations.

4.1 Introduction

As we approach the limits of technology scaling, there is a growing emphasis on efficient and high-performance processor designs. Exploring and evaluating the design space of these next-generation architectures is an essential part of this research. However, the

Alen Sabu and Changxi Liu contributed equally to this research.

traditional dependence on extremely time-consuming microarchitectural simulations for large, realistic workloads proves impractical in addressing this challenge. For multi-threaded workloads, this issue is further exacerbated by the complex interactions between multiple threads and the synchronization techniques employed to achieve scalable performance. One solution to address this issue is sampled simulation, which selects a representative subset of regions to simulate in detail and interpolates the performance of the entire application based on this. Prior works [2, 8, 20, 33, 34, 130] have demonstrated that, due to the repetitive behavior of workloads, sampling can often reduce the simulation time by orders of magnitude while preserving the original program characteristics.

SimPoint [20] reduces the simulation time by leveraging the application's phase behavior for single-threaded workloads. It does so by splitting the application into *fixed-size* regions, clustering them based on their execution behavior, and then simulating a representative element from each cluster in detail to extrapolate the performance of the entire application. However, a major drawback of this method is that it uses fixed-size regions for analysis, which do not often align with the actual periodicity [19] of program execution. Simpoint 3.0 [131] introduces variable length regions but does not address application periodicity. Moreover, since an applications phase behavior [22, 132] is strongly correlated to the code it executes, it can exhibit a hierarchy of phase behaviors that can be observed at different interval lengths [32]. Consequently, a single fixed region size cannot effectively capture the full spectrum of phase behaviors and often leads to suboptimal phase classification [25].

Later works, such as BarrierPoint [34], TaskPoint [44], and LoopPoint [8], address this shortcoming by utilizing the program structures and constructs within the application code to split the application into a series of independently analyzable regions to build a representative sample. Unfortunately, however, both BarrierPoint and TaskPoint only apply to specific classes of applications. BarrierPoint targets applications that use global barriers for synchronization, whereas TaskPoint targets task-based applications. While LoopPoint applies to generic multi-threaded applications, the regions it selects do not necessarily align with the application's phase behavior. Moreover, all these techniques use large region sizes ($\approx$100 million instructions or more per thread), suitable for microarchitecture-level simulations (which take a few hours) but not for RTL-level simulations, which may take weeks to months for completion. In addition, no previous methodology provides a solution to detect small regions needed for RTL-level simulation, as they would typically result in aliasing [32], leading to unpredictable results. In this work, we propose a solution to solve both of these issues to achieve high performance and accuracy.

The goal of this work is to address the generic sampling problem by selecting representative regions that align with the application phases for simulation. Utilizing the innate program structures instead of fixed-length intervals allows for flexible region sizes that are more likely to be aligned with the application periodicity, thereby reducing the chances of aliasing [32]. To do this, we present a novel methodology, Viper, that enables fast and efficient analysis prior to sampled simulation. In short, we make the following contributions to this work:

- We propose a novel methodology, Viper, that goes beyond prior state-of-the-art sampled simulation techniques to allow for fine-grained region selection and accurate performance reconstruction.
- We present a methodology that meets the requirements for RTL simulations for accurate performance estimations. We show this by performing experiments on microarchitecture-level and RTL-level simulators, enabling the detailed evaluation of large benchmarks.
- We provide an extensive evaluation of Viper and demonstrate best-in-class accuracy (average sampling error of just 1.32%) and speedup of up to 2,710$\times$, with an

average of 358× for the train input set of SPEC CPU2017 benchmarks. We also explore the accuracy and performance trade-offs of Viper in comparison with prior works.

The rest of the chapter is organized as follows. In Section 4.2, we discuss the relevant background and the challenges involved in the simulation of multi-threaded applications. Section 4.3 presents the Viper methodology in detail. We then discuss the experimental infrastructure in Section 4.4, followed by an extensive evaluation of Viper in Section 4.5 showcasing the applicability of the proposed methodology and conclude the chapter in Section 4.6.

4.2 Background and Motivation

In this section, we present the background to understand the key features of sampled simulation. We also discuss the challenges in simulating large workloads and how the existing sampling methodologies are insufficient to address them.

4.2.1 Program Sampling

Sampling is the process of selecting a minimal subset or a sample from a population to represent the entire population. The attributes or characteristics of the population are estimated using the selected sample. We employ this technique to reduce the simulation time of large workloads by simulating a representative sample from the entire program execution. Prior works [2, 20] split an application into a series of execution slices and cluster these slices with similar execution features into groups. These techniques demonstrate high performance by simulating selected representative slices from each group to represent the entire cluster of software slices.

Single-threaded sampling is largely considered to be a solved problem, whereas multi-threaded sampling has been a long-standing problem due to the complexity of the workload behavior: threads that sleep, synchronize, or are being delayed in spin-loops, among other issues. Alameldeen et al. [73] demonstrated the limitations of non-determinism with multi-threaded workloads and demonstrated that IPC can be a poor performance indicator [31], leading to inaccurate estimation of speedup or run time.

While initial works on multi-threaded sampling [79] focused on handling applications with uncorrelated thread behaviors, subsequent research [32, 33] considered time as the sampling unit that applies to synchronizing multi-threaded workloads. However, a major drawback of this approach is that the whole application needs to be simulated sequentially (i.e., it cannot be parallelized), and thus, the maximum attainable simulation speedup is limited by the number of instructions in the whole application. Techniques like BarrierPoint [34] and LoopPoint [8] consider application barriers and loops, respectively, to define a unit-of-work [31]. BarrierPoint works on inter-barrier regions that can be so large that it is infeasible to simulate them, limiting scalability. LoopPoint divides the application into similarly sized regions enclosed within loop entries, ensuring size limits. However, the regions may not align with application phases. While LoopPoint regions are large enough to ensure accuracy and prevent aliasing, they are often too long for RTL-level simulation.

4.2.2 Checkpointing Techniques

Checkpointing is a widely used technique to save the state of a simulation at a particular point in time, which can then be restored later, allowing for further simulation or debugging. Checkpointing is often used to parallelize simulation as well as to improve performance by reducing the amount of time that needs to be spent re-simulating portions of an application that have already been executed. For example, Checkpoint/Restore In Userspace (CRIU) [133] is a well-known checkpointing mechanism on Linux. CRIU has been integrated with major container engines like docker [134]. In addition, gem5 [6, 135] uses its own checkpointing format that is useful to create microarchitecture-level snapshots of simulation that can be restored later. For x86 systems, the PinPlay infrastructure [77] supports storing the application state as architectural checkpoints, called Pinballs, which can be replayed on PinPlay-enabled tools and simulators. Recent works on executable checkpoints, like ELFies [47], are promising in terms of usability and portability, as it is supported on popular microarchitecture simulators like gem5 [6] and Sniper [14].

4.2.3 Microarchitectural State Warmup

Modern processors employ various techniques to improve performance, such as branch prediction, caching, and speculative execution. These techniques can have a significant impact on the workload execution run time. While simulating the key parts of an application, it is important to rebuild or warm up the microarchitectural state of the system. This ensures that subsequent simulations or performance measurements accurately reflect the behavior of the processor. Methodologies like LoopPoint [8] rely on simulating a large region right before the start of the simulation region to warm up the microarchitectural state, while SMARTS [2] or time-based sampling techniques [32, 33] enable functional warming during the entire simulation. TurboSMARTS [29] uses a microarchitecture-level checkpointing mechanism to handle warmup that captures and stores the functionally warmed system state before each simulation region. Checkpoint-based warmup techniques require a large amount of storage. Moreover, it may not always be suitable for microarchitecture design-space exploration that runs experiments altering the memory hierarchy configuration, like cache sizes or the number of cache levels, because it would invalidate the checkpoint for those regions, requiring new memory checkpoints for each cache configuration.

4.2.4 The Quest for Advanced and Efficient Sampling

With the widening gap between simulator performance and the processors they model, running a cycle-accurate full-system simulation of large designs can be extremely time-consuming. Current sampling solutions are primarily targeted for microarchitecture-level simulations. Some recent works [48] attempted to adapt these solutions for RTL-level simulations on Verilator [49] using smaller region sizes aiming to improve simulation efficiency, which, however, resulted in accuracy that is typically not acceptable. The result is that it is currently infeasible to evaluate the performance of large workloads on the RTL level. Recent works [136, 137, 138] addressed the problem of accelerating RTL simulation by leveraging techniques like batch processing, task-level dataflow execution, low-level parallelism, and selective execution. These orthogonal techniques to speed up simulation may not scale well for very large workloads. In addition, while FPGA simulation infrastructures, such as Diablo [139] or FireSim [50], offer a faster alternative for simulation, FPGAs are specialized devices with inherent limitations in terms of memory capacity and logic capacity. This means that it is often not possible to fit large, realistic processor models on FPGAs. This highlights the need for developing specialized workload sampling methodologies that can be flexibly applied to both microarchitecture-level and RTL-level simulations. These methodologies should support finer region granularities that align with the dynamic phase behavior exhibited by the application. By tailoring the sampling approach to capture the specific characteristics and phases of the workload, more accurate and efficient sampled simulations can be performed.

4.3 The Viper Methodology

In this section, we describe the details of our proposed sampled simulation methodology, Viper (depicted in Figure 4.1). Viper consists of four main steps: (i) Pre-profile Analysis

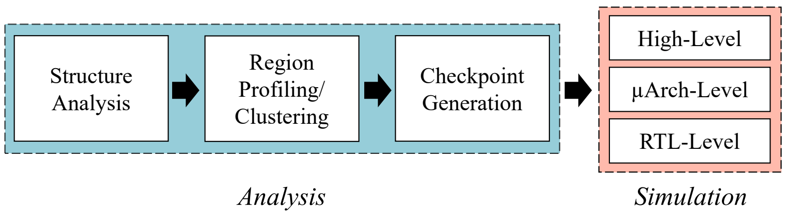


Figure 4.1: The workflow of Viper showing region identification, clustering, and simulation. The hierarchical structure of an application is used to identify regions. Sampled simulation is performed based on the clustering information of the regions. The simulation can be performed on various kinds of simulators depending on the level of detail required.

which marks the region boundaries at which we split the application, (ii) Region Profiling, where the profiling information in the form of feature vectors is collected for each region, (iii) Clustering, which groups together regions with similar execution behavior based on the profiling information, and (iv) Simulation, where each application region is simulated either in Detailed Mode or Fast-forward Mode based on the clusters formed. The full application performance is reconstructed from the performance of each region. In the subsequent subsections, we provide details on how each of these stages operates.

4.3.1 Exploring the Hierarchical Structure of Program Execution

Multi-threaded applications typically execute in a hierarchical flow, exhibiting different cyclic behavior patterns at varying interval lengths. These repeating patterns, often referred to as *phases*, are strongly correlated to the code executed by the application [18, 26, 32]. Thus, by analyzing the inherent program structures in an application's code, one can effectively capture the variations in its phase behavior. In Viper, we utilize this principle to identify *phase markers* [26] – the points within a program that corresponds to change in the application's phase behavior. Phase markers can be used to split the application into a series of independently analyzable regions.

There are several kinds of program constructs in a parallel multi-threaded code region, such as barriers and loops, which can serve as *potential phase markers* of the application. Choosing barrier counts or loop counts over instruction counts to represent work can accurately demarcate multi-threaded regions over several runs.

- Barriers: Multi-threaded applications include single-threaded and multi-threaded code regions, with thread synchronization at boundaries using barriers that can be detected by compiler-generated instructions or functions to mark new code regions in machine code. In OpenMP-enabled applications, the GCC compiler generates the `_omp_fn` identifier that can be used to detect barriers.
- Loops: Typically, generic multi-threaded applications consist of various levels of nested loops. In our analysis, we use the application's dynamic control-flow graph (DCFG) [123] to identify the loops in the outermost level of the code region as *task loops* and the remaining as inner loops or *ordinary loops*. The DCFG is utilized to identify loop headers, and for each loop, information about their outer loops and associated subroutines is then collected. This helps to determine whether a loop is the outermost one in the current subroutine and if the current subroutine is the outermost in the given multi-threaded region.

After identifying potential phase markers in the application, we prioritize them for use as region boundaries. Barriers receive the highest priority due to their natural alignment of threads. Prior studies [34] support this, highlighting that partitioning at barrier boundaries prevents aliasing issues and increases accuracy. Next, task loops within a code region receive the next highest priority, marking boundaries between parallel tasks. Lastly, inner loops or ordinary loops are considered for finer granularity, albeit with lower priority, as ordinary loops typically do not act as phase markers in large applications. We then select a subset of these potential markers as region boundaries, considering their priorities. We also ensure that the resulting region sizes are suitable for analysis, meeting both a minimum ($\delta_{min} = 10,000,000$) instruction threshold to capture variations in

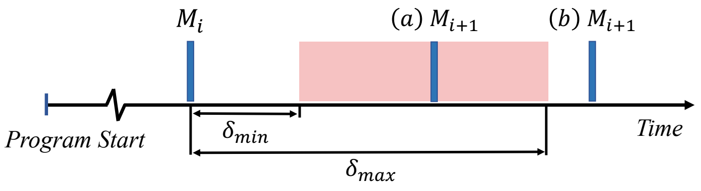


Figure 4.2: The selection of region boundaries (or markers) in an application using Viper. Marker M_i signifies the beginning of the current region with expected region lengths to be between δ_{min} and δ_{max} instructions. M_{i+1} is finally identified in accordance with case (a) or (b) (described in section 4.3.1), which marks the end of the current region.

phase behavior and avoid aliasing issues [14] and a maximum ($\delta_{max} = 100,000,000$) threshold for efficient simulation in a reasonable amount of time.

Region Boundaries

Once the list of potential phase markers is identified, the next step is to collect the highest-priority phase marker from every T ($T \approx$ 1,000,000) instructions. From this highest-priority list, we further select a subset of phase markers to serve as the region boundaries, subject to the constraints that the resulting region sizes approximately fall within the range of [δ_{min}, δ_{max}] instructions as illustrated in Figure 4.2. This is done by employing a greedy algorithm that selects only the highest priority potential phase marker available beyond an interval of δ_{min} instructions but within the next δ_{max} instructions as the next region boundary (Figure 4.2a). If no such marker exists, the first potential phase marker encountered is selected as the next region boundary, regardless of its priority (Figure 4.2b). Region boundaries are represented as triplets: (*Image*, PC_{offset}, *Count*), denoting the object/library, instruction address offset from the Image's base address, and the address's count.

Figure 4.3 shows the classification of all the markers identified by Viper in SPEC CPU2017 applications, along with the chosen markers that serve as the region boundaries. We observe that applications like `638.imagick_s.1`, `657.xz_s.1`, and `657.xz_-`

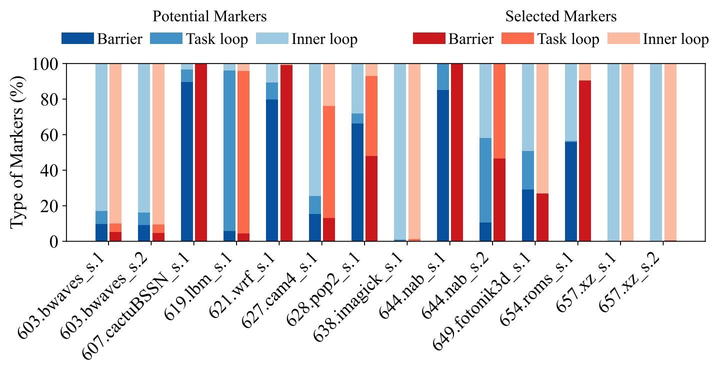


Figure 4.3: The percentage distribution of the type of markers (barriers, task loops, and inner loops) identified in the 8-threaded SPEC CPU2017 benchmarks using train inputs. Potential Markers denote all the available markers in the application, while Selected Markers signify the markers that serve as the boundaries of regions.

`s.2` have a few or no barriers. Therefore, most of the selected markers are ordinary loops that serve as region boundaries. On the other hand, Viper selects as many barrier-bounded regions as possible, as observed in cases such as `607.cactuBSSN_s.1`, `621.wrf_s.1`, `644.nab_s.1`, `654.roms_s.1`, etc.

4.3.2 Region Profiling

Accurately capturing the execution behavior of a multi-threaded code region can be complex as threads synchronize at different points using various synchronization primitives, and the execution pattern of each thread may vary across multiple runs due to differences in memory access patterns [73]. In Viper, we achieve this by using basic block vectors or BBVs as described in prior works [8, 20, 34]. A BBV is the execution fingerprint of a particular interval represented using basic blocks and their counts. BarrierPoint [34] showed that using LRU stack distance vectors (LDVs) along with BBVs can result in better clustering results. An LDV represents a fingerprint of the LRU-stack distance vector for a particular interval, which helps distinguish the regions that execute the same code but have different memory access patterns. We combine BBVs and LDVs on a per-thread level for each region to form per-thread signature vectors or SVs [34].

In order to represent a multi-threaded region, we concatenate the per-thread SVs to form a multi-threaded SV, which captures the amount of parallelism among the threads. The multi-threaded SVs are used for clustering to determine the similarity among the identified regions. We collect all the signature vectors using a high-level emulator.

4.3.3 Determining the Region Similarity

Once the application regions are identified and profiled, the next step is to determine the regions with similar execution characteristics in order to group them together and determine representative regions from among them. This is done based on the profiling information collected for each region which consists of multi-threaded SVs derived from the BBVs and LDVs of all threads, which are projected down to a smaller, fixed dimension. In our experiments, we use 1024 dimensions which could result in higher sampling accuracy and is a good trade-off with respect to the performance. The resulting SVs are then clustered using the k-means [113] clustering algorithm to group similar regions. We use the SimPoint [20] infrastructure to perform the clustering.

4.3.4 Fast and Accurate Fast-Forwarding

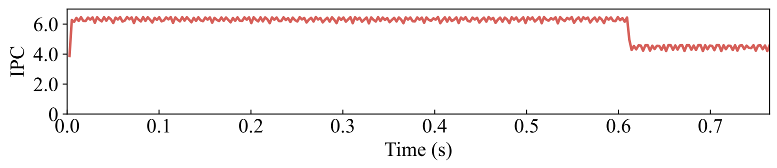
To speed up the simulation, representative regions of the application identified in the clustering stage are simulated in detail, whereas all the other regions are fast-forwarded. Note that this is applicable only for microarchitecture-level simulators, and for RTL-level simulators, we create simulation checkpoints as discussed in Section 4.3.6. During the fast-forwarding phase, we ensure that all of the application threads make similar forward progress in time at regular intervals. This is particularly important because both the functional and timing simulations are disabled during this phase, which can lead to thread orderings that would not typically occur.

4.3.5 The Warmup Challenge

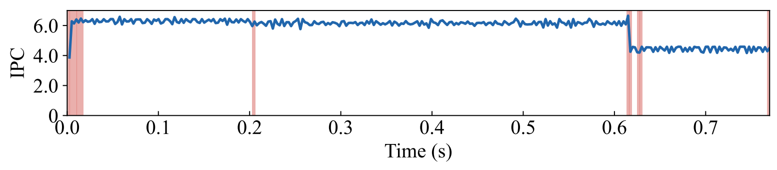
One of the major challenges in sampled simulation is to build an accurate microarchitectural state before the start of every region to be simulated in detail. It is essential to choose a method that is flexible to support different cache configurations and can quickly build the right state, as this can significantly impact the overall speedup achieved. In this work, we choose the memory timestamp record (MTR) [5] warmup technique that can quickly build the cache state at run time. From our experiments, we observed that the harmonic mean of the slowdown due to MTR reconstruction is just 7.97% for SPEC CPU2017 benchmarks using train inputs. We implement MTR to collect the cache line information accessed by each `Load` and `Store` instruction during simulation, ordered in LRU fashion per set. The requests are then injected into the cache in the right order to rebuild the appropriate cache state before the simulation. We focus explicitly on cache warming in simulation, as smaller structures like prefetchers tend to warm up rapidly. For our RTL-level simulations, we simulate a warmup region right before the start of detailed performance measurements of the simulation region.

4.3.6 Generating Simulation Checkpoints

Checkpointing is a widely used technique to capture the system state as a checkpoint and later restore it. We use the application binaries to guide the microarchitecture-level simulations. In order to guide RTL simulations, we create RISC-V full-system checkpoints using MINJIE infrastructure [48]. The checkpoints are restored later to simulate them in parallel on the RTL implementation of XiangShan [48] RISC-V processor using Verilator [49].



(a) The aggregate IPC of the full run.



(b) The aggregate reconstructed IPC using Viper.

Figure 4.4: Plot (a) shows the aggregate IPC of the full run, and plot (b) shows the reconstructed IPC of the `644.nab_s.1` benchmark using Viper. This example shows the benchmark running with *test* inputs using 8 threads. The shaded regions in the plot (b) represent the regions simulated in detail.

4.3.7 Simulation of Representative Regions

The region that lies the closest to the cluster centroid is taken as the representative of that cluster. We identify all the cluster representatives, and these regions are simulated in parallel using the generated checkpoints. In order to show that the proposed methodology works for both microarchitecture-level simulators and RTL-level simulators, we perform our simulations on Sniper (microarchitecture-level) as well as on Verilator (RTL-level). We use the MTR warmup technique to rebuild the right micro-architecture state and inject it into the simulator before the detailed simulation, as discussed in Section 4.3.5. The performance of the overall workload is estimated from the performance obtained from the simulation of the representative regions. Figure 4.4 shows Viper's IPC reconstruction from representatives for the `644.nab_s.1` benchmark (SPEC CPU2017) using test inputs.

4.4 Experimental Setup

In this section, we discuss the experimental setup to evaluate the Viper methodology. We describe the workloads and the platform used to conduct the experiments.

4.4.1 Simulation Tools

We implemented the support for Viper on Sniper [14] version 7.4, which is configured to model Intel's Gainestown microarchitecture, which is the latest hardware-validated microarchitecture available on Sniper. More details on the configuration used for the simulation are shown in Table 4.1. We modified the front-end of Sniper to support Viper's region specification. However, we expect that implementing this region specification support on other software simulators like gem5 [6, 135] or ZSim [93] is possible. For RTL-level simulations, we use Verilator [49] to simulate the RISC-V processor XiangShan [48] using the checkpoints generated using the MINJIE platform. In this work, we generate the simulation checkpoints using NEMU [48]. The methodology is also applicable to other RTL simulators (like VCS [94]) if corresponding checkpoints are generated.

Table 4.1: The configuration of Gainestown microarchitecture.

Component	Parameters
Processor	8 cores, 2.66 GHz, 128-entry ROB
Branch predictor	Pentium M, 8 cycles penalty
L1-I/D	32KB, 4/8 way, LRU
L2 cache	256KB, 8 way, LRU
L3 cache	8MB per core, 16 way, LRU

4.4.2 Benchmarks Used

SPEC CPU2017 benchmark suite [117] is a widely used collection of applications used for computer architecture evaluation and exploration. The benchmarks are written in C, C++, Fortran, or a combination of these programming languages. We use the multi-

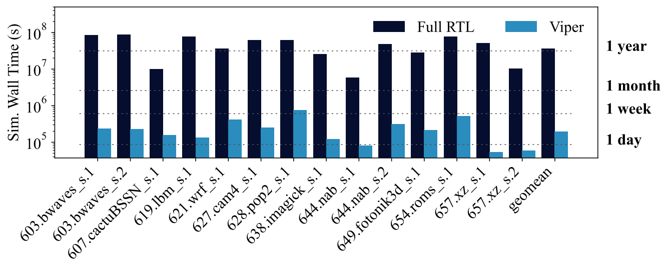


Figure 4.5: A comparison of the estimated wall time to simulate SPEC CPU2017 benchmarks using train inputs and 8 threads for the full simulation (Full RTL) and Viper. We use the simulation rate of XiangShan on Verilator and assume parallel simulation of all the representative simulation checkpoints.

threaded OpenMP-based subset of the SPEC CPU2017 benchmarks that are enabled for multi-threaded execution. We use the *speed* version of these benchmarks configured to use eight statically scheduled threads. Note that these are multi-threaded benchmarks that synchronize and share memory. SPEC CPU2017 benchmarks use three different inputs: *test*, *train*, and *reference* (ref). We configure the SPEC CPU2017 benchmarks to use the train inputs for our evaluation. These benchmarks are compiled for x86-64 architecture using GCC 6.4.0 and gfortran with the `-O3` optimization compiler flag. The multi-threaded benchmarks are configured to use *passive* OpenMP thread wait policy.

4.4.3 Analysis Tools

We use Intel's Pin [116] to build the analysis and profiling tools (Pintools) that we use for this methodology. We also utilize the Dynamic Control Flow Graph (DCFG) tool [123] included in the Pin distribution to collect potential markers that are used to identify regions. DCFG collects the trace information of the application, which can be utilized by implementing a pintool to detect barriers and loops that can act as region markers.

4.5 Evaluation

In this section, we describe the experimental results of the proposed methodology. We also present the key factors that affect the performance of the methodology.

4.5.1 Comparison with State-of-the-Art

We first show the estimated wall time of full RTL simulation and Viper using XiangShan on Verilator. Then we evaluate the accuracy and performance of Viper using Sniper and compare it with LoopPoint (Chapter 3), the state-of-the-art sampled simulation methodology for multi-threaded applications [8]. We then conduct detailed studies on how region length affects speedup and accuracy. We do not evaluate the accuracy of Viper on XiangShan as full RTL simulation takes more than a year for SPEC CPU2017 benchmarks using train inputs. In our experiments, we calculate the average value by taking the geometric mean of the values across all benchmarks.

RTL-level Simulation. Figure 4.5 shows the total time required to simulate SPEC CPU 2017 benchmarks using Verilator. Unlike prior works, we observe that the sampling efficiency is bounded by the largest region identified by the sampling methodology. Viper could significantly reduce the simulation time of these large workloads from more than a year to less than a week or even a day in some cases.

Accuracy. Viper achieves similar or better error rates as compared to prior multi-threaded sampling methodologies like BarrierPoint or LoopPoint. To measure the sampling accuracy of the proposed methodology, we compare the simulation runtimes T_{full} obtained from the full simulation and T_{sample} obtained from the sampled simulation. The absolute runtime prediction error Δ can be represented as $\Delta = |1 - \frac{T_{sample}}{T_{full}}|$.

Figure 4.6 shows a comparison of absolute run time prediction errors with Viper and LoopPoint obtained for the 8-threaded SPEC CPU2017 benchmarks using train inputs.

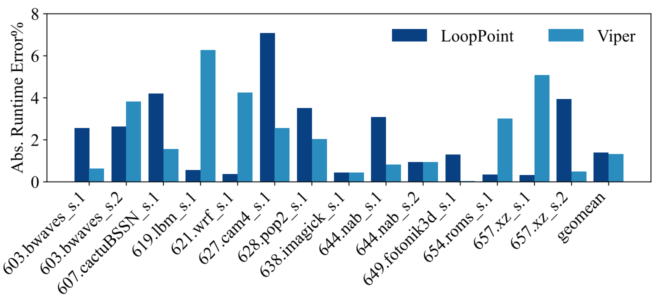


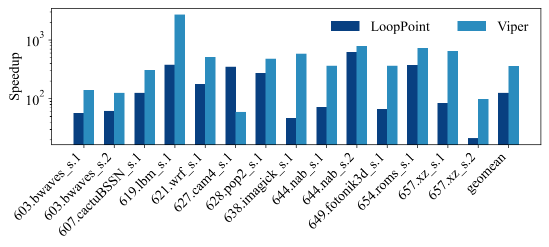
Figure 4.6: A comparison of the absolute runtime prediction error for Viper and LoopPoint. We use SPEC CPU2017 benchmarks that use train inputs and 8 threads.

Viper performs similarly to LoopPoint while achieving lower maximum and average (1.32%) errors. The results validate that choosing regions that are aligned to application phases, while potentially much smaller in length, can achieve better accuracies.

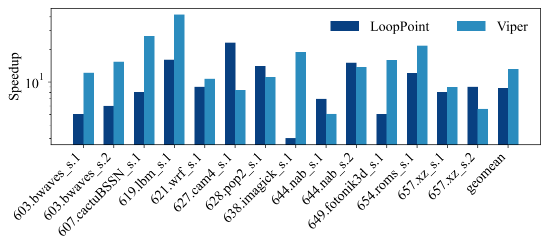
We evaluate the performance of Viper for 16 threads using the same set of SPEC CPU2017 benchmarks along with train inputs (except for `657.xz_s.1` and `657.xz_-s.2`, which run only with one thread and four threads, respectively). For the rest of the benchmarks, we observe an average absolute error in the run time of 1.79%. The maximum error that we observe is 5.29% (for `603.bwaves_s.2`), whereas the minimum error is 0.01% (for `638.imagick_s.1`).

Speedup. The speedup is the ratio of the wall time required for the full simulation to that of the sampled simulation. We define serial speedup as the speedup achieved when the samples are simulated sequentially, whereas parallel speedup is the speedup achieved when the samples are simulated in parallel.

We compare the speedup of the proposed methodology with LoopPoint as shown in Figure 4.7a (parallel speedup) and Figure 4.7b (serial speedup). Viper outperforms LoopPoint in all but one case for parallel speedup, as shown in Figure 4.7a. We observe that Viper samples fewer but larger loop-bounded regions compared to LoopPoint for `627.cam4_s.1` resulting in longer simulation times. This is because `627.cam4_s.1` has



(a) Parallel Speedup



(b) Serial Speedup

Figure 4.7: A speedup comparison of LoopPoint and Viper for the 8-threaded SPEC CPU2017 benchmarks using train inputs.

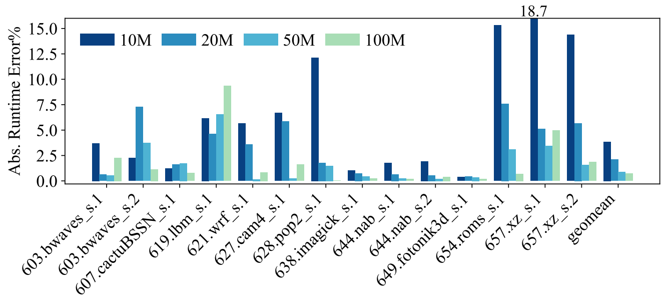


Figure 4.8: Runtime prediction error for 8-threaded SPEC CPU2017 benchmarks using train inputs for different region sizes.

larger loops and unlike LoopPoint, Viper does not split applications at random loops. In the case of serial simulations, Viper outperforms LoopPoint in most cases (9 out of 14) in Figure 4.7b. The maximum serial speedup achieved by the proposed methodology is 6.23×. The primary reason behind achieving more speedup is that the region size of Viper corresponds to the phase boundaries of the application, unlike the fixed region sizes in LoopPoint.

4.5.2 Varying Region Sizes

We use Viper methodology to illustrate the experimental results using different region sizes to show their effect on error rates. We also show the importance of choosing regions inherent to the application structure instead of fixed-size slices. We use Viper to select

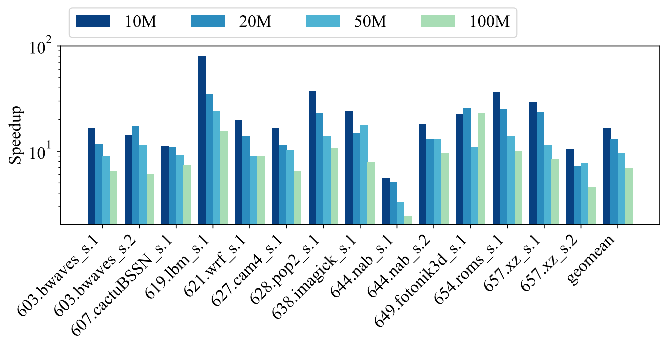


Figure 4.9: The speedup achieved for 8-threaded SPEC CPU2017 benchmarks using train inputs. Viper is used to identify regions of fixed sizes.

fixed-size regions of 10 million, 20 million, 50 million, and 100 million instructions.

We show the accuracy in predicting the run time of each of the benchmarks. As shown in Figure 4.8, there is no correlation between the region sizes and accuracies. For example, in the case of `628.pop2_s.1`, `638.imagick_s.1`, or `654.roms_s.1`, the error decreases with an increase in region size. However, larger region size does not always yield better accuracy in some other cases. For example, benchmarks like `603.bwaves_s.1`, `621.wrf_s.1` and `627.cam4_s.1` achieve their best accuracies when the region size is around 50 million. We infer from the experiment that there is no general region size that can be used for every application, which motivates us to choose application-dependent regions.

The average error of Viper-100M (regions of size $\approx$ 100M) is 0.74%, whereas that for Viper is 1.32%. Although using a larger region size yields a slightly better average error, Viper consistently achieves better accuracies for most benchmarks.

Speedup. As Figure 4.9 shows, the speedup is larger for smaller atomic region sizes in most cases (although the errors can be higher). For smaller region sizes, clustering allows there to be fewer instructions to be simulated in detail overall, which allows for a larger speedup. However, in certain cases, the number of regions to be simulated in detail can be much more when the region sizes are smaller. For example, `649.fotonik3d_s.1` achieves a smaller speedup at region size *20M* when compared with that of region size *50M*. Comparing Figure 4.7b with Figure 4.9, we observe that the speedup achieved using Viper-100M is similar to that of LoopPoint.

4.6 Conclusion

In this work, we propose a novel sampled simulation methodology and infrastructure called Viper that shows significant improvement in performance over the existing methodologies which is applicable to both microarchitecture-level and RTL-level simulators. Viper is both a fast (358× speedup on average) and an accurate (with an average error of just 1.32%) simulation methodology as evaluated with the multi-threaded subset of SPEC CPU2017 benchmarks using train inputs.

Chapter 5

Pac-Sim: Simulation of Multi-threaded Workloads using Intelligent, Live Sampling

If you want to find the secrets of the universe, think in terms of energy, frequency, and vibration.

— Nikola Tesla

Modern systems are becoming increasingly complex and dynamic. With the high level of dynamic optimizations in these systems, it is crucial to simulate next-generation multi-core processors in a way that can respond to system changes and accurately determine system performance metrics. We propose Pac-Sim, a novel sampled simulation methodology that overcomes the limitations of traditional approaches by enabling fast and accurate simulations even in the presence of dynamic hardware and software behavior. This is achieved through live sampling, eliminating the need for upfront workload analysis.

5.1 Introduction

Computer architecture research heavily relies on simulations for design space exploration. However, microarchitectural simulation can become extremely time-consuming, particu-

Alen Sabu and Changxi Liu contributed equally to this research.

larly as the complexity of modern architectures has increased over time. This is especially true in the post-Dennard era, where architectures are rapidly evolving to incorporate complex dynamic optimization techniques at both the hardware and software levels to improve system performance gains at runtime. Hardware-based dynamic techniques such as dynamic cache reconfiguration [38, 39, 40], DVFS [35, 36, 37], TurboBoost [41] and power management [140] techniques trigger optimizations based on dynamically identified hardware states (such as core frequency, cache reuse distance, etc.) to improve both energy-efficiency and overall performance of the system. Similarly, runtime information at the software level can be used to dynamically optimize code execution, to further enhance the system performance. Some of the recent efforts on software-based optimization focus on dynamically scheduling tasks among threads [42, 141] to ensure efficient resource utilization and employing just-in-time (JIT) compilation techniques [142, 143, 144, 145] that generate high-performance instructions to optimize program execution online. However, since these techniques utilize dynamic system state information in order to deploy optimizations at runtime, the execution behavior of an application (and, therefore, its performance) may vary greatly across multiple executions. This inherent variability may lead to an inaccurate performance evaluation when using existing simulation methodologies.

Conventionally, sampled simulation has served as a reliable and efficient technique to accelerate the performance estimation of multi-threaded workloads. In order to achieve these results, most prior works relied on either (i) profile-driven sampling [8, 20, 34] or (ii) statistical sampling [2, 146]. Profile-driven sampled simulation methodologies such as SimPoint [20], BarrierPoint [34], and LoopPoint [8] (Chapter 3) split the execution of an application into a series of repeatable regions and cluster them based on their execution features. A representative element from each cluster is then analyzed or simulated in order to extrapolate the performance of the entire application. However, these methodologies incur a significant cost in terms of the preprocessing effort that is needed to identify representative regions. These costs include the time required to profile and cluster the execution features of all application regions, along with the storage required. While it has been previously argued that these costs are a one-time investment and will be amortized over multiple runs, this argument does not necessarily hold for systems that optimize code execution dynamically. In such cases, the program execution paths followed by an application may vary considerably due to changes of hardware and software parameters that are being optimized. Therefore, the profiling information collected for one specific run would not necessarily extend to the program execution paths followed in the subsequent runs.

On the other hand, methodologies such as SMARTS [2] and PCantorSim[146] rely on statistical sampling techniques to speed up simulation-based performance measurements while meeting a given error bound. Unlike profile-driven sampling, these methodologies require minimal preprocessing and do not rely on the reproducibility of program execution paths. They are thus applicable to dynamically optimized systems. However, the simulation speedups achieved using these techniques are considerably lower than the profile-driven counterparts, and adjusting settings to achieve higher performance could lead to high errors.

For the above-mentioned reasons, it becomes challenging to sample and simulate generic multithreaded applications for dynamic hardware and software using existing methodologies. Architects need a simulation methodology that can dynamically adapt to changes in the system at runtime while accurately estimating the application's performance without relying on the reproducibility of its execution. To this end, we propose Pac-Sim, a novel sampled simulation methodology that can, at runtime, efficiently analyze and sample the application to select the representative regions to be simulated in detail. The result is a methodology that enables both fast and accurate performance evaluation without the need for up-front analysis. We accomplish this by making use of an intelligent online predictor and classifier that quickly and accurately decides whether the upcoming region needs to be simulated in detail.

In short, we make the following contributions:

i. We propose Pac-Sim, a methodology that goes beyond prior sampled simulation techniques to be the first to allow for dynamic hardware and software support. The methodology requires no upfront analysis and relies on an online predictor for sampling decisions enabling the fast analysis of co-designed workloads.
ii. We experimentally demonstrate that Pac-Sim consistently improves performance in terms of speedup and accuracy over prior works that use offline profiling, as Pac-Sim utilizes a lightweight but accurate online sampling technique.
iii. We provide an extensive evaluation of Pac-Sim using standard benchmarks to compare against prior works and demonstrate best-in-class accuracy (average error of 1.63%). For the SPEC CPU2017 benchmarks (train inputs) running eight threads, we show a maximum serial speedup of 123.32× (26.09× on average) and a maximum parallel speedup of 523.5× (210.3× on average).
iv. Finally, we showcase several case studies demonstrating that Pac-Sim is applicable to a number of research scenarios, including (but not limited to) the investigation of optimization techniques such as dynamically scheduled software and improving research into dynamic hardware and hardware-software co-design.

The rest of the chapter is organized as follows. In Section 5.2, we discuss the relevant background and the challenges involved in the simulation of dynamic applications on modern architectures. Section 5.3 presents the Pac-Sim methodology in detail. We then describe the experimental infrastructure in Section 5.4, followed by an extensive evalu-

We use the terms “online” and “offline” to distinguish between events that occur during and prior to the simulation of an application, respectively.

Pac-Sim has been open-sourced and can be found at `https://github.com/snipersim/snipersim`.

Table 5.1: This table summarizes previously proposed sampled simulation methodologies for both single-threaded and multi-threaded applications. We categorize these methodologies into two main groups: *Profile-driven* and *Statistical*. The table also identifies the *Analysis Type* used by each methodology. Notably, some methodologies require an upfront analysis or profiling phase to extract application-specific characteristics. Additionally, the table indicates which methodologies are amenable to parallel simulation, which determines the maximum speedup of the methodology. The field *Warmup* shows the warmup technique used to reconstruct the microarchitectural state at the beginning of the detailed simulation.

	Methodology	Analysis Type	Parallel Simulation	Warmup	Applicability
Profile-driven	Simpoint [20]	●		Prev Region	Single-threaded
	LiveSim [27]	●		Checkpoint	Single-threaded
	BarrierPoint [34]	●		Prev Region	Multi-threaded
	TaskPoint [43]	●		Prev Region	Task-based
	LoopPoint [8]	●		Prev Region	Multi-threaded
Statistical	SMARTS [2]	◐		Functional	Single-threaded
	SimFlex [30]	◐		Checkpoint	Multi-program
	Time-Based Sampling [32, 33]	◐		Functional	Multi-threaded
	Pac-Sim (*this work*)	○		Statistical	Multi-threaded

● Online Profiling ◐ Offline Analysis ○ Offline Profiling

ation of Pac-Sim in Section 5.5 along with case studies to demonstrate the applicability of the proposed methodology. Finally, we present the related work in Section 5.6 and conclude the chapter in Section 5.7.

5.2 Simulating Modern Architectures

In this section, we provide the necessary background and prior work of sampled simulation. Table 5.1 summarizes the widely used sampled simulation methodologies applicable to CPU workloads. We also discuss the challenges in simulating modern workloads and how the existing sampling methodologies are insufficient to address them.

Sampling Single-threaded Workloads. Sampling and workload reduction tech-

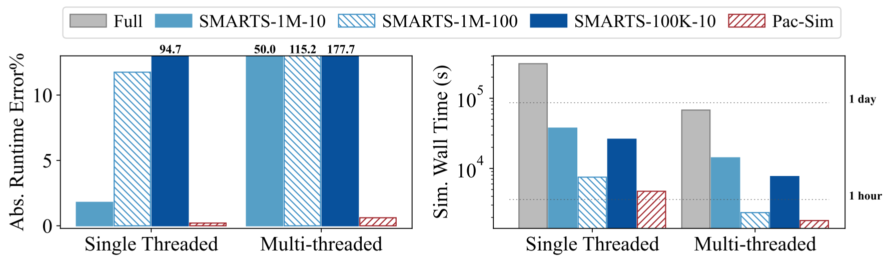


Figure 5.1: Performance comparison of Pac-Sim with SMARTS [2] in different settings for the SPEC benchmark `644.nab_s.1` (multi-threaded version uses 8 threads). The left graph shows the comparison of runtime prediction errors using different sampled simulation techniques, whereas the right graph shows the overall simulation time (running on a parallel simulator). Both figures use lower-is-better metrics. *SMARTS-A-B* repeatedly switches between a single detailed simulation region of length A and B fast-forward regions of length A.

niques are extensively utilized in computer architecture research for the purpose of program characterization and to reduce simulation time. Sampling methodologies allow for the evaluation of a subset of the workload (a representative sample) in detail that can be used to reconstruct the performance of the whole workload accurately. These methodologies split the workload into different regions (or slices) based on predetermined conditions in order to identify a representative sample. Prior works that explored CPU workload sampling, like SimPoint [20] and SMARTS [2], tend to utilize fixed instruction counts to determine regions. However, instruction count-based techniques could lead to inconsistent and, therefore, invalid regions [31, 32, 73]. Some previous works [26, 123] proposed software phase markers that identify procedure and loop boundaries that correlate with phase changes to mark region boundaries instead of using fixed-sized regions.

Sampling Multi-threaded Workloads. SimFlex [30], which selects sampling units for the simulation of server throughput workloads, does not appear to be generally extensible to synchronizing multi-threaded workloads [32]. In the presence of synchronizing threads, the application performance tends to vary more frequently [31, 32]. Sampling methodologies such as SMARTS, SimFlex, and PCantorSim [146] rely on statistical confidence, and adjusting settings to achieve higher performance could lead to high errors, as shown in Figure 5.1. Time-based Sampling methodologies [32, 33] are the first to address the problem of sampling synchronizing multi-threaded applications. These methodologies are generally applicable and are suitable for the sampled simulation of dynamic systems. However, Time-Based Sampling techniques are the slowest of all the sampling techniques, and as a result, they are not practical for handling long-running workloads. On the other hand, methodologies like BarrierPoint [34], TaskPoint [43], and LoopPoint [8] select specific program constructs, such as barrier synchronization primitives, task instances, and loops, respectively, to identify periodic behavior. This enables the utilization of representative-sized regions for simulation, regardless of the program's length.

Feature Vectors. Profiling captures feature vectors to characterize the execution behavior of an application across regions. Previous works have introduced several microarchitecture-independent feature vectors, of which basic block vectors (BBVs) [20, 52] are the most widely used for performance characterization. Lau et al. [18] showed a strong correlation between BBVs and region performance. Apart from BBVs, Shen et al. [75] introduced LRU stack distance vectors (LDVs) [147] to summarize program behavior for different regions. BarrierPoint [34] combines BBVs and LDVs into a signature vector (SV) in an attempt to represent more accurate features of multi-threaded applications. Furthermore, Cotson [80] and Dynamic sampling [148] record statistics such as the number of instructions executed, memory accesses, exceptions, bytes read or written, etc., in order to plot the feature of a given region. Unfortunately, none of these offline techniques can handle runtime optimizations that impact applications.

Overheads. Figure 5.2 illustrates the overhead of profiling data for LoopPoint (the evaluation was performed using the LoopPoint tools [149]) methodology, indicating that profile-driven methodologies incur significant overheads. When it is required to emulate

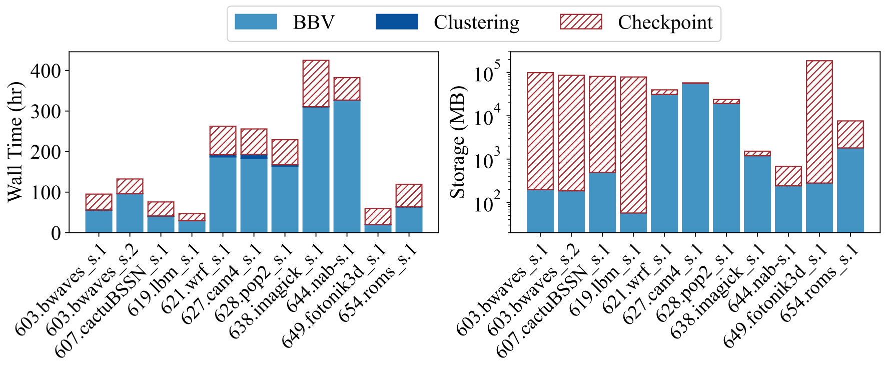


Figure 5.2: The figure shows the resource utilization of a recent multi-threaded sampled simulation technique, LoopPoint, for the SPEC CPU2017 benchmarks with the `ref` inputs running eight OpenMP threads. The graph on the left shows the time required to generate the profiling data (with checkpoints stored as pinballs [3]), whereas the graph on the right shows the amount of storage required.

an architecture (for example, simulating ARM or RISC-V binaries on x86) during profiling, it is necessary to resort to functional simulation to gather feature vectors, which can be a time-consuming process. For instance, Sandberg et al. [78] demonstrated that it took up to a month to generate profile data for SPEC CPU2006 benchmarks using simulators like gem5. Prior works [8, 34] argue that this overhead is amortized over multiple runs as the profiling results will be reused. However, this assertion does not hold in the case of dynamic software and hardware where certain performance optimization decisions are made using runtime information. For example, profiling for asymmetric cores, such as the *big.LITTLE* cores is challenging as the operating frequency (and other dynamic hardware settings) of each core may not be known during profiling. Handling and storing simulation checkpoints can be a daunting task. For instance, x86 architecture checkpoints like ELFies [47] require a significant amount of storage space. Checkpoints are often specific to a simulator or are tied to particular software/hardware configurations. Microarchitectural checkpoints, requiring detailed hardware information like cache states, are specific to the underlying hardware configuration.

Hardware and Software Dynamism. Researchers have introduced several dynamic optimization techniques in hardware and software to achieve higher performance and reduce power consumption. Techniques such as dynamic voltage and frequency scaling (DVFS) and cache reconfiguration have been developed to adjust the hardware state in response to executed instructions and active processes. Software optimization techniques [142, 143, 144, 145] generate optimized code sequences at runtime. Additionally, dynamic scheduling techniques [42] have been developed for multi-threaded applications. In such cases, profile-driven sampling methodologies could show different performance results for each execution. Methodologies such as trace-based simulations [150] or deterministic replay platforms [77] can guarantee consistent performance across multiple executions but demand extensive profiling and large storage resources. Dynamic hardware events, such as changes in core frequency, cache size, etc., can be unknown during profiling. These events, when they are performance and power-dependent, become difficult to predict. Sherwood et al. [22] utilize a Markov predictor to predict the phase behavior at runtime. Kihm et al. [151] propose switching to the detailed simulation mode whenever the BBV variance exceeds a specified threshold. However, these methods have only been demonstrated with single-threaded applications as the phase behavior of synchronizing multi-threaded applications varies frequently due to the interaction of threads.

Requirements for Fast and Accurate Simulation.

Sampled simulation techniques that do not require upfront application analysis demonstrate significant potential under dynamic software and hardware constraints. The inherent variability of dynamic software behavior renders a single analysis insufficient, while the unpredictable nature of modern hardware compromises the reliability of upfront analysis. Additionally, the overhead due to the detailed application analysis becomes a bottleneck for researchers engaged in fields like hardware-software co-design. Therefore, it is imperative to leverage the best aspects of SimPoint-like and SMARTS-like methodologies to achieve optimal simulation efficiency and accuracy. In our approach, we integrate application analysis to guide sampled simulations, similar to SimPoint-like methodologies, but without the need for upfront preprocessing, as seen in SMARTS-like methodologies. In our work, we capture the dynamic information of the software and hardware to make intelligent simulation decisions through online learning. Therefore, our methodology is capable of handling hardware state changes, software dynamism, and other factors influencing application performance. To achieve optimal performance with online analysis, efficient and lightweight profiling, clustering, and warmup techniques are essential.

In short, to quickly estimate the performance of multithreaded applications running on next-generation hardware, a sampled simulation methodology is needed that can dynamically adapt to changes in the system at runtime while accurately determining relevant performance metrics. In Section 5.3, we will provide a thorough discussion of these aspects and present our solution for fast and accurate simulation.

5.3 The Pac-Sim Methodology

In this section, we describe our proposal for an end-to-end sampled simulation methodology, Pac-Sim (see Figure 5.3), that supports both dynamic hardware and software without requiring up-front workload analysis. Pac-Sim consists of five main stages: Marker Detection, Region Profiling, Clustering, Prediction, and Simulation, which are all carried out online. We have carefully designed each of these stages to minimize the runtime overhead of the methodology while maintaining the sampling accuracy. An important advantage of an online sampled simulation methodology like Pac-Sim is its ability to accurately determine the execution profile of an application without relying on the reproducibility of a program's execution paths. This characteristic allows Pac-Sim to accurately analyze and evaluate dynamic multi-threaded applications, accounting for any performance variability that may occur at runtime.

Pac-Sim operates by making use of the program structure and runtime hardware state to identify the regions and their boundaries online. Each of these region boundaries or *markers* defines the ending of the current region and the beginning of the next region (Section 5.3.1). Once a marker is identified, Pac-Sim collects the profiling data and simulation results of the current region (Section 5.3.2) and clusters it with the previously identified regions to determine its cluster ID (Section 5.3.3). This cluster ID is added to the program execution history, which is then used by the *Predictor* (Section 5.3.4) along with the current marker and hardware state to predict whether the next region needs to be simulated in detailed mode or fast-forward mode.

While we only demonstrate the effectiveness of Pac-Sim in estimating the performance of synchronizing multi-threaded workloads in this work, our methodology has the potential to support a variety of modern workload classes, such as cloud and mobile applications, and could also be implemented for full system simulations. However, in such cases, various factors must be taken into consideration, such as kernel and driver performance, which can significantly impact the overall efficiency of the workloads. In this work, we focus on user-space workloads, and enabling support for the above-mentioned use cases is out-of-scope in this context, which we leave for future work.

5.3.1 Online Region Detection

Previous research [8, 26, 34] has shown that certain program constructs, such as barriers or loops, can be utilized to characterize the phase behavior of multi-threaded applications by splitting them into a series of individually analyzable regions. Since barriers represent the global synchronization points within a program execution, all threads align at these points, making them natural boundaries for application regions. However, relying solely on barriers to split an application may not be ideal, especially in the presence of large

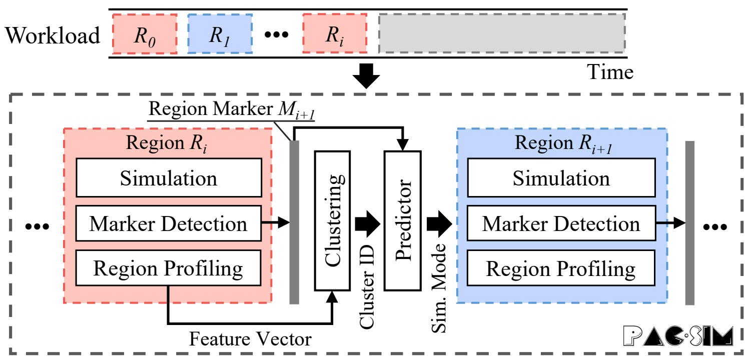


Figure 5.3: The overall workflow of Pac-Sim methodology. At any given time, the regions of a multi-threaded workload till R_i are identified (as shown above). First, Pac-Sim monitors the application code structure to determine an appropriate region marker M_{i+1}, which marks both the end of the region R_i and the start of the region R_{i+1}. Next, the feature vector and simulation results for R_i are collected, and a prediction mechanism determines the simulation mode for region R_{i+1}. Finally, region R_{i+1} will be simulated, either in detail or in fast-forward mode.

inter-barrier regions, as this can lead to low simulation speedups as representatives can still be too large to complete detailed simulation in a reasonable amount of time. In contrast, loops offer a finer level of granularity, allowing for greater control over the size of regions. Typically, multi-threaded applications consist of both loops and barriers in varying proportions. The online *Marker Detector* combines both of these program constructs to effectively split multi-threaded applications into regions with sizes that are well-suited for clustering while also avoiding aliasing [152]. The Marker Detector uses the following approach in order to identify the barrier- and loop-based markers online:

Barriers. Typically, a multi-threaded region begins with a `fork` call, which spawns additional worker threads and ends with a `join` call, which terminates the current thread and synchronizes with other threads. A new region is triggered at events of thread creation and termination, as regions with different active threads have different performances. For multi-threaded programs that use the OpenMP library, special function names are generated depending on the compiler used. We utilize this information in the online Marker Detector to quickly and efficiently detect barriers with low overhead.

Loops. Both loop and conditional statements use conditional branch instructions, with the target address usually given as an offset from the instruction pointer. The key difference between the two statements is that the offset of the branch instructions in a loop statement is usually negative, whereas that in a conditional statement is positive. In most cases, selecting conditional branches with negative offsets is sufficient to identify loop markers [123]. In the rare case of exceptions, Pac-Sim predicts the simulation mode of the next region to be detailed mode. We also make sure to disregard spinloops from our analysis.

As an application executes, the Marker Detector identifies markers online, splitting the application into multiple regions. While doing so, it also monitors the region sizes to ensure they fall approximately within the bounds of δ_{min} and δ_{max} instructions. A minimum number of instructions, δ_{min}, is necessary to capture the frequent variations in the multi-threaded program behavior and accurately cluster the obtained regions. Whenever the Marker Detector chooses barrier-based markers as region boundaries, the size of the region can be as small as δ_{min} instructions but no larger than δ_{max} instructions. Otherwise, the Marker Detector chooses the first loop-based marker it encounters beyond δ_{max} instructions as the next region boundary. For loop-bounded regions, it is necessary to keep region sizes large enough to avoid aliasing [32]. In our experiments with fixed region sizes of 10 million, 20 million, 50 million, and 100 million instructions, the SPEC CPU2017 benchmarks showed average error rates of 6.9%, 3.3%, 1.8%, and 1.8%, respectively. We set the lower bound δ_{min} to be 20 million to ensure sampling accuracy and the upper bound δ_{max} to be 50 million for better performance.

Hardware State. The Marker Detector also monitors the hardware state of the simulated system. If it detects changes, the current region is ended at the next marker so that each region has a consistent hardware state. Once a marker is detected, the program counter (PC) and the hardware state of the simulated system are collected and stored corresponding to the marker. The collected hardware state includes the system parameters, like processor frequency, cache size/configuration, power management techniques, etc., that can be configured during runtime.

5.3.2 Online Region Profiling

Conventionally, BBVs have been used to characterize the execution behavior of code regions, as they have been shown to exhibit a strong correlation with the region's performance [18]. BBVs record the execution counts of each basic block (i.e., code blocks with single entry and exit points) within a given code region. The number of dimensions for a BBV depends on the number of basic blocks executed, which could range anywhere from thousands to even millions for very large applications. This presents a major challenge for online analysis of BBVs as the time and effort required for this stage would significantly increase as the vector dimensionality increases. SimPoint [20] uses random linear projection [153] to overcome this problem. However, this method is not suitable for our online algorithm as the matrix-vector multiplication operations involved could introduce significant runtime overheads.

To overcome these issues, we propose a fast online BBV generation technique (illustrated in Figure 5.4). Rather than creating a fixed-size BBV for each region, we use an online projection technique to generate fixed-size vectors BBV_i' for each basic block BB_i, where the elements of BBV_i' are computed by multiplying the instruction count of a basic block with the hash results of its program counter (PC) value. We use the hash function `drand48()`, which generates pseudo-random numbers for an integer value input. The initial four dimensions of the online BBV are determined using the hash values utilizing inputs PC, PC+1, PC+2, and PC+3, respectively. The values of the subsequent four dimensions are generated using the output of the preceding four dimensions as inputs to the hash function. We experimentally determined that using 16 dimensions adequately captures the representation of a region using the online BBV. The resultant BBV_i'

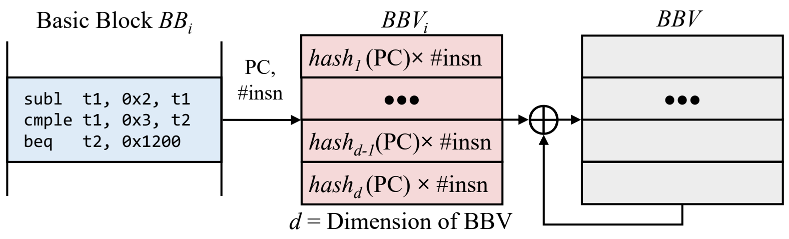


Figure 5.4: The figure shows the workflow of online BBV generation. Whenever a basic block BB_i is encountered, a corresponding execution fingerprint BBV_i is generated using hash functions applied to the program counter of BB_i and the number of instructions it contains. $hash_1$ to $hash_d$ are d distinct hash functions, where d is the dimension of the BBV. The BBV for each region is obtained by accumulating all BBV_is that belong to the region.

vectors are then accumulated to obtain the per-thread BBV (BBV'_{online}) for the given region, which can be represented as:

$$BBV'_{online} = \sum_i BBV'_i = \sum_i (BBV_i \cdot M_{proj}),$$

where the values of the elements in M_{proj} are generated using hash functions as mentioned above. This BBV'_{online} for a region is analogous to the BBV utilized in SimPoint, which is obtained through random linear projection. The projected down BBV used in SimPoint, $BBV'_{offline}$, is obtained from the dot product of the actual BBV of the region and projection matrix M_{proj}:

$$BBV'_{offline} = BBV \cdot M_{proj} = \sum_i (BBV_i \cdot M_{proj}).$$

We then normalize these per-thread BBVs and concatenate them into a single global-BBV vector to represent the software feature of a given multi-threaded code region. In Pac-Sim, it is necessary to maintain the online BBV to capture the dynamic program behavior. Using online BBVs to represent regions eliminates the need to perform computationally intensive dimensionality reduction techniques during simulation.

5.3.3 Determining Region Similarity

Pac-Sim employs an online clustering mechanism to group regions with similar execution behavior based on the feature vectors collected for each region in the online profiling stage. The clustering, which is done at the end of simulating each region, is required for the learning process of the Predictor. Prior works, like SimPoint [20], cluster feature vectors using the k-means algorithm [113]. However, k-means uses an iterative refinement technique that is computationally intensive, and therefore, a more efficient algorithm might be better suited for online analysis.

In order to reduce this computational load and enable real-time clustering, we devise an alternative technique for clustering feature vectors (i.e., global-BBVs) in Pac-Sim. In our technique, we maintain two separate queues: (i) detailed queue and (ii) fast-forward queue. The detailed queue includes the BBVs corresponding to the regions that have been simulated in detail, while the fast-forward queue includes those corresponding to the regions that have been fast-forwarded. When a new BBV is recorded, it is first compared with the BBVs in the detailed queue. If its distance from any of these BBVs is less than the specified threshold θ, then we return the cluster ID of the closest region. If there is no region whose distance is less than θ, we repeat the same procedure with the regions in fast-forward queue. If we still don't find similar regions, we assign a new cluster ID for the current region and insert it into the BBV queue corresponding to its simulation mode. In our experiments, we set $\theta = 0.05$ to ensure a reasonable simulation accuracy while maintaining high speedups. To further improve the efficiency of our clustering technique, we incorporate the triangle inequality optimization [154] into our algorithm, which can skip redundant BBV distance calculations. We use Euclidean distance for all BBV distance calculations.

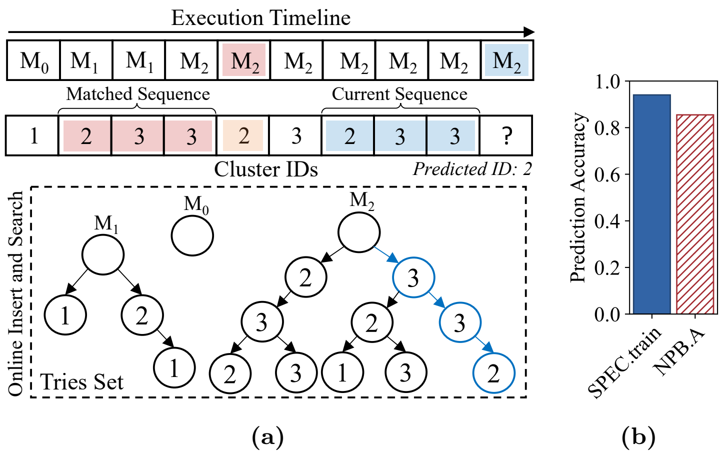


Figure 5.5: The predictor utilizes the trie [4] data structure to quickly predict the cluster ID of the next region by searching for a similar history with the same region start marker M_i. In this example, the cluster ID of the next region is predicted to be 2 since the prior region with the cluster ID of 2 has the same start marker M_2 and the longest matching sequence ($3 \rightarrow 3 \rightarrow 2$). Plot (b) shows the accuracy of the predictor for different benchmark suites.

5.3.4 Prediction Mechanism

Pac-Sim employs a *Predictor* – an online prediction mechanism that leverages region markers, execution history, and hardware state to predict the phase behavior of the next region in an application and decide its simulation mode at runtime.

Region Markers. The Marker Detector identifies PC-based region markers that act as the boundaries of the regions. In certain cases, using region markers to classify regions is effective for applications where the same part of the code displays similar phase behavior, as in the case of `619.lbm_s.1` and `644.nab_s.1` using train inputs.

Execution History. When executing the same part of the source code, differences in memory access patterns, branching, etc., can result in varying phase behavior at runtime. We, therefore, make use of execution history, which is a sequence of the cluster IDs of prior regions, to predict these differences in the phase behavior among applications.

Hardware State. Pac-Sim takes into account the state of the simulated system, such

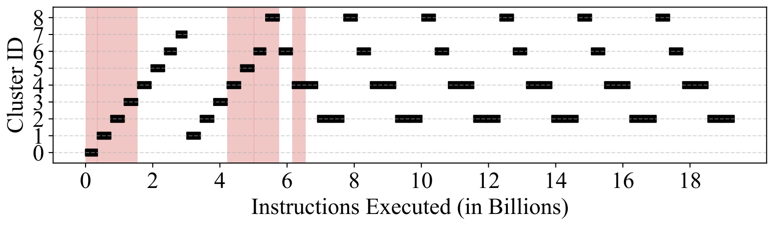


Figure 5.6: The graph shows the regions identified using Pac-Sim for the NPB benchmark `ft`, grouped together with the respective cluster they belong to. The shaded portion represents the regions that are simulated in detail.

as core frequency, while executing each region. *Predictor* predicts the next region to be detailed mode if there are no prior similar regions with the same hardware state. The Predictor decides the cluster ID of the next region by choosing the cluster ID of the previous region with the same region marker and has the longest matching sequence. For the regions that do not have a previous region with the same start marker or the same history, Pac-Sim enables detailed execution for that region. Then it decides the simulation mode of the next region by checking whether prior regions with cluster ID and the current hardware state are simulated in detailed mode. This history is learned online and is updated every time Pac-Sim finishes simulating a region.

To accelerate this stage for large application lengths, we further optimize our clustering algorithm to reduce its average search time complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. This is achieved by maintaining the execution history in a *trie* [4] data structure, with a maximum depth of 16, which allows for more efficient *search* and *insert* operations. In Pac-Sim, we utilize the trie data structure to maintain the execution history of the application being simulated and quickly predict the cluster ID of the next region based on this information.

Figure 5.5a illustrates the usage of tries to predict the cluster ID of the next region by considering the example of a hypothetical execution sequence. *Insert:* The cluster ID of the current region is inserted into the trie. In Figure 5.5a, when the online clustering of the fifth region is finished, we insert the current cluster ID 2 for both branches corresponding to the three histories: 3, 3 $\rightarrow$ 3, and 2 $\rightarrow$ 3 $\rightarrow$ 3. *Search:* Once Insert of the current region is completed, the cluster ID of the next region is predicted by searching the trie for a matching cluster ID sequence. The search operation ends when the sequence matches one of the leaf node paths. Note that two regions having the same marker do not necessarily mean that the regions belong to the same cluster.

Figure 5.5b shows the average accuracies of the online predictor for the benchmarks of SPEC CPU2017 and NPB are 94% and 85%, respectively, ensuring the sampling accuracy and performance of Pac-Sim. The accuracy of the predictor is determined by comparing the predicted cluster ID prior to simulating the region with the actual cluster ID obtained through clustering after simulation. Figure 5.6 shows the results of the Predictor in clustering different regions identified by Pac-Sim simulating the `ft` benchmark from the NPB benchmark suite using eight threads. We observe that the majority of regions from each cluster are simulated in detail (shaded portions). This is in accordance with the learning phase of our algorithm where Pac-Sim works to establish a comprehensive understanding of the phase behavior of the application.

5.3.5 Simulation by Application Reconstruction

Previously proposed multi-threaded sampling methodologies [8, 34] rely fully on offline analysis to determine the regions that need to be simulated in detail. Pac-Sim assumes no prior knowledge about the nature of the workload that it is about to simulate. Instead, it (a) samples regions online during the simulation and (b) uses the detailed simulation results of previous regions to estimate the performance of the current fast-forwarded region by applying the four different methods described below successively until convergence is reached.

i. Use the detailed performance metrics of a region that belongs to the same cluster and has the same start marker as the current region.

ii. Use the detailed performance metrics of a region that belongs to the same cluster.

iii. Use the performance of a region with the closest BBV ($\theta = 1$) and the same number of active threads.

iv. Use the average performance of the regions that have the same number of active threads.

We extrapolate the runtime T_r of a fast-forwarded region r using the region r' by $T_r = T_{r'} \frac{insn_r}{insn_{r'}}$, where $T_{r'}$ is the runtime of the previous region r' identified above, and $insn_r$ and $insn_{r'}$ are the maximum instruction counts among all threads for the regions r and r', respectively.

Runtime Hardware Events. Pac-Sim takes into account the state of the simulated system while estimating the performance of the fast-forwarded region. As runtime hardware events can happen at any time, we do not guarantee the regions that are divided by those events to be large enough. In such cases, we estimate the performance of these regions using the closest previous region with the same hardware state, as these regions are too small to be clustered. Moreover, the impact of these regions on the overall application performance is typically negligible as the regions are too short.

5.3.6 Sampled Simulation in Parallel

Pac-Sim is primarily targeted for runtime varying scenarios using live sampling. However, for statically scheduled multi-threaded applications, Pac-Sim can support sampled simulation in parallel, similar to checkpoint-based mechanisms, to further speed up the sampled simulation. The workflow of Pac-Sim for parallel simulation is shown in Figure 5.7. Previous methods, like LiveSim [27] and LoopPoint [8], require offline analysis and store checkpoints for sampled simulation. A huge amount of storage is required for these methods, as mentioned in Section 4.2. Pac-Sim starts in emulation mode, collect-

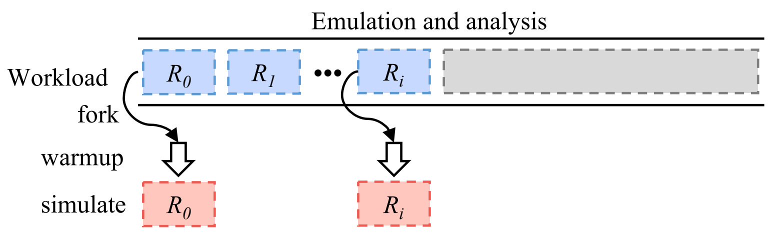


Figure 5.7: The workflow of Pac-Sim when the representative regions are simulated in parallel. Pac-Sim starts in the emulation mode, collecting feature vectors and MTR [5] warmup data online, and then predicts the simulation mode of the next region. For regions predicted for detailed mode, Pac-Sim forks new processes to perform warmup and detailed simulation.

ing feature vectors and warmup data online, and then predicts the simulation mode of the next region. For regions predicted for detailed mode, Pac-Sim forks new processes, which run in parallel, to perform warmup and detailed simulation. Pac-Sim reconstructs the performance of the entire application once the whole application is emulated and the simulation of all regions is completed.

5.3.7 Microarchitectural Warmup

One of the major challenges of sampled simulation is to choose the right warmup technique that can directly build up an accurate microarchitectural state prior to the detailed simulation of a region. Methodologies like SMARTS [2] and time-based sampling techniques [32, 33] keep functional warming enabled for the entire sampled simulation, leading to large slowdowns. We find that the statistical warmup techniques [5, 88, 91, 92] can reconstruct the accurate microarchitectural state of a simulated system online. We choose the memory time-stamp record (MTR) [5] technique to be used with Pac-Sim. MTR can rapidly collect memory reference patterns during the fast-forward mode and reconstruct the cache state before switching to detailed simulation. In this work, we limit the simulation infrastructure to explicit cache warming, as the smaller structures tend to be warmed relatively quickly.

We employ a cache warmup strategy by populating the cache structures with the N most recent unique memory accesses, where N represents the total number of cache lines being simulated. We maintain a minimum region length of 20 million instructions to mitigate the impact of smaller structures on sampling accuracy. While prior research explores warming up various microarchitectural structures [86, 87], our experiments demonstrate that focusing solely on last-level cache warmup is sufficient for high sampling accuracy. Pac-Sim can be configured with other warmup techniques, but evaluating their effectiveness is beyond the scope of this work.

5.4 Experimental Setup

In this section, we describe the experimental setup used to evaluate Pac-Sim. We begin by providing the specifics of the simulation framework, comprising the simulator used and details of the simulated architecture employed in our experiments. We then describe the different workloads that are used to evaluate the performance of our methodology, including SPEC CPU2017 [117], PARSEC [155], and NAS Parallel Benchmarks (NPB) [112] with different multi-threaded programming models, namely OpenMP [125] and OmpSs [42]. The default parameters of Pac-Sim used in our experiments are listed in Table 5.2.

Table 5.2: The default parameters of Pac-Sim used in our experiments.

Parameters	Values
Min. Region Length (instructions)	20,000,000
Max. Region Length (instructions)	50,000,000
Dimensions of Online BBV	16
Max. Depth of Trie used in Predictor	16
Clustering Threshold	0.05

5.4.1 Simulation Tools

In this work, we use a modified version of the Sniper multi-core simulator [14, 156] (version 7.4), which is updated to support loop-based and barrier-based region specifications in order to evaluate Pac-Sim. Sniper is a many-core simulator using high-level abstract models and is widely used for architectural evaluation and design space exploration. Note that our methodology does not utilize any features specific to the Sniper simulator. Therefore, porting the methodology to other simulators, such as gem5 [6] or ZSim [93], should be relatively straightforward. To demonstrate that Pac-Sim is indeed a microarchitecture-independent methodology, we experimentally evaluate it by running simulations upon two different processor configurations that mimic the performance/behavior of Intel's Gainestown, Skylake [157], and Sunnycove [158] microarchitectures using Sniper. The configuration details for each of these models are listed in Table 5.3.

Table 5.3: The configuration parameters we used for Gainestown, Skylake, and Sunnycove microarchitectures on Sniper.

Component	Gainestown Parameters	Skylake Parameters	Sunnycove Parameters
Processor	1, 8 cores	1, 8 cores	1, 8 cores
Core	2.66 GHz, 128-entry ROB	2.66 / 3.7 GHz, 224-entry ROB	3.60 GHz, 352-entry ROB
L1-I / L1-D	32 KB, 4 / 8 way, LRU	32 KB, 8 / 8 way, LRU	32 / 48 KB, 8 / 12 way, LRU
L2 cache	256 KB, 8 way, LRU	1 MB, 16 way, LRU	1.25 MB, 20 way, LRU
L3 cache	8 MB (shared), 16 way, LRU	22 MB (shared), 12 way, LRU	16 MB (shared), 16 way, LRU

To enable high-performance simulation, Pac-Sim intelligently switches among the three simulation modes supported by Sniper, namely, fast-forward mode, cache-only mode, and detailed simulation mode. The fast-forward mode is used to reach a particular point in an application during simulation without enabling the performance models. The cache-only mode performs the functional warming of the caches, whereas the detailed simulation mode is the default simulation mode that enables the timing model for performance

Note that Gainestown is the latest microarchitecture available on Sniper simulator that has been validated against hardware. We made modifications to the back-end of Sniper to support the contention model and instruction latencies for Skylake and Sunnycove architectures.

estimation. For Pac-Sim, we capitalize on the split execution and timing model of Sniper to fast-forward in the front-end of the simulator so that the simulation wall time is further minimized.

Every time Pac-Sim switches from fast-forward to detailed simulation mode; the cache state is reconstructed at the beginning of the region using the memory time-stamp record (MTR) [5] technique. We implement MTR in Sniper to collect the cache line information accessed by each *Load* and *Store* instruction during simulation, ordered in LRU fashion per set, and then inject the requests into the cache in the correct order to rebuild the appropriate cache state.

5.4.2 Benchmarks Used

To demonstrate the applicability of Pac-Sim, we experimentally evaluate the methodology using multiple benchmark suites such as (i) the SPEC CPU2017 benchmark suite [117], (ii) the NAS Parallel Benchmarks (NPB) [112] version 3.4.2, and (iii) the PARSEC [155] version 3.0 benchmark suite. Note that these are multi-threaded benchmarks that synchronize frequently and share memory.

We configure these benchmarks to use two different multi-threaded programming models, namely OpenMP [125] and OmpSs [42]. OpenMP [125] provides a set of compiler directives, library routines, and environment variables that help developers to parallelize their code. On the other hand, OmpSs [42] extends OpenMP, and it is able to dynamically manage and schedule tasks to maximize multi-threaded application performance. We set up the multi-threaded benchmarks to use *passive* thread wait policy, meaning that the threads will sleep while waiting for other threads at a synchronization point.

SPEC CPU2017 is a collection of benchmarks used for performance evaluation in computer architecture research. In our experiments, we use the *speed* version of multi-threaded SPEC CPU2017 benchmarks that are parallelized with OpenMP. The bench-

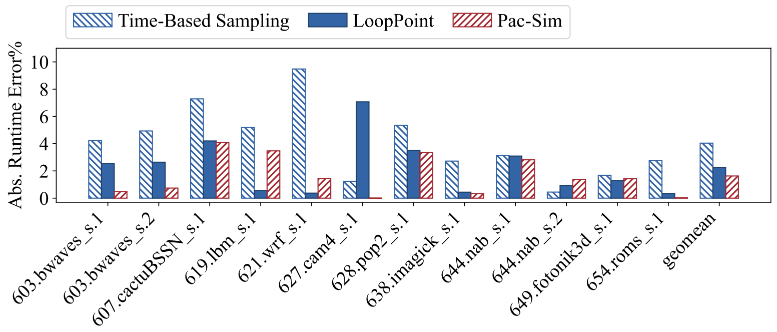
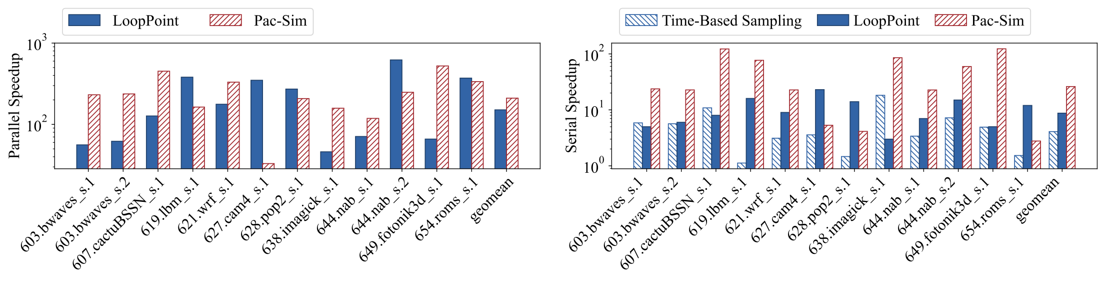


Figure 5.8: A comparison of the absolute runtime prediction error using different methodologies, namely, Time-Based Sampling, LoopPoint, and Pac-Sim for 8-threaded SPEC CPU2017 benchmarks using train inputs. On average, Pac-Sim achieves better accuracy compared to Time-Based Sampling and LoopPoint.

marks are compiled using GCC 6.4.0 and GFortran with the `-O3` compiler flag for x86-64 architecture. We configure these benchmarks to run with eight threads and evaluate them using the *train* input set. NAS Parallel Benchmarks (NPB) [112] is another set of benchmarks widely used to evaluate the performance of highly parallel systems in computer architecture. The reference implementations of these benchmarks are available in the two most commonly used programming models, i.e., MPI and OpenMP. In our experiments, we use the OpenMP-based implementation with input *class A* and generate the binaries using icc compiler (with `-O2` flag) as part of the Intel oneAPI (version 2022.0.2) toolkit. We also present experimental evaluations of Pac-Sim using PARSEC, which is another standard benchmark suite consisting of computationally intensive applications designed to facilitate the study of multi-core systems with shared memory. PARSEC implementations are available in both OpenMP and OmpSs [159] versions. In our experiments, we use both these versions with the *simlarge* input set.

5.5 Evaluation

In this section, we first present a comprehensive evaluation of Pac-Sim, comparing its efficacy with the current state-of-the-art. Additionally, we provide experimental evi-



(a) Parallel Speedup **(b)** Serial Speedup

Figure 5.9: The parallel and serial speedups achieved using Pac-Sim for 8-threaded SPEC CPU2017 benchmarks using train inputs. For speedup calculations, the simulation walltime corresponding to Pac-Sim includes both online analysis and simulation time, whereas, for LoopPoint, we consider only the checkpoint simulation time, excluding the time required for offline profiling and checkpoint generation. Pac-Sim outperforms both Time-Based Sampling and LoopPoint in terms of speedup achieved. Note that Time-Based Sampling techniques are not suitable for sampled simulation in parallel.

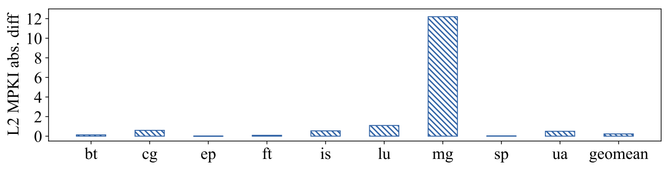
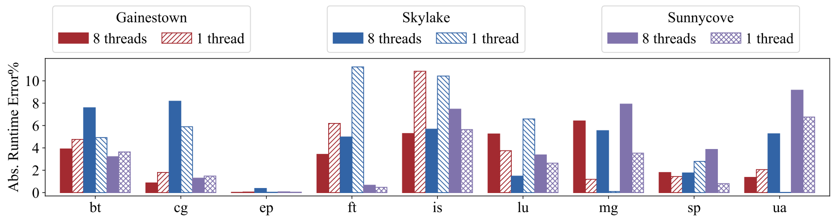


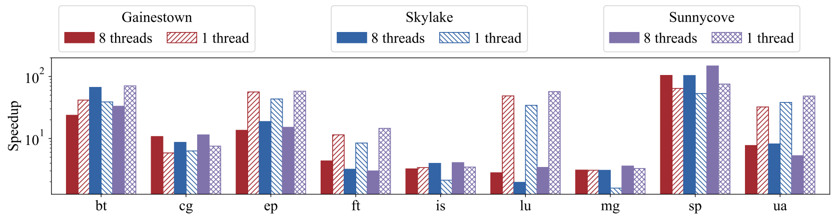
Figure 5.10: The absolute differences in predicting L2 cache misses per kilo instructions (MPKI) using Pac-Sim as compared to the full detailed simulation. In this experiment, we use the NPB benchmarks with class A inputs running eight threads. The geometric mean of the absolute differences in predicting L2 MPKI is 0.23.

dence showing that Pac-Sim is indeed a hardware-independent methodology. Finally, we present case studies that demonstrate the applicability and effectiveness of Pac-Sim in estimating workload performance in dynamic, multi-threaded hardware and software environments. Note that, throughout this chapter, the term runtime refers to the simulated runtime of the application, whereas the term wall-time refers to the actual time taken by the simulator to finish the run.

Evaluation metrics. In order to evaluate the effectiveness of any simulation methodology, it is crucial to quantitatively measure its performance in terms of two critical metrics: *accuracy* and *speedup*. In our experiments, we define these metrics in the fol-



(a) Accuracy



(b) Serial Speedup

Figure 5.11: The accuracy and serial speedup achieved for Pac-Sim methodology when simulated using three different microarchitectures, namely, Gainestown, Skylake, and Sunnycove, for NPB benchmarks with class A inputs running eight threads and one thread.

lowing manner:

Accuracy: We assess the accuracy of our proposed methodology by comparing the simulation runtime obtained from the full simulation and the sampled simulation in terms of absolute runtime prediction error Δ_{time}, which is defined as

$$\Delta_{time} = \frac{|T_{full} - T_{sample}|}{T_{full}},$$

where T_{full} represents the simulation runtime obtained from the full run, and T_{sample} represents the simulation runtime extrapolated from the sampled simulation. It is important to note that in our evaluation, we use the runtime (execution time as inferred from simulation) of the application as the performance metric to measure the accuracy of sampling. This is because time-per-program is the gold-standard performance measure, and IPC is not a valid performance metric for multi-threaded applications [31].

Speedup: In our experiments, we calculate the speedup by taking the ratio of the wall-clock time for the full simulation to that of the sampled simulation and the average speedup by computing the geometric mean of the speedups across all benchmarks. Serial speedup is defined as the speedup achieved when all representative regions are simulated sequentially, while parallel speedup is obtained when the representative regions are simulated in parallel, assuming infinite resources.

5.5.1 Comparison with the State-of-the-Art

In this section, we evaluate the performance of Pac-Sim in comparison to the state-of-the-art profile-driven sampled simulation methodology, LoopPoint [8]. While several other profile-driven methodologies exist, LoopPoint provides the benefit of being applicable across a variety of application and synchronization types. It has also been shown to outperform other multithreaded sampled simulation methodologies (such as BarrierPoint) in terms of speedup and accuracy, thus serving as a strong baseline for our evaluations. We also compare Pac-Sim with Time-Based Sampling [32, 33] techniques, which repeatedly alternate between detailed simulation and fast-forwarding of regions. For a fair comparison, we adopt Pac-Sim's approach of injecting the warmup state at the beginning of each detailed simulation region. We now report the results of our simulation experiments evaluating and comparing the performance of these two methodologies using the SPEC CPU2017 benchmarks.

Accuracy. Figure 5.8 shows a comparison of absolute runtime prediction errors for Pac-Sim, Time-Based Sampling, and LoopPoint obtained for the 8-threaded SPEC CPU2017 benchmarks using train inputs. Our analysis reveals that, in most cases, Pac-Sim performs comparably with LoopPoint in predicting the runtime of the applications, with the individual errors differing by no more than 2 to 3%. The relatively higher errors for some applications, such as `619.lbm_s.1`, are because Pac-Sim relies on online extrapolation to estimate application performance using the limited profile data that is available from regions that have already been simulated. Whereas methodologies like LoopPoint rely on offline profiling and, therefore, utilize the information about the whole application. We also evaluate the accuracy of Pac-Sim to determine the L2 cache misses per kilo instructions (MPKI), as shown in Figure 5.10. The final results show that the average absolute difference of MPKI for all benchmarks evaluated is 0.23, demonstrating Pac-Sim accurately extrapolates microarchitectural metrics of the whole application from the selected samples.

Speedup. Figure 5.9 shows the speedup comparison of Pac-Sim, Time-Based Sampling, and LoopPoint for the SPEC CPU2017 benchmarks using train inputs running eight threads. Figure 5.9a shows the parallel speedup for which Pac-Sim outperforms LoopPoint in most cases (7 out of 12 benchmarks). The primary reason for this is that Pac-Sim uses smaller regions as compared to LoopPoint. Although Pac-Sim requires emulation of the entire application, the online analysis overhead is minimized, and therefore, the average parallel speedup for SPEC CPU2017 benchmarks (train inputs) using Pac-Sim is 210.3×, which is larger than that obtained for LoopPoint (150.97×).

Figure 5.9b shows the serial speedup, and we observe Pac-Sim outperforms LoopPoint in most cases, attaining a maximum serial speedup of 123.32×. Specifically, Pac-Sim outperforms LoopPoint by 1.8× and 1.4× for the serial and parallel speedup, respectively. While the online analysis can introduce some runtime overheads, the performance advantages of Pac-Sim seem to outweigh these overheads in most cases. We observe that the performance of Pac-Sim surpasses that of Time-Based Sampling across all evaluated benchmarks. This is because Pac-Sim selects regions for detailed simulation through online learning, in contrast to Time-Based Sampling. Furthermore, the use of program constructs like barriers and loops in Pac-Sim for region selection enables identifying repetitive program behavior in multi-threaded workloads. However, there are some cases where LoopPoint performs better than Pac-Sim, such as for `627.cam4_s.1` and

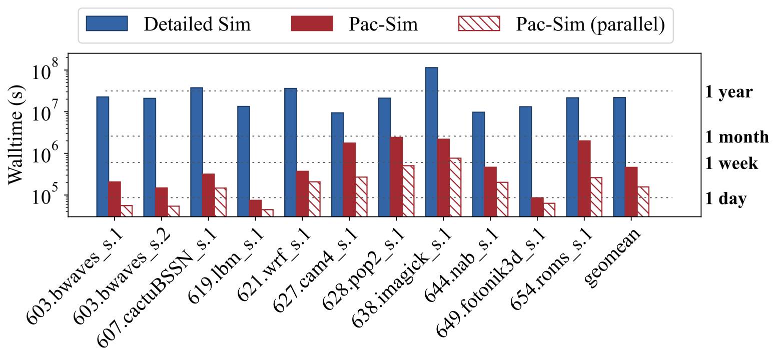


Figure 5.12: A comparison of the estimated walltime for fully detailed simulation and sampled simulation using the serial and parallel versions of Pac-Sim for 8-threaded SPEC CPU2017 benchmarks using ref inputs. The estimated walltime includes the time required for online analysis, warmup, and simulation.

`628.pop2_s.1` benchmarks in Figure 5.9b. This is mainly because Pac-Sim uses a small clustering threshold (0.05) for the online clustering in order to maintain higher accuracy, compared with Time-Based sampling and LoopPoint.

Efficacy in Evaluating Realistic Workloads. The full detailed simulation of SPEC CPU2017 benchmarks with reference inputs takes an extremely long time – about a year on average using multi-core simulators like Sniper. Instead, we estimate their simulation walltime by considering the instruction count of the benchmark using reference inputs along with the average simulation rate of the benchmark using train inputs. The walltime of Pac-Sim includes the time required for online analysis and emulation of the entire workload along with the time for detailed simulation of the representative regions. Figure 5.12 shows that Pac-Sim takes less than a week, on average, to run the entire application sequentially, while the parallel version of Pac-Sim takes about 1.8 days on average. In experiments where the microarchitecture structures like cache size are adjusted or when the application itself undergoes instruction-level modifications, it is necessary to regenerate the checkpoints. In such cases, Pac-Sim is more appropriate as LoopPoint takes 6.2 days on average (shown in Figure 5.2) to complete its preprocessing before simulation.

Microarchitecture-agnostic sampling. In addition to achieving high accuracy and speedups, Pac-Sim also provides the advantage of being a microarchitecture-independent methodology. We experimentally demonstrate this by evaluating our methodology with two different processor configurations, namely the Gainestown and Skylake microarchitectures, for the NPB benchmarks that run using one thread and eight threads. The accuracy and speedup numbers obtained in our experiments are plotted in Figure 5.11a and Figure 5.11b, respectively. From Figure 5.11a, we can observe that the absolute runtime errors estimated by Pac-Sim for all NPB applications are quite low (all under 8%) and are similar for both these processor configurations (differing by 5% at most). Moreover, the speedups obtained for both configurations are similar for most benchmarks, as observed in Figure 5.11b. Hence, the choice of a target microarchitecture for evaluation does not affect the efficacy of Pac-Sim.

Wall-time Distribution. We show the time spent by Pac-Sim in different stages of sampled simulation. Figure 5.13 shows the average time spent in the online analysis stage for NPB (class A inputs) and SPEC CPU2017 (train inputs) benchmarks is 7.88% and 11.20%, respectively. This is the result of the optimizations described in Section 4.3, which are applied to the analysis part. Moreover, Pac-Sim spends 8.25% and 16.00% of the execution time on warmup for NPB and SPEC CPU2017 benchmarks, respectively. This is because Pac-Sim needs to reconstruct the memory access patterns at the beginning of the detailed simulation of a region. Note that Pac-Sim reduces the time spent in both profiling and analysis of the benchmarks significantly as compared to prior profile-driven methodologies for sampled simulation.

Memory Overhead. The memory overhead of Pac-Sim is minimal (no disk access) and allocated once per simulation. For the online analysis, Pac-Sim needs to record the BBV information of the previously simulated regions. The space complexity of the online analysis is $\mathcal{O}\left(\frac{insn_{full}}{region\_length}\right)$, where $insn_{full}$ is the total dynamic instructions

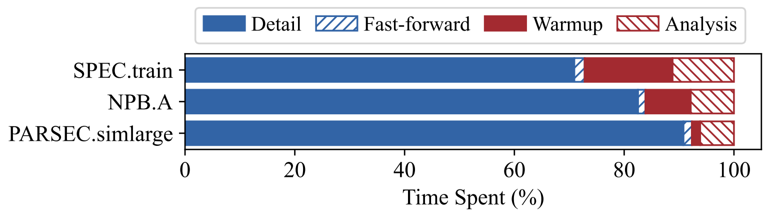


Figure 5.13: The graph shows the percentage of time that Pac-Sim spends at each phase during the sampled simulation of each benchmark suite (average across all benchmarks). The Analysis component includes online marker detection, region profiling, clustering, and prediction.

executed by the application and $region\_length$ is the average length (size) of the regions identified. The MTR warmup injection technique keeps counters for each cache line accessed (num_cache_lines), and the space complexity is $\mathcal{O}(num\_cache\_lines)$.

5.5.2 Case Studies

We showcase the versatility of Pac-Sim through several compelling case studies. Firstly, we demonstrate that our methodology remains agnostic to dynamic thread scheduling decisions made during runtime, highlighting its robustness and adaptability. Next, we provide examples of how Pac-Sim operates seamlessly in the presence of various runtime hardware events, further cementing its reliability. Finally, we exhibit the applicability of the proposed methodology in hardware-software co-design studies, showcasing its potential to facilitate more efficient and effective design processes.

5.5.2.1 Dynamically Scheduled Software

The advent of multi-core and many-core architectures has necessitated the efficient parallel execution of dynamically scheduled multi-threaded applications to maximize system performance. However, the non-determinism resulting from the execution of such applications on multi-core platforms often leads to notable performance variability across multiple runs. This variability can be attributed to software-level factors such as dynamic job scheduling by the operating system, thread migration between cores, load balancing optimizations, and contention for shared resources at runtime.

Table 5.4: Table shows the IPC of `freqmine` benchmark from the PARSEC benchmark suite using the *simlarge* input for threads 0 through 7. Pac-Sim shows the details of dynamically scheduled software whose IPC and thread mapping differ across two runs.

Thread ID	0	1	2	3	4	5	6	7	Aggr.
IPC$_{run1}$	0.15	0.09	1.75	**0.43**	**0.07**	0.07	0.10	0.09	2.75
IPC$_{run2}$	0.15	0.09	1.76	**0.07**	**0.44**	0.07	0.09	0.10	2.76

Table 5.4 illustrates the thread-level differences in terms of instructions per cycle (IPC) for two different runs of the OpenMP-parallelized `freqmine` application from the PARSEC benchmark suite. There are variations in the per-thread IPCs between the two runs, particularly for thread IDs 3 and 4. To investigate the impact of these variations on conventionally used sampling techniques, we conducted two independent profiling runs of `freqmine` using LoopPoint. The experimental results showed significant variability, with 14% of the regions clustered differently between runs. This presents a challenge for sampled simulation, which relies on profiling data from a prior execution to guide simulation in subsequent runs. Dynamically scheduled applications, by nature, have profiling data that fluctuates across executions. To address this challenge caused by runtime variability, Pac-Sim performs online profiling and simulation within the same run. This allows Pac-Sim to capture any performance variations that might occur during execution.

To demonstrate the effectiveness of Pac-Sim in this regard, we now present an experimental study of dynamically scheduled multi-threaded versions of PARSEC with simlarge inputs and NPB with class A inputs. While the per-thread behavior varies for dynamically scheduled applications, the global execution time and global IPC remain consistent across multiple runs. In Table 5.4, we observe that while there are some variations in per-thread behavior, the aggregate IPCs across the two runs remain nearly unchanged. Figure 5.14 demonstrates the average runtime prediction errors of Pac-Sim simulating

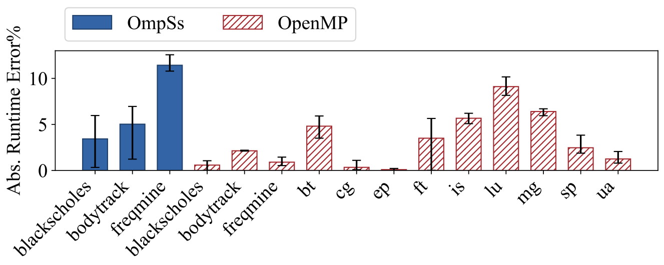


Figure 5.14: Figure shows the average error rates (from five different runs) and error bars in predicting the runtime of dynamically scheduled benchmarks. We use PARSEC benchmarks with the *simlarge* input using OmpSs and OpenMP, and NPB benchmarks with *class A* inputs using OpenMP runtime.

dynamically scheduled multi-threaded applications. We run the benchmarks multiple times in full detailed mode and using Pac-Sim. The errors are calculated by comparing the runtime obtained using Pac-Sim with the average runtime obtained from the full detailed simulations. The results show that Pac-Sim achieves a very low error in predicting the runtime of dynamically scheduled software (3.81% on average). The benchmark, `freqmine`, which shows the largest IPC variation (without spinloops) maintains an average error of 11.43%. Moreover, Pac-Sim demonstrates speedups of up to 43.96× (6.29× on average) for all dynamically scheduled benchmarks.

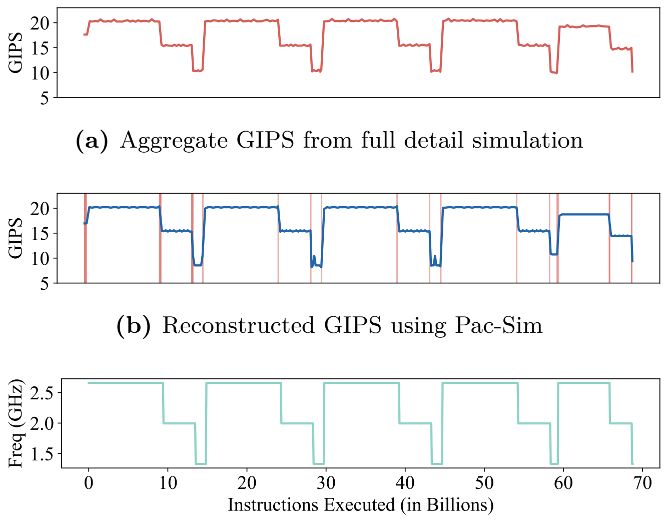
5.5.2.2 Dynamic Hardware Events

Dynamic event-based hardware optimizations help improve performance gains and energy efficiency in modern architectures. DVFS [35, 36, 37] is one of the most widely employed dynamic hardware event-based optimization techniques. It monitors core frequencies and load variations in order to match the system power consumption with the required level of performance by triggering voltage and frequency optimizations at runtime. These optimizations may lead to a diverse range of dynamic hardware states (i.e., core frequency, power configurations) over a given run, consequently resulting in a significant degree of performance variability for a given workload across different executions.

Pac-Sim deals with this performance variability by monitoring the simulated hardware events during execution. While prior sampled simulation methodologies can be modified to support dynamic hardware events triggered only at region boundaries (to maintain consistent hardware state within a region), Pac-Sim allows hardware events at any point during the application execution. Each time an event occurs, Pac-Sim triggers a new region to ensure hardware state consistency within that region. The predictor then speculates the cluster ID of the next region and checks the execution history to determine whether similar regions (i.e., regions with the same cluster ID and hardware state) were previously encountered. If a match is found, the region is fast-forwarded; otherwise, a detailed simulation is triggered.

We now present an experimental study demonstrating the effectiveness of Pac-Sim in handling the variability caused by dynamic hardware events by specifically considering the case of DVFS-optimized workloads. In our experiments, we evaluate the performance of the benchmarks by comparing the results of Pac-Sim with the baseline while changing the frequency at predetermined intervals; however, just like in actual DVFS-optimized executions, the information on the frequency changes is not available to the simulator a priori. In order to evaluate the performance of Pac-Sim, we consider a DVFS scenario in which the processor frequency { switches among a fixed range of values, i.e., { ∈ {1.33 GHz, 2.00 GHz, 2.66 GHz} as shown in Figure 5.15c.

For this scenario, we measure the aggregate giga/billion instructions per second (GIPS) values obtained from both the full detailed simulation and Pac-Sim over the entire execution. The findings of our experiment are presented in Figure 5.15. We observe that the GIPS values obtained from both the full simulation (Figure 5.15a) and Pac-Sim (Figure 5.15b) exhibit a great deal of similarity, indicating Pac-Sim's effectiveness in estimating the performance of a dynamically optimized workload with a high level of accuracy. Furthermore, our findings reveal that Pac-Sim simulates only a small fraction



(a) Aggregate GIPS from full detail simulation

(b) Reconstructed GIPS using Pac-Sim

(c) CPU Frequency

Figure 5.15: The aggregate giga (billion) instructions per second (GIPS) of the full run (a), reconstructed GIPS using Pac-Sim (b), and the varying CPU frequency for all CPUs (c) `644.nab_s.1` benchmark with train inputs running 8 threads. The shaded regions in (b) represent the regions simulated in detail. The figures share the same x-axis.

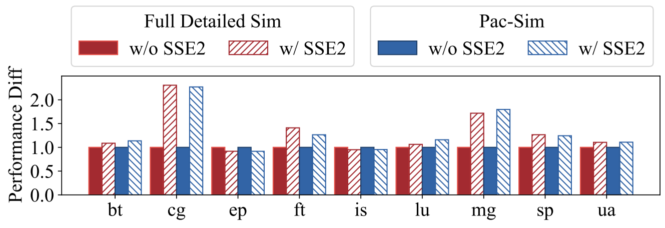


Figure 5.16: The figure shows the absolute difference in performance (in terms of runtime) for NPB benchmarks using class A inputs and 8 threads with (w/) and without (w/o) SSE2 simulated in detailed mode and with Pac-Sim.

of the entire application in detail (depicted by shaded regions in Figure 5.15b). Notably, most of the detailed simulation occurs either at points of change in the phase behavior of the application or hardware states. This demonstrates that Pac-Sim can use this information to identify a minimal representative subset for applications using online analysis.

5.5.2.3 Hardware-Software Co-design

Hardware-software co-design is an emerging field of study that optimizes the system performance by concurrently designing the compiler and hardware components of a system to exploit the synergy between the two. Prior works [160, 161, 162] have investigated several directions in this context. To identify the most effective strategies, hardware-software co-design research relies on fast and accurate architectural simulation methodologies to explore the design space efficiently. However, among existing methodologies, the profile-driven methodologies [20, 34] incur significant profiling and preprocessing costs, as shown in Figure 5.2, whereas the statistical sampling methodologies [2, 146] (which do not rely on preprocessing) have low speedups.

Pac-Sim addresses these issues by sampling and analyzing the regions online. Thus, it incurs no additional profiling cost if new compilers are used or new applications are generated, enabling fast and efficient exploration of hardware-software co-design space. To demonstrate the effectiveness of Pac-Sim in this regard, we now present a performance evaluation study of the NPB benchmarks under different compiler optimization techniques. We study the impact of SIMD (Single Instruction, Multiple Data) optimizations on the generated binaries using both Pac-Sim and full detailed simulations. SIMD-enabled processors are equipped with special-purpose registers that can simultaneously load multiple machine words and perform operations on them in parallel in order to improve processor performance. For instance, the Streaming SIMD Extensions 2 (SSE2) instruction set uses 128-bit XMM registers to process packed data elements at once.

In our experiments, we measure the performance improvement (in terms of runtime) obtained by enabling SSE2 and compare it against the baseline. The results of our simulations are presented in Figure 5.16. We observe that the average difference in the performance improvements obtained from full detailed mode and Pac-Sim is 3.65%.

Specifically, Pac-Sim reveals the performance effects of SIMD instructions. For example, some benchmarks achieve a significant speedup over the baseline as these applications meet the `icc` vectorization criteria [163]. `ft` calculates a 3D fast Fourier transform, and its innermost loop consists of multiply-add statements with contiguous memory accesses and no data dependency. On the other hand, `is`, which uses the quick sort algorithm, is hard to vectorize. The SIMD overheads resulting from register transfer costs exacerbate the overall application performance.

5.6 Related Work

We have discussed the most relevant previous works in Section 4.2. Sampled simulation has been an active research area for several decades, and several techniques were proposed [2, 8, 20, 27, 30, 32, 33, 34, 43, 78, 80, 164] in this direction for different workload classes primarily for the reduction of simulation time and resources. Analytical modeling is yet another solution to evaluate a complex workload quickly. Prior works proposed analytical models to derive the performance of processors [84, 85], cache miss rates [86], branch miss rates [87], DVFS performance [165], etc. However, analytical performance modeling can be limited in supporting new designs, requiring new models for each.

5.7 Conclusion

In this work, we propose a novel methodology, Pac-Sim, that allows for the sampled simulation of dynamic software that responds to workload and run-time execution conditions. Pac-Sim is the first, to the best of our knowledge, to propose a sampling solution that simulates these dynamic conditions in both a fast (up to 523.5$\times$ speedup, 210.3$\times$ on average) and accurate way (average errors of 1.63% and 3.81% for statically and dynamically scheduled benchmarks, respectively).

Chapter 6

XPU-Point: Simulator-Agnostic Sample Selection Methodology for Heterogeneous CPU-GPU Applications

Reality is merely an illusion, albeit a very persistent one.

— Albert Einstein

The end of Dennard scaling has driven chip design towards multi-core and heterogeneous architectures. As multi-core architectures are reaching their scaling limits, the focus has been pivoted to heterogeneous architectures. However, performance evaluation of heterogeneous systems using full-program simulations is prohibitively slow. We introduce XPU-Point, a novel methodology to identify representative regions within heterogeneous CPU-GPU workloads, enabling fast and accurate performance evaluation through sampled simulations.

6.1 Introduction

Computation exists everywhere in this era, spanning from large-scale systems to low-power devices and mobile CPUs. There has been a profound increase in the demand

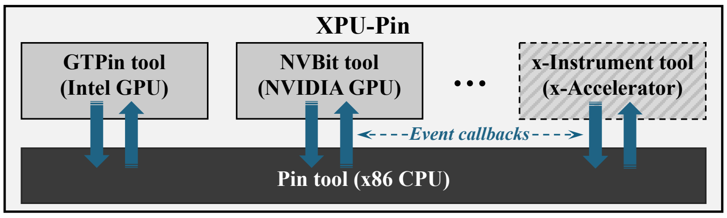


Figure 6.1: A high-level schematic of XPU-Pin. The x86 CPU instrumentation tool Pin interacts with GPU instrumentation tools (like GTPin and NVBit) for event-based callbacks. Integration with similar tools for other hardware components (x=TPUs, NPUs, accelerators, etc.) is feasible. The simulation phase (not shown), which is performed using a variety of tools, is handled separately.

for high-performance computing (HPC) resources in recent years [166]. However, the limitations of multi-core architectures to scale due to the associated power and thermal constraints (power wall) restricts their ability to deliver significant performance improvements [167, 168]. This has resulted in a shift toward domain-specific architectures and accelerators like GPUs [169], TPUs [170], and FPGAs [171]. Embracing heterogeneity in architectures is one way forward for continued performance improvements [12] to meet these growing computational demands. The use of a combination of architectures is needed to continue to scale the performance of future systems [172], to achieve accelerator-level parallelism [173].

The prevalence of CPU-GPU architectures in heterogeneous computing arises from their ability to address the evolving demands of modern workloads, coupled with their well-established programming models and their ability to exploit parallelism at a massive scale. GPUs have emerged as the most widely used general-purpose accelerators in modern data centers [53] and supercomputers [54] that accelerate massively parallel big data analysis [55, 56] and machine learning [57, 58] workloads. While previous works have investigated characterizing workloads that consist of CPU components [2, 20, 30, 32, 34] and GPU [45, 46, 59, 60] components independently, as well as their comparative analyses [61], hybrid solutions that support analysis and workload reduction, like sampling, for multiple types of heterogeneous workloads, from CPUs, GPUs, and even custom

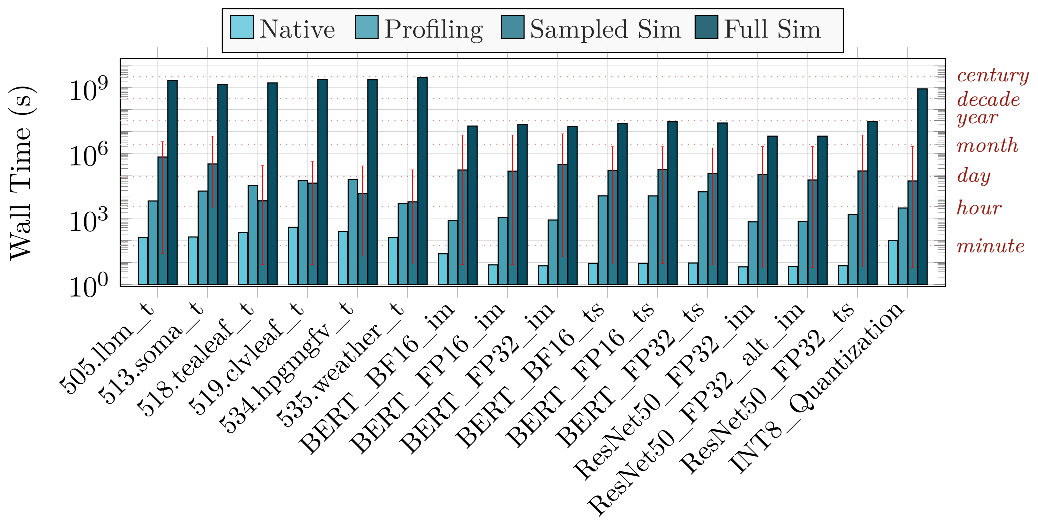


Figure 6.2: The wall time (in seconds) for evaluating realistic heterogeneous CPU-GPU workloads such as SPEChpc 2021 benchmarks (tiny set) using ref inputs and PyTorch Inference runs. Benchmarks were evaluated in (a) native run, (b) profiling using XPU-Point, (c) parallel simulation of the representative regions identified using XPU-Point (mean wall time with error bars indicating the shortest- and longest-running regions), and (d) full-detailed simulation. The experiments are conducted on machines that use Intel Sapphire Rapids CPU and Intel Ponte Vecchio GPU. The simulation wall times are estimated using the simulation rate of gem5 [6] and Accel-Sim [7]. im=Imperative, ts=TorchScript.

hardware accelerators (like FPGAs), have not yet been identified. Given the importance of these workloads, from HPC systems to data center use, simulation of heterogeneous workloads is key to understanding the interactions between compute components and how these interactions can affect overall runtime performance.

The growing significance of heterogeneous computing architectures necessitates a refined approach to performance analysis. While GPUs have become indispensable for accelerating workloads like AI training and inference, the CPU plays a critical role in scheduling tasks and managing memory. A performance bottleneck within the CPU can have a cascading effect, given Amdahl's law [174], impacting overall system performance. Prior works [175] discuss the shortcomings of traditional GPU-centric analysis methods that overlook the role of the CPU in data movement and task management. Sampled simulation techniques, while prevalent for independent CPU and GPU performance analysis, suffer limitations when applied to tightly coupled CPU-GPU systems. Such simulations or performance analyses often neglect the effects of inter-core communication and cache coherency, that significantly impact the microarchitectural state. Additionally, they may not accurately capture synchronization behavior, leading to unrealistic execution order and resource usage.

Existing instrumentation and analysis tools are insufficient to capture the interactions between CPU and GPU in heterogeneous applications. While there are instrumentation and analysis frameworks for CPUs, such as Pin [116] or DynamoRIO [176, 177] for x86 programs, and for GPUs, such as GTPin [81] for Intel GPU programs and NVBit [178] for NVIDIA GPU programs, there is no framework for co-analysis of CPU and GPU code. In this chapter, we introduce *XPU-Pin*, a novel framework designed to bridge this gap by enabling simultaneous analysis of both CPU (x86) and GPU (Intel, NVIDIA) code. XPU-Pin has a Pin-based driver that loads the GPU tool library (GTPin or NVBit) explicitly and triggers it, as shown in Figure 6.1. CPU and GPU analyses can thus be integrated within the same environment, simplifying development and allowing for a unified and more accurate analysis. Additionally, the GPU tool can trigger functions registered by the driver on certain GPU events, such as the start or end of a GPU kernel. The CPU and GPU tools can thus coordinate their analysis around GPU events.

Evaluating the performance of large workloads on heterogeneous systems presents significant challenges due to long simulation times, which can take several months or years, as illustrated in Figure 6.2. Training large language models (LLMs) with multi-billion parameters, like GPT-4 [179], LLAMA2 [180], or Gemini [181], can take several months, while the inference runs may take several seconds even on powerful hardware [182, 183]. Simulation serves as a powerful tool for architects to explore potential hardware improvements that suit certain workload types. However, simulating such workloads in their entirety can be prohibitively long. Workload sampling stands as a popular technique for CPUs [2, 8, 20, 32, 33] and GPUs [16, 45, 46, 59], presenting a compelling solution by selecting a representative subset of the workload for detailed simulation. This approach delivers substantial speedups while maintaining accurate performance measurements. However, there are currently no established sampling solutions that apply to heterogeneous workloads. We build on XPU-Pin to propose *XPU-Point*, a unified sampling solution for heterogeneous workloads that can accurately build a representative sample for the fast and accurate performance analysis of the workloads. Through XPU-Point, we propose a comprehensive methodology across a broad spectrum of real-world workloads, from scientific simulations to artificial intelligence. This enables computer architects and performance researchers to quickly estimate the performance of long-running, heterogeneous workloads using sampled simulation on existing simulators [97, 135] which was not possible before.

The accuracy of the XPU-Point methodology is assessed (sample validation) based on sampling errors – the difference between the full workload performance and the performance extrapolated from the samples. Traditionally, sample validation is performed based on a detailed, and slow, timing simulation platform. We have identified two issues with simulation-based sample validation (i) it assumes that an accurate simulator for the target system is available which is not the case in the early stages of system design and (ii) it requires simulation of the entire test program to get the full workload performance which can be impractically slow as illustrated in Figure 6.2. We instead separate sample validation from simulation and perform the validation on real hardware with *XPU-Timer* (Figure 6.3). Using XPU-Timer, sample validation can be performed at near-native speed, whereas simulation-based validation can be significantly slower.

The high-level overview of the entire framework is shown in Figure 6.3. The focus of XPU-Point methodology is on selecting samples for simulation and validating those

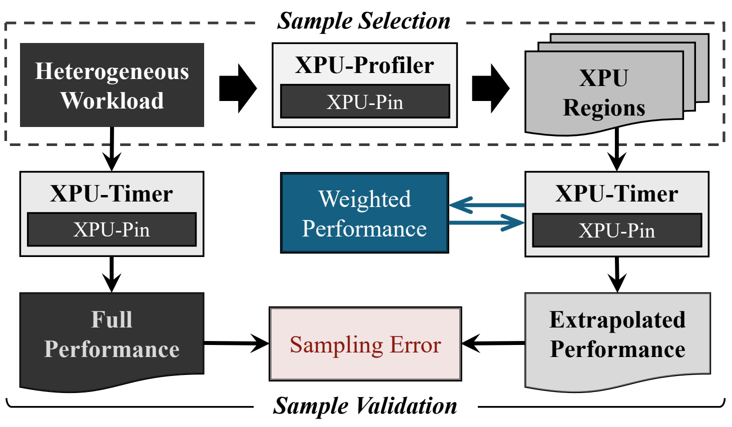


Figure 6.3: The end-to-end workflow of the XPU-Point methodology to sample heterogeneous workloads. XPU-Point uses XPU-Profiler to capture execution profiles of a heterogeneous workload. Once the representative regions (samples) are identified for the workload, their performance, as estimated by XPU-Timer (or a heterogeneous simulator), is extrapolated and compared with that of the full workload to validate the sample.

samples in a simulator-independent manner. The samples can be used to drive simulation using the platform of choice. Leveraging XPU-Point samples for simulation is left for future work.

In this work, we make the following contributions:

i. We propose XPU-Point, a methodology that goes beyond prior workload sampling techniques to be the first to allow for the support of heterogeneous applications. This enables researchers to conduct a unified and accurate performance evaluation of large-scale applications through sampled simulation.

ii. We introduce XPU-Pin, an instrumentation framework that we developed in this work to evaluate heterogeneous CPU-GPU applications. XPU-Point is built upon XPU-Pin, and supports both Intel- and NVIDIA-based CPU-GPU workloads.

iii. We experimentally assess the efficacy of XPU-Point in terms of sampling accuracy and potential simulation speedup of CPU-GPU workloads on various hardware platforms using our XPU-Timer tool instead of using a simulator.We have open-sourced the XPU-Pin framework and the XPU-Point project on GitHub [184].

iv. We extensively evaluate XPU-Point across several heterogeneous workloads, including industry-standard benchmarks such as SPECaccel 2023, SPEChpc 2021, AutoDock, GROMACS, and PyTorch inference demonstrating high accuracy (absolute sampling errors less than 5%).

The rest of the chapter is organized as follows. In Section 6.2 and Section 6.3, we discuss the background and challenges involved in the performance evaluation of heterogeneous workloads on modern architectures. Section 6.4 presents the XPU-Point methodology in detail. We then describe the experimental infrastructure in Section 6.5, followed by an extensive evaluation of XPU-Point in Section 6.6 along with case studies to demonstrate the applicability of the proposed methodology. Finally, we present the related work in Section 6.7 and conclude the chapter in Section 6.8.

6.2 XPU-Pin Framework

In this section, we explore prominent solutions for binary instrumentation, along with insights into programming models designed for heterogeneous workloads. We also delve into the development of XPU-Pin, a tool we specifically built to facilitate the co-analysis of CPU-GPU heterogeneous workloads.

6.2.1 Instrumentation and Analysis Tools

6.2.1.1 Intel Pin

Pin [116] is an x86 binary instrumentation framework that allows users to write Pin tools, which are C/C++ programs specifying analysis to be done at certain points (for example, every instruction, basic block, etc.) in program execution. Pin works in two modes, namely JIT mode and Probe mode. JIT mode works by loading an input x86 binary in memory and translating (just-in-time) its x86 code into x86 code to another region of memory called the code cache. The translation overhead in the JIT mode can result in a 40% slowdown [185] in performance even without instrumentation taking place. The slowdown and performance perturbation can be exacerbated further by the additional analysis routines. In contrast to JIT mode, probe mode works by loading an input x86 binary in memory and patching the code at certain probe points as specified by the Pin tool. Probe mode does not translate the code, but instead redirects the code inline to analysis routines. In general, Probe mode demonstrates lower overheads at the cost of programmer effort.

6.2.1.2 Intel GTPin

Intel's GPU instrumentation framework, GTPin [81], works by inserting analysis code into the GPU program via GTPintools as shared libraries. The Intel Graphics compiler generates code for the specific Intel GPUs at run time. GTPin then dynamically adds extra code specified by a GTPintool into the generated code. This modified code is then offloaded to the GPU and runs there. Any results created by the extra GTPintool code are stored in a memory buffer, and that buffer is processed on the CPU at various synchronization points.

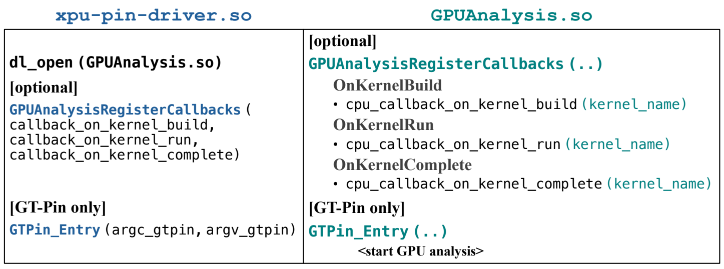
6.2.1.3 NVIDIA NVBit

NVBit [178] is a dynamic binary instrumentation framework for NVIDIA GPUs that works on the Linux operating system. It provides a high-level Application Programmer's Interface (API) for writing instrumentation tools as Linux-shared libraries. A tool library is injected in a GPU application using the LD_PRELOAD [186] feature in Linux. NVBit tools can inspect and modify the NVIDIA GPU assembly code (*SASS*) of GPU applications without requiring recompilation.

6.2.1.4 Implementation of XPU-Pin Framework

Existing tools mainly focus on either CPU or GPU components of an application due to the limitations of traditional instrumentation tools. Intel Pin [116] and DynamoRIO [176, 177] are used to analyze CPU applications, while GTPin [81] and NVBit [178] are used for GPU kernel analysis. In contrast, our newly designed framework, XPU-Pin, allows users to analyze and instrument heterogeneous CPU-GPU workloads with a single tool. XPU-Pin starts as an x86 analysis tool based on Pin (either JIT or Probe mode) and then invokes the GPU analysis shared library. The straightforward approach of linking in a GPU analysis library is not an option, as Pin requires all the libraries a Pin tool uses to be built with a special Pin runtime provided. Modifying GPU analysis tools to use special Pin runtime can be restrictive and not practical with a large number of legacy GPU analysis tools. For Intel GPUs, an alternative to linking in a GTPin tool library is to explicitly load it at runtime from the driver Pin tool. However, this requires using `dlopen()` from the the application's runtime instead of Pin runtime. For NVIDIA GPUs, the NVBit analysis framework utilizes the LD_PRELOAD feature to inject itself into the application. This mechanism remains effective when combined with x86 Pin. CPU and GPU analyses can thus be integrated within the same environment, simplifying development and allowing for a unified and more accurate analysis.

Legacy GPU analysis tools can thus be executed unmodified with x86 CPU analysis using Pin (either using `dlopen` or LD_PRELOAD). For coordinated CPU and GPU analysis, GPU analysis tools can implement an optional callback handler registration. When the Pin driver explicitly loads the GPU analysis library, it calls this registered handler. The GPU tool then uses this handler to obtain and store pointers to Pin driver functions, which it can later invoke in response to specific GPU events (like kernel start and end). This mechanism enables CPU and GPU tools to synchronize their analysis based on these GPU events. Figure 6.4 shows the control flow for an XPU-Pin tool



```
xpu-pin-driver.so                              GPUAnalysis.so

dl_open (GPUAnalysis.so)                       [optional]
[optional]                                     GPUAnalysisRegisterCallbacks (..)
GPUAnalysisRegisterCallbacks (                     OnKernelBuild
callback_on_kernel_build,                          • cpu_callback_on_kernel_build (kernel_name)
callback_on_kernel_run,                            OnKernelRun
callback_on_kernel_complete)                       • cpu_callback_on_kernel_run (kernel_name)
                                                   OnKernelComplete
                                                   • cpu_callback_on_kernel_complete (kernel_name)
[GT-Pin only]                                  [GT-Pin only]
GTPin_Entry (argc_gtpin, argv_gtpin)           GTPin_Entry (..)
                                                       <start GPU analysis>
```

Figure 6.4: The control flow of XPU-Pin co-analysis tool for an x86 CPU and Intel GPU or NVIDIA GPU.

combining x86 Pin tool (xpu-pin-driver.so) and GTPin analysis tool (GPUAnalysis.so). Including a GPU analysis library can inadvertently cause the CPU analysis tool to treat it as part of the application. To prevent this, the CPU tool must explicitly exclude it from instrumentation. The XPU-Pin framework provides an API enabling the CPU analysis tool to retrieve the name of the GPU analysis library for this purpose.

6.3 The Imperative For Efficient Simulation of Heterogeneous Systems

This section highlights the need for efficient methodologies to evaluate the performance of large workloads running on heterogeneous computing systems in a fast and accurate way.

6.3.1 The Trend Towards Heterogeneity

The traditional Moore's law [9, 187] driven performance improvements have diminished in recent years [188]. Furthermore, multi-core scaling may be reaching its limits due to power constraints [167]. This marked a significant shift towards heterogeneous architectures, primarily driven by the increasing complexity and computational demands posed by artificial intelligence (AI) workloads. As the demand for high-performance computing (HPC) systems and data centers continues to grow, understanding and optimizing the performance of applications running on heterogeneous architectures becomes critical. The coexistence of multiple processing units, such as CPUs and GPUs, in these systems (typically known as an XPU) has become a standard of modern computing.

6.3.2 Limitations of Traditional Analysis Methodologies

Traditional heterogeneous systems tend to underutilize the available computing power of CPU and GPU resources [189, 190]. Most traditional heterogeneous applications use the CPU to schedule computing tasks for accelerators like GPUs. While the highly parallel computation happens in the GPU, the CPU waits, causing the CPU cycles to be wasted. However, this may not be the case with emerging applications that may fully utilize the CPU resources by executing tasks concurrently on the CPU and GPU. Independent performance evaluations of CPU and GPU using simulation techniques can yield misleading microarchitectural state estimations, especially in tightly coupled systems. These evaluations cannot accurately capture the shared memory and cache access patterns, which are influenced by the underlying cache coherency protocols. This independent analysis can lead to inaccurate microarchitectural states due to the misrepresentation of synchronization between the processing units. Consequently, resource usage and execution order might be misrepresented. Therefore, co-analysis and co-simulation techniques are essential for accurate microarchitectural state evaluation in CPU-GPU systems.

6.3.3 Effective Sampling of Heterogeneous Workloads

There are several methodologies that address the problem of sampling single-threaded [2, 20, 27] and multi-threaded [8, 32, 33, 34, 43] CPU workloads. There are several sampled simulation techniques that consider GPU workloads [16, 45, 46, 59, 60, 82, 83] to speed up GPU-only simulation. Among the prior works for the sampled simulation of GPU workloads, Kambadur et al. [59] proposed a GTPin-based methodology for the sample selection of Intel GPU workloads. The methodology utilizes basic block information (along with other program features) to characterize program execution. Prior works like TBPoint [45] and PKA [46] utilize handpicked feature vectors, including kernel size and control flow divergence, to classify similar GPU regions. Photon [16] employs a cluster-based summarization technique that groups similar warps based on their behavior and constructs basic block vectors (BBVs) for each cluster by aggregating individual warp profiles and concatenating them. All these works consider GPUs as independent computing units, which rely on the assumption that the heterogeneous workload could be divided into CPU-only and GPU-only components. Under this assumption, the total execution time of the heterogeneous application can be calculated by summing up the CPU execution time, GPU execution time, and the data transfer time between CPU and GPU. However, this assumption may no longer be valid for emerging workloads. Independent analyses may result in inconsistent timings for workload-specific CPU and GPU events, such as kernel launches, memory allocations, and warp divergence. Therefore, a combination of CPU-only and GPU-only sampling methods for heterogeneous systems could lead to inaccurate performance measurements.

6.3.4 Effects of Microarchitectural Warmup

In sampled simulation, microarchitectural warmup is necessary to ensure the simulated microarchitecture reflects a realistic state prior to detailed performance measurements. Previously proposed microarchitecture warmup methodologies [5, 88, 91, 92] enable the detailed simulation of the regions of interest starting at the right state. Warmup methodologies can be categorized into functional warming and statistical warming. Functional warming techniques [32, 33] rely on actual program execution, whereas statistical warming [5, 88, 91, 92] leverages profiling tools to gather memory access information.

XPU-Point provides a framework for collecting memory access information necessary for developing integrated CPU-GPU warmup methodologies in heterogeneous systems. Architects tend to integrate computing devices like CPUs and GPUs to share L3 cache,

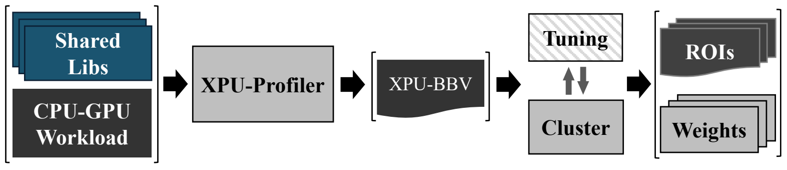


Figure 6.5: The workflow of XPU-Point methodology to capture representative regions (or ROIs) along with their corresponding weights suitable for the sampled simulation of heterogeneous workloads.

memory, etc., for higher throughput. For instance, NVIDIA Grace-Hopper [191] utilizes a CPU-GPU coherent memory model, while Apple's M-series processors deliver CPUs and GPUs on the same die that share memory. The trend towards tightly coupled CPU-GPU computing will continue, especially with the advancement in interconnects (like CXL [192] and NVLink [193]) and chiplet-based [194] IC packaging [195] technologies. This enables the tight integration of CPUs and GPUs within a single package, as seen in Intel's Lunar Lake architecture [196] or AMD's Exascale Heterogeneous Processor (EHP) architecture [99, 197, 198]. XPU-Point can be extended to gather shared memory access patterns, enabling the generation of combined warmup data that addresses this crucial requirement in integrated GPU systems. Multi-GPUs are widely used in high-performance computing [199] and large language models (LLM) [200] to accelerate their applications. In multi-GPU systems, CPUs coordinate the interaction between the GPUs. In this context, XPU-Point may be extended to collect warmup data to enable accurate sampled simulation of multi-GPU systems.

6.4 XPU-Point Sample Selection Methodology

In this section, we introduce XPU-Point, a novel methodology to sample heterogeneous CPU-GPU workloads. To the best of our knowledge, XPU-Point represents the first solution to efficiently co-sample heterogeneous workloads. The overall workflow for the XPU-Point methodology is outlined in Figure 6.5. The methodology relies on our XPU-

Profiler to generate the combined execution signature vectors of CPUs and GPUs. These heterogeneous execution vectors are clustered to identify representative regions that can be used for simulation-based performance evaluations of future heterogeneous architectures.

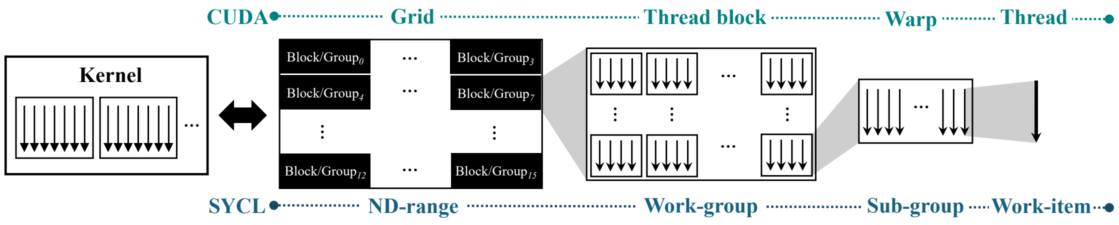


Figure 6.6: A comparison of the hierarchical structures used in CUDA and SYCL programming models to distribute kernel execution tasks, showing the level of granularity at which work is assigned to the execution units. CUDA primarily utilizes the SIMT execution model, while in SYCL, underlying architecture and implementations determine the execution model.

6.4.1 Workload Distribution on GPUs

GPUs follow a hierarchical structure in both their hardware and programming models to efficiently manage the massive number of threads. For example, NVIDIA GPUs typically comprise multiple Streaming Multiprocessors (SMs), with each SM containing several CUDA cores. To leverage the parallel architecture of NVIDIA GPUs, several threads (usually 32 or 64 threads) are grouped into a warp (or wavefront). NVIDIA GPUs primarily utilize a Single Instruction, Multiple Thread (SIMT) [201] model of CUDA, where threads within a warp share the same program counter (PC) and consequently execute the same instruction concurrently on the same CUDA core. Furthermore, multiple warps are grouped into thread blocks, which are then scheduled for execution on the same SM and utilize the shared cache. The GPU kernels (functions offloaded to the GPUs for parallel processing) are structured as Grids to orchestrate the execution across all thread blocks.

The concept of work groups in SYCL directly maps to how Intel's Xe [202] cores distribute tasks. Work-groups, analogous to thread blocks, group a defined number (SIMD-width) of threads for cooperative execution and data sharing. Intel GPUs typically employ a more flexible SIMD (Single Instruction, Multiple Data) [203] model redesigned for high performance [204] on Intel GPUs in the SYCL programming model. This means that threads within a work-group can execute a single instruction on multiple data elements simultaneously. Work-groups are subdivided into sub-groups (similar to warps) that share resources like local memory. Execution occurs on Vector Engines (VE) within the Xe cores. SYCL employs queues to manage the submission of work-groups for execution on the VE. ND-range defines the high-level structure of the kernel for parallel execution across the processing elements, specifying a multidimensional grid of thread blocks to be launched on the GPU. Figure 6.6 shows the workflow of a GPU kernel execution on Intel and NVIDIA systems.

XPU-Point takes into account both CUDA and SYCL programming models to represent the amount of execution done by the device. The execution in traditional CPU workloads can be quantified by the number of instructions or basic blocks (code blocks that have single entry and exit points) executed by each thread. In this work, we adopt a similar approach to quantify the execution of GPUs. However, GPU execution differs due to the SIMT/SIMD paradigm, where multiple threads or work-items collaborate to execute an instruction. To account for this, XPU-Point leverages warps or subgroups as the fundamental unit of execution on GPUs, analogous to instructions on CPUs.

6.4.2 Slices of Heterogeneous Applications

A slice (or region) represents a chosen segment of the application's execution flow generated by splitting the application at a well-defined point. For a slice to be effective for workload characterization, it must be *repeatable* across multiple runs of the application to ensure consistency in the behavior for accurate sampled simulation and performance

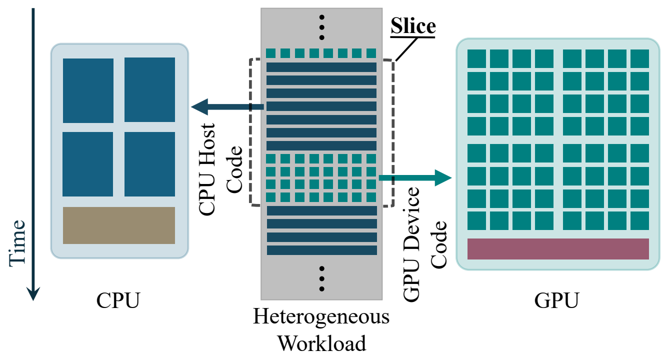


Figure 6.7: The representation of a slice (or region) in XPU-Point. A slice is defined as the execution window between consecutive kernel calls within a heterogeneous application.

evaluation. Traditional CPU workload sampling methodologies such as SimPoint [20], BarrierPoint [34], and LoopPoint [8] identify slices based on repeatable program constructs. Simpoint, for instance, focuses on identifying intervals based on fixed-size instructions. Meanwhile, BarrierPoint and LoopPoint target regions that are delineated by synchronization barriers and loops, respectively. Previously proposed sampled simulation techniques for GPU workloads [16, 45, 46] focus solely on the GPU kernels, completely ignoring any interactions with the CPUs. In this work, we propose a novel approach for slice identification, which is a contiguous code segment that spans from the end of one kernel call to the end of the subsequent kernel call, as shown in Figure 6.7. Therefore, the slice of a heterogeneous application includes both CPU and GPU execution. Similar to loops in LoopPoint or inter-barrier regions in BarrierPoint, the slices identified by XPU-Point are repeatable across multiple executions on platforms with similar compute capabilities.

6.4.3 Capturing Heterogeneous Execution Profiles

Understanding execution profiles within CPU-GPU systems demands a comprehensive representation that integrates execution profiles from both processing units. Traditionally, the classification of regions based on the similarity of the code executed [18, 19] works well for CPU workloads. The regions are represented as basic block vectors (BBVs), which comprise basic blocks and their frequency. A number of prior works on selecting representative regions of a workload have been built on this idea of representing regions using code signatures. In the case of multi-threaded workloads where threads split the work to execute on multiple cores, the amount of work done by the threads is represented by concatenating the BBVs of each thread. Concatenating BBVs across CPU and GPU threads is evident to be a promising technique, as it merges CPU thread profiles with detailed GPU data, including both global and individual thread profiles. Prior works show that concatenating per-thread profiles to form CPU BBV [8, 34, 164] or per-warp profiles to form GPU BBV [16] leads to the accurate representation of thread-level parallelism.

In this work, we devise a technique to represent the heterogeneous regions of the workload. We utilize XPU-Profiler, the profiling tool that we built upon XPU-Pin, to simultaneously generate BBVs for CPU and GPU execution. Within this framework, we refer to CPU BBVs as those derived from program execution on the CPU, while GPU BBVs refer to those obtained during program execution on the GPU. We demonstrate that concatenating all GPU warps (or sub-groups) is efficient in representing the GPU BBV. By concatenating the CPU BBV and GPU BBV forming XPU-BBV (shown in Figure 6.8), we construct a unified representation that captures the behavior of a heterogeneous region. Previous studies [8, 34] have demonstrated that concatenating feature vectors effectively captures individual thread behavior.

6.4.4 Selecting the Representative Slices

Representing the sheer number of GPU threads (or warps) in an XPU-BBV leads to a significant increase in vector dimensionality, resulting in the *curse of dimensionality* [205]. This phenomenon slows down clustering algorithms, which are critical for

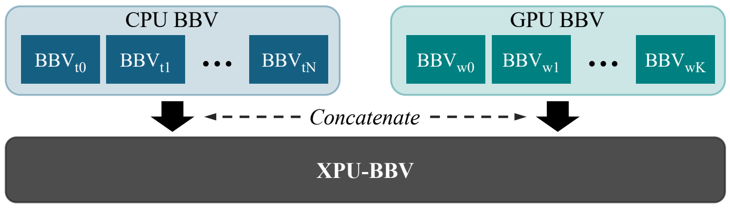


Figure 6.8: The concatenation of CPU and GPU BBVs into a longer, combined XPU-BBV that represents a heterogeneous region in XPU-Point methodology.

identifying representative regions from the profile data. To address this challenge, we employ traditional dimensionality reduction techniques such as random linear projections. The algorithm selected for a desired level of dimensionality reduction is pivotal in minimizing information loss within the profile data [206]. This ensures that the resultant lower-dimensional space retains the vital characteristics necessary for accurate workload characterization. Due to the differences in magnitude between CPU and GPU dimensions, these feature vectors need to be projected separately. Further, we employ k-means clustering algorithm [76] to cluster these heterogeneous regions as represented by the lower-dimensional BBVs. The region closest to the centroid of each cluster serves as the representative of the cluster [20]. Advances have been made in program representation using execution embeddings [207] and in clustering using deep neural networks [208]. We believe that such improvements are orthogonal to the basic idea of XPU-Point and can be incorporated here.

6.4.5 Sample Validation and Tuning

The representativeness of the regions selected using the proposed methodology needs to be validated. Sample validation, employing real hardware measurements like those demonstrated in prior works [60], can be leveraged here. To validate the representativeness of the selected slices within heterogeneous workloads, we introduce XPU-Timer, a tool built upon XPU-Pin. XPU-Timer leverages the x86 `rdtsc` instruction to provide system timestamps (TSCs) at critical points during the native execution of the workload: program start, program finish, and the boundaries of each predefined slice. These timestamps allow us to extract the execution time for each representative slice. By weighing these region execution times with their corresponding weights, we extrapolate the execution time of the entire program. The difference between the full execution time measured and the extrapolated time is known as the sampling error (or prediction error). A lower sampling error indicates a more accurate selection of representative slices.

6.4.6 Estimating the Full Application Performance

Microarchitecture simulation and exploration greatly benefit from sampling large, heterogeneous workloads. Instead of simulating entire workloads for microarchitecture exploration, which is computationally expensive, representative slices of the workload can be simulated rapidly. These slices capture the complex characteristics of heterogeneous workloads, enabling researchers to explore how future microarchitectures can be optimized for such workloads.

XPU-Point identifies representative slices of a heterogeneous workload that can be used for detailed cycle-level or cycle-accurate microarchitecture simulations. The regions can also be simulated on execution-driven heterogeneous simulators like Multi2Sim [96] and gem5-gpu [97]. We would like to point out that the XPU-Pin framework can be used for heterogeneous trace generation to support trace-driven simulators like MacSim [95]. Accurate performance measurements in sampled simulations rely on a warmed-up microarchitectural state before detailed simulation. However, warmup reconstruction for heterogeneous systems remains an open research area. As this falls outside the scope of our current work, we will not explore it further in the chapter.

6.5 Experimental Setup

Establishing the fidelity of workload sampling techniques – the ability to accurately represent full program behavior using the selected sample set – is essential. Simulation-based performance comparisons are typically employed to assess this characteristic. However, this approach is impractical for large, heterogeneous workloads. Thus, we follow a hybrid evaluation approach by employing simulation-based performance evaluation for shorter applications and hardware-based performance measurement using XPU-Timer for all applications.

For our evaluation, we use a combination of standard heterogeneous benchmarks as well as real-world HPC and AI workloads that use both CPUs and a GPU for computation. We evaluate SPECaccel 2023 [209] benchmarks and SPEChpc 2021 [210] benchmarks, along with real-world workloads like AutoDock [211, 212, 213], GROMACS [214], and PyTorch [215] inference runs.

Table 6.1: The combinations of CPUs and GPUs for Intel- and NVIDIA-based systems used to evaluate XPU-Point methodology.

CPU	**GPU**
Intel Alder Lake [216]	Intel Discrete Graphics 2 (DG2)
Intel Alder Lake	Intel Iris Xe (integrated)
Intel Ice Lake-SP [217]	Intel Ponte Vecchio (PVC) [218]
Intel Sapphire Rapids [219]	Intel Ponte Vecchio (PVC)
Intel Cascade Lake [220]	NVIDIA A100 [221]
Intel Skylake [222]	NVIDIA GeForce GTX 1080
Intel Skylake	NVIDIA TITAN Xp

Further, we also evaluate all of the workloads with XPU-Timer using native hardware runs on both Intel-GPU-based and NVIDIA-GPU-based heterogeneous systems, and, therefore, we separately compile the benchmarks suitable for these systems. For Intel-based systems, we use Intel's oneAPI [223, 224] toolkit to build the benchmarks, whereas for NVIDIA-based systems, we use the NVIDIA CUDA toolkit [225]. The machines that we used for our evaluation are listed in Table 6.1. Our focus is on demonstrating the methodology's efficacy across heterogeneous workloads on both Intel-based and NVIDIA-based GPU systems. To isolate this aspect, we present the evaluation results for each system type in separate graphs. This approach avoids comparisons of individual machine performance and emphasizes the broader applicability of the methodology. Given the evaluations done in this work, we aim to show that this methodology is applicable across other heterogeneous architectures.

6.6 Evaluation

This section evaluates the effectiveness of XPU-Point in selecting representative regions using realistic CPU-GPU heterogeneous workloads. The aim of this work is to allow for fast and accurate microarchitecture simulations of these workloads to explore future heterogeneous systems.

We extrapolate the performance of the full workload from the performance of N representative regions using the formula:

$$P_{proj} = \sum_{i=1}^{N} P_i \times multiplier_i,$$

where P_{proj} denotes the projected or extrapolated performance of the full workload. In addition, P_i and $multiplier_i$ denote the performance obtained and the multiplier associated with the representative region $region_i$, respectively. The multiplier of a representative region is dependent on the number of regions that belong to the cluster that $region_i$ represents [34]. This formula allows us to extrapolate performance metrics like runtime, cache behaviors, branch behaviors, and IPC for the entire workload.

Sampling Error and Speedup. We quantify the difference between the extrapolated performance metrics and the actual measured performance of the full workload to obtain sampling error or prediction error [20]. We estimate the performance of the workloads and the representative regions leveraging the system timestamp counter (TSC) on real hardware, which is equivalent to runtime obtained through microarchitecture simulation. The long simulation times required for full workloads make simulation-based validation impractical for large workloads.

The sampling error Δ_{sample} can be computed using the formula:

$$\Delta_{sample} = \left| 1 - \frac{P_{proj}}{P_{real}} \right|,$$

where P_{real} is the actual performance obtained through the measurement of the full workload. Usually, sampling error is expressed as a percentage (error rate). This is obtained by multiplying the absolute value of the sampling error (Δ_{sample}) by 100.

We define the speedup obtained using XPU-Point as the reduction in the amount of work to be analyzed or simulated in detail after sampling [34]. That means,

$$speedup = \frac{num\_slices}{N},$$

where N is the number of representative regions, and num_slices is the total number of slices in the entire workload. The speedup we show here is equivalent to the serial speedup, which is achieved by simulating the representative regions sequentially. Note that this speedup represents the minimum achievable reduction in simulation time. The simulation of these representative regions can be parallelized, which could lead to significantly higher speedups than the values presented here.

Cross-microarchitecture Validation. XPU-Point relies on the microarchitecture-independent selection of representative regions. This allows researchers to profile an application binary on one hardware and reuse the chosen regions for simulations on different hardware within the same architecture. This is possible because XPU-Point

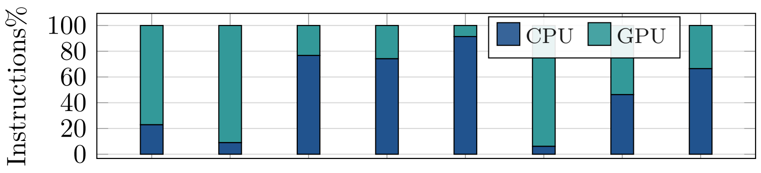


Figure 6.9: The instruction split between CPU and GPU for loop executions in SPECaccel 2023 benchmarks using train inputs.

utilizes BBVs to capture the control flow structure of the program, a characteristic independent of the underlying architecture. To verify the effectiveness of this approach across microarchitectures, XPU-Point employs cross-microarchitecture validation. This validation involves selecting regions on one machine using XPU-Profiler and then validating their representativeness on another machine with a different CPU-GPU combination within the same architecture using XPU-Timer.

Runtime Overhead. XPU-Point, like other dynamic binary instrumentation tools, introduces analysis-dependent runtime overhead. XPU-Profiler has a large overhead due to extensive library usage and process/thread creation of large workloads. By default, it instruments all libraries, processes, and threads. However, XPU-Point offers the flexibility to reduce the analysis overhead by instrumenting specific processes/threads and libraries. The XPU-Timer tool employs a Pin-probes mode driver, avoiding CPU instrumentation altogether. The GPU component of the tool utilizes low-overhead instrumentation to track key events like GPU initialization and kernel start/stop.

6.6.1 Comparison with GPU Sample Selection

We present a detailed analysis of SPECaccel 2023 [209], a benchmark suite with computationally intensive parallel heterogeneous applications that exercises the performance of the accelerator (GPU in our case), host CPU, memory transfer between host and accelerator, compilers, and the runtime system [226]. We used Intel x86 Sapphire Rapids server with Intel Data Center GPU Max 1100 for this evaluation.

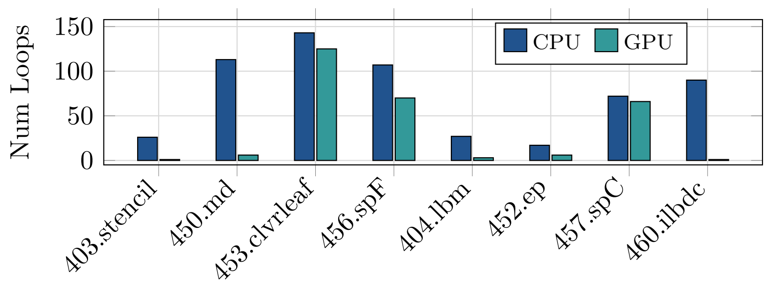


Figure 6.10: The number of loops executed on CPU and GPU in SPECaccel 2023 benchmarks using train inputs.

Figure 6.10 shows the analysis of loops in the main image of the benchmarks identified using Intel Software Development Emulator (Intel SDE) [227]. The number of loops on the GPU was obtained using Intel GTPin [81]. For SPECaccel 2023 workloads using CPU and GPU, we wanted to test the effect of focusing just on the GPU computation. We tested two profilers: XPU-Profiler that collects combined CPU-GPU BBVs; and GPU-Profiler that collects the GPU BBVs. In both the cases, the region boundaries are kernel boundaries leading to the same number of BBVs. GPU-Profiler uses Intel GTPin to collect per-thread BBVs for the entire computation which are copied to the CPU at the end of each GPU kernel execution. The average slowdown of the GPU-Profiler for SPECaccel test cases was 4.7×. XPU-Profiler on the other hand uses Pin JIT mode instrumentation at the basic block level. Synchronization between multiple threads is necessary in this case. The average slow-down for the XPU-Profiler for the SPECaccel test cases was 102×.

Figure 6.11 plots the sampling errors for SPECaccel 2023 benchmarks using XPU-Point and GPU-Point evaluations. Overall, the sampling errors with GPU-only approach (geometric mean of 23.9%) are higher than those with the combined CPU-GPU approach (geometric mean of 0.99%). In the case of 452.ep, focusing on just the GPU computation in isolation predicts the overall performance with a low error (1.34%) although still higher than the combined CPU-GPU approach (0.10%). 404.lbm demonstrates another

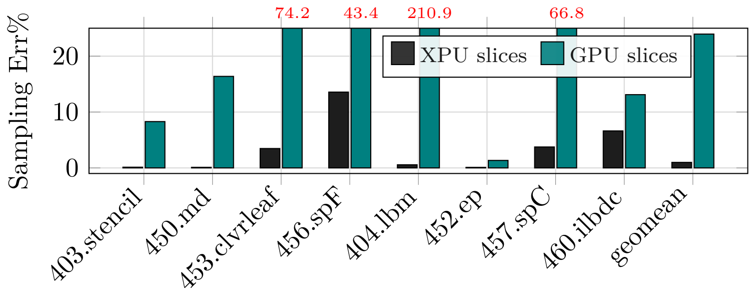


Figure 6.11: The sampling errors for the SPECaccel 2023 benchmarks with GPU-only profiles (GPU-Point) vs. CPU-GPU profiles (XPU-Point).

extreme, where the GPU-only approach only found one representative region leading to 210% sampling error. Using heterogeneous profile, 15 regions were identified by XPU-Point leading to a much lower sampling error (0.56%).

6.6.2 Sample Validation using Native Hardware

While simulation provides a controlled environment for workload analysis, validating samples on native hardware is often practical for large workloads. To enable this, we employ XPU-Timer to gather precise performance metrics from native hardware executions, as mentioned in Section 6.4.5. The results of sample validation using XPU-Timer, categorized by benchmark suite, are presented here.

6.6.2.1 SPEChpc 2021

The SPEChpc 2021 benchmark suites [210] provide application benchmarks from well-selected science and engineering codes that are portable across CPUs and accelerators. The suites include Tiny, Small, Medium, and Large workloads, supporting multiple programming models and requiring varying amounts of memory and number of ranks to run. We chose the Tiny workload (60 GB memory requirement) and limited our testing to a single node/rank. We tested both the *test* and *ref* inputs for evaluation. The sampling errors and speedups for the benchmarks running test inputs are shown

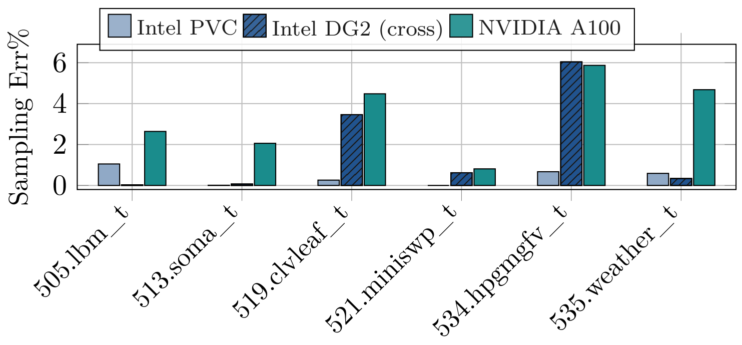


Figure 6.12: The sampling errors plotted for the **SPEChpc 2021** benchmarks with **test** inputs from the tiny set. The benchmarks are sampled on an Intel PVC machine are cross-validated on an Intel DG2 machine. The benchmarks were also sampled and validated on an NVIDIA A100 machine.

in Figure 6.12 and Figure 6.13 for the Intel-based systems and NVIDIA-based systems. Due to the huge memory requirements, we evaluated the ref input set of SPEChpc benchmarks only on the Intel-based systems, and the sampling errors and speedups are shown in Figure 6.14 and Figure 6.15.

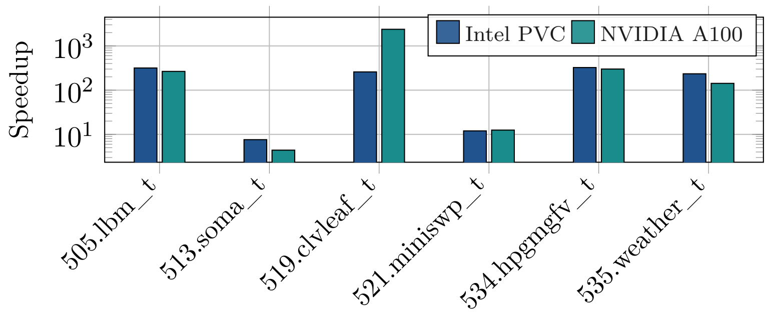


Figure 6.13: The simulation speedup plotted for the **SPEChpc 2021** benchmarks with **test** inputs from the tiny set. The benchmarks are sampled on an Intel PVC machine are cross-validated on an Intel DG2 machine. The benchmarks were also sampled and validated on an NVIDIA A100 machine.

6.6.2.2 AutoDock-GPU

AutoDock is a widely used software that performs molecular docking simulations. AutoDock is commonly used for benchmarking and performance evaluation of heterogeneous sys-

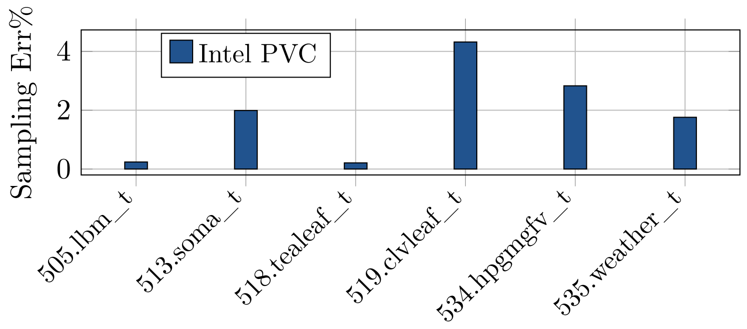


Figure 6.14: The sampling errors obtained for the representative regions identified for **SPEChpc 2021** benchmarks that use **ref** inputs from the tiny set. The representative regions of the benchmarks are generated and validated on an Intel PVC machine.

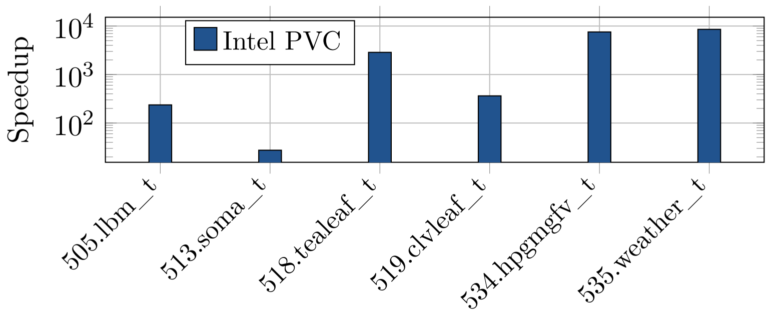


Figure 6.15: The speedup obtained for the representative regions identified for **SPEChpc 2021** benchmarks that use **ref** inputs from the tiny set.

tems. Figure 6.16 and Figure 6.17 show the sampling results obtained on Intel-based systems and NVIDIA-based systems for AutoDock using XPU-Point. We use three different platforms to validate the sample selected for the AutoDock [211, 213] application using various inputs. The SYCL implementation of AutoDock [212] is used for evaluating Intel GPU systems.

6.6.2.3 GROMACS

GROMACS [214, 228] is a widely used open-source tool [229] for the simulation of molecular dynamics, which uses both CPU and GPU for computation. We manually configure the type of computation to be offloaded to the CPU and GPU, as shown

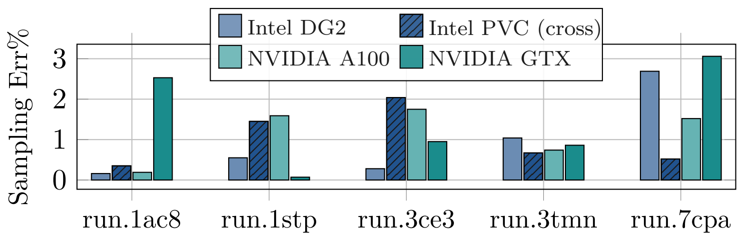


Figure 6.16: Sampling errors for **AutoDock** (work-item=8) using different inputs on Intel and NVIDIA GPU platforms.

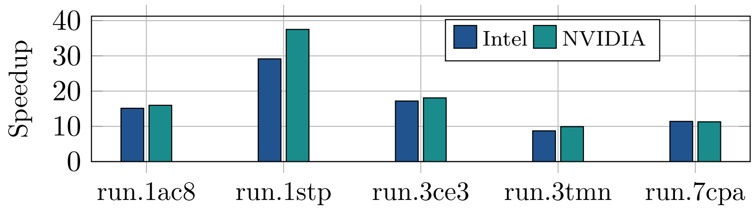


Figure 6.17: The speedup obtained for **AutoDock** (work-item=8) using different inputs on Intel and NVIDIA GPU platforms.

in Table 6.2, and identify the representative regions. We evaluate all possible cases of configuring GROMACS to split the computation across both CPU and GPU. The sampling errors and speedups are reported in Figure 6.18 and Figure 6.19 for Intel-based systems and NVIDIA-based systems. We infer that the GROMACS workload Type F, with the most number of slices, benefits the most from the XPU-Point methodology achieving the maximum speedup. For Type A, the regions are expected to be larger due to the predominantly CPU-intensive nature of the workload.

Table 6.2: The classification of **GROMACS** based on the offloading device for the execution of each calculation. We also use `-nsteps 200` with `-notunepme` for all types. The last column shows the number of slices for each type.

Type	nb	pme	pmefft	bonded	update	#slices
A	GPU	CPU	CPU	CPU	CPU	305
B	GPU	CPU	CPU	GPU	CPU	506
C	GPU	GPU	CPU	CPU	CPU	707
D	GPU	GPU	CPU	GPU	CPU	908
E	GPU	GPU	GPU	CPU	CPU	3730
F	GPU	GPU	GPU	GPU	CPU	3931

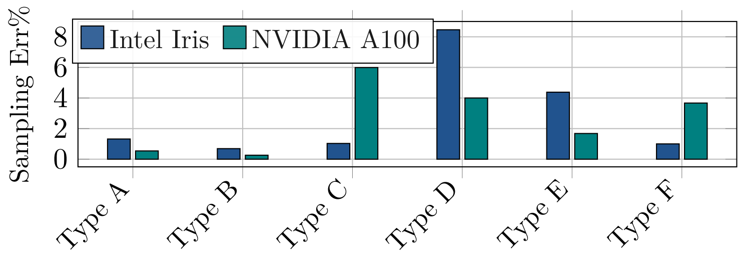


Figure 6.18: The sampling errors for **GROMACS** in different settings on Intel Iris and NVIDIA A100 using XPU-Point.

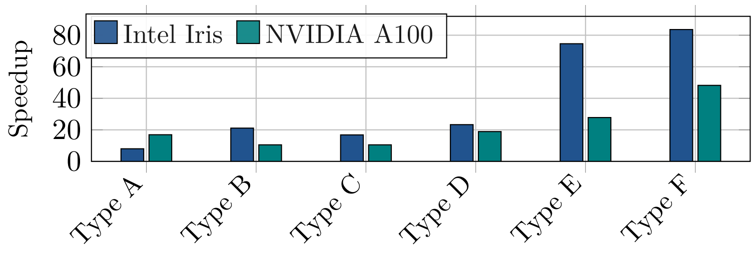


Figure 6.19: The speedup obtained for **GROMACS** in different settings on Intel Iris and NVIDIA A100 using XPU-Point.

6.6.3 Evaluation of PyTorch Inference Workloads

We evaluated PyTorch [230] inference workloads running text processing tasks using the BERT (Bidirectional Encoder Representations from Transformers) [231] model and image classification tasks using the ResNet50 (Residual Network with 50 layers) [232] model. It compares performance across various configurations: data precision (BF16, FP16, FP32, and INT8 quantization) and execution mode (imperative Python vs. pre-compiled TorchScript). INT8 quantization represents numbers using just 8 bits, significantly improving performance compared to higher precision formats but requiring a pre-quantization process. These workloads were optimized with the Intel Extension for PyTorch [215] to be evaluated on the machines with Intel PVC GPUs.

We present the sampling errors of PyTorch workloads using XPU-Point in Figure 6.20 and Figure 6.21. Profiling more libraries caused the XPU-Profiler overhead to increase as

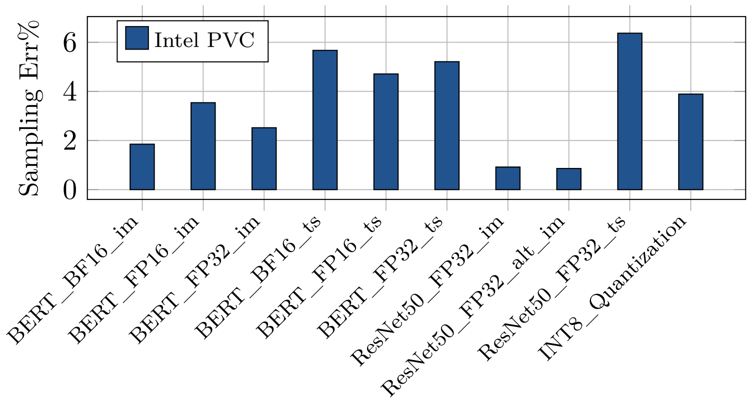


Figure 6.20: The sampling errors obtained for **PyTorch Inference runs** using XPU-Point on Intel PVC. im=Imperative, ts=TorchScript.

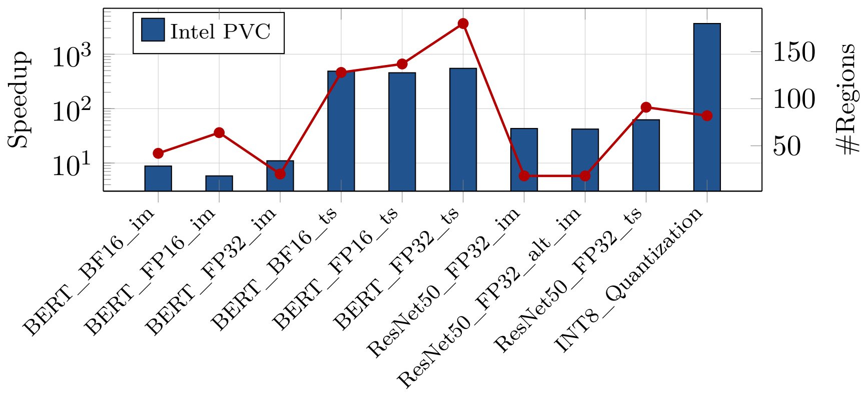


Figure 6.21: The speedups obtained during the simulation of **PyTorch Inference runs**. The line graph (plotted with the secondary y-axis) shows the number of representative regions selected using XPU-Point. im=Imperative, ts=TorchScript.

expected, as we observe for `BERT_BF16_Ts`, `BERT_FP16_Ts`, and `BERT_FP32_Ts`, although the cost of profiling will be amortized over multiple simulations. In general, the profiling and analysis of all shared libraries is necessary. To speed up the analysis, we chose to analyze the libraries that significantly impact workload runtime.

The PyTorch workloads use a large number of libraries (more than 150), processes (around 60), and threads (more than 100), causing a large overhead in analyzing them. In this work, we focused on the main libraries and processes during instrumentation.

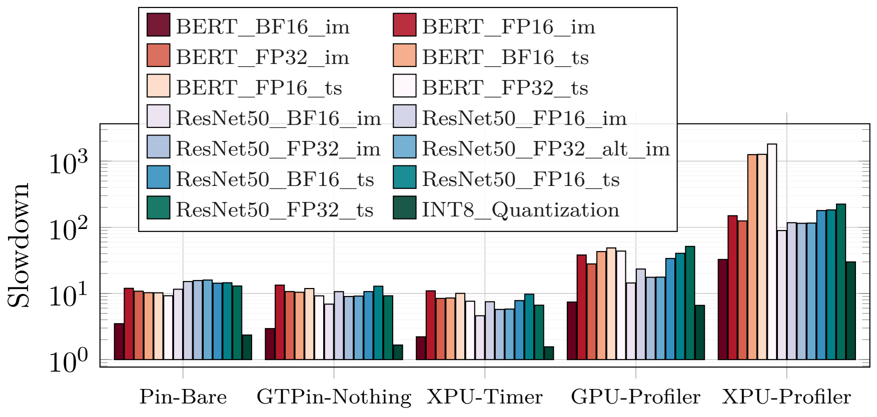


Figure 6.22: The slowdowns (normalized with the native runtime of the application) for **PyTorch Inference runs** on Intel Ponte Vecchio GPU. The slowdown in *Pin-Bare* mode measures the slowdown due to running the benchmarks under Pin with no instrumentation. To evaluate the slowdown caused by the GTPin Tool, we use a basic instrumentation tool, *Nothing*. *XPU-Timer* uses XPU-Pin to collect the timing information of the benchmarks. The *GPU-Profiler* profiles the benchmarks using GTPin to collect BBVs. *XPU-Profiler* uses XPU-Pin to collect BBVs of the CPU-GPU execution concurrently.

Figure 6.22 shows the run-time overhead of various evaluation tools used with these workloads.

6.7 Related Work

We have discussed the most relevant previous works in Section 6.3. Workload sampling has been an active research area for several decades, and several techniques were proposed for CPUs [2, 8, 20, 27, 30, 32, 33, 34, 43, 78, 80, 164] and GPUs [16, 45, 46, 59, 60, 82, 83] in this direction primarily for the reduction of simulation time and resources. Several CPU simulators [6, 14, 93], GPU simulators [7, 98, 101, 102], and heterogeneous CPU-GPU simulators [95, 96, 97, 99, 100, 233] are available for performance estimation. However, simulating large workloads on cycle-level simulators is prohibitively time-consuming.

Several programming models cater to heterogeneous computing, including OpenMPI [234], StarPU [235], OpenCL [236], OpenMP [237], OmpSs [42], CUDA [225, 238], and AMD HIP [239]. OpenCL and CUDA are the most widely adopted programming models for heterogeneous platforms. OpenCL is an open standard for programming heterogeneous platforms that enables programmers to write portable code. CUDA is a vendor-specific programming model optimized for NVIDIA GPUs, offering a suite of libraries and tools to maximize performance. Prior works [240, 241, 242, 243] compare the performance of CUDA and OpenCL programming models and show that the translation of one model to another works well for various applications. SYCL [244] is a modern heterogeneous programming model built on C++. SYCL programs are structured with two distinct components: host code and device code (kernels) where the host code is executed on the CPU, and the kernels execute on either the host or an accelerator (like GPU). There are several implementations of source-to-source translation tools from CUDA to SYCL [212, 245]. In this work, we use SYCL (Intel's implementation [246]) and CUDA programs to evaluate Intel and NVIDIA GPU systems, respectively.

6.8 Conclusion and Future Directions

In the wake of the ever-increasing demands of AI workloads, effectively evaluating large workloads on heterogeneous architectures has become ever more significant. This chapter proposes XPU-Point, a methodology for the sample selection of heterogeneous CPU-GPU workloads. XPU-Point leverages XPU-Pin, our instrumentation framework to combine CPU and GPU analysis. We demonstrate the accuracy and efficiency of the XPU-Point through the evaluation of real-world heterogeneous workloads, highlighting its ability to significantly reduce the simulation time. This work forms the basis for selecting representatives to use in a host of simulators, from cycle-level to higher-level architectural simulation methodologies.

XPU-Point is the first, to our knowledge, to propose a sample selection methodology targeted for heterogeneous workloads. The methodology lays a solid foundation for future enhancements, including support for multiple accelerator types. Our current focus is on the sample selection of workloads where the primary process manages kernel launches. To address complex scenarios, CPU computation loops can be combined with GPU kernel invocations to form repeatable code regions. Although not discussed in this chapter, XPU-Point can be extended to support multi-GPU systems, which can be enabled by combining synchronized region profiles from individual GPUs. As system complexity increases, particularly with advanced interconnects like CXL and NVLink, system-specific considerations become crucial for effective sampling techniques. This work assumes that the simulator models interconnect effects, making the proposed methodology broadly applicable.

Chapter 7

ROIperf: Rapid Validation and Iterative Tuning of Workload Sampling Methodologies

The measure of the information content is the measure of the degree of uncertainty or the degree of surprise.

— Claude Shannon

Accurately evaluating processor performance for future architectures on cycle-accurate simulators is time-consuming. While workload sampling offers a faster alternative, validating the representativeness of selected regions of interest (ROIs) requires full program simulations, which becomes impractical for large workloads. This work introduces ROIperf, a framework that leverages real hardware to evaluate both the full workload and ROIs. This allows for faster validation of workload sampling, particularly for complex, long-running workloads.

7.1 Introduction

Cycle-accurate, detailed simulation of computer systems tends to be extremely slow, with simulation speeds of complex, modern processor designs can be as low as a few thousand instructions per second, that is, more than 100,000× slower than native speeds.

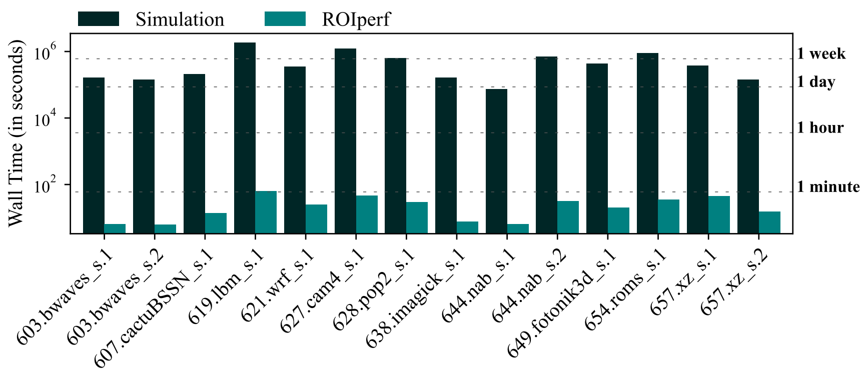


Figure 7.1: A comparison of the total wall-time required to validate the representative regions identified for the multi-threaded SPEC CPU2017 benchmarks using *train* inputs (the gap is expected to increase for *ref* inputs). The bars show a comparison of the minimum wall time taken to validate the regions (selected using LoopPoint [8] methodology) on a cycle-level simulator and the ROIperf framework.

Simulating large modern applications with trillions of instructions in their entirety is, therefore, not practical when using these methods directly. Instead, simulation of *regions of interest* (ROIs) from large application executions and extrapolating the full-program performance is a standard technique employed [2, 8, 20, 30, 32, 33, 34, 65]. To gain confidence in the extrapolated results, it is necessary to validate that the ROIs selected closely represent full-program behavior [62, 63, 64]. Traditionally, such validation is done by comparing the simulated performance of the entire program with the performance extrapolated from ROI simulations. However, since full-program simulation for most realistic applications is impractical, to begin with, such simulation-based validation is limited to either short-running programs and/or using fast but inaccurate simulators.

Performance monitoring on native hardware offers a significantly faster alternative for sample validation compared to traditional architecture simulators. Figure 7.1 shows that the validation of representative regions using our proposed ROIperf methodology can be performed at near-native speed, while simulation-based validation can take weeks or months. Accurately identifying representative regions within an application typically involves iterative parameter tuning and re-validation. For example, applications like `gcc` may require up to five times more representative regions than other applications, as shown in previous works [20]. Without extremely fast techniques, it becomes impractical to validate the efficacy of workload sampling methodologies for large-scale applications. In this work, we aim to provide a solution to this challenge to enable rapid sample validation without the need for long-running simulations.

Although measuring full-program performance on native hardware is well-established [247, 248], isolating and measuring the performance of specific regions of interest presents a significant challenge. To keep simulation times in check, ROIs are often significantly smaller than the full-program execution. These ROIs might only consist of a few million instructions, running for just milliseconds on real hardware. Precisely gathering performance data solely for the ROIs on native hardware with high fidelity is challenging. Loop-based representations of ROIs, for instance, offer high accuracy and reproducibility [8]. However, current hardware architectures lack native support for directly identifying such representation of regions. In an attempt to address this challenge, we present ROIperf, a methodology that incorporates lightweight instrumentation to achieve the necessary control and precision for isolating regions of interest. ROIperf utilizes Pin in probe mode [116], which is low overhead as it operates by patching an in-memory image of the application instead of using just-in-time (JIT) compilation, which can introduce significant performance overheads [185] that interfere with the workload behavior. While the instrumentation capability of a Pin probe tool is limited, its low overhead makes it ideal as a building-block for ROIperf. ROIperf uses the Pin probe to merely hook into the application execution at the beginning and register callbacks based on hardware performance counters guided by the ROI specification.

The Repeatability Challenge

Profile-based simulation region selection techniques, like SimPoint [20], typically require at least two program executions. The first run gathers profiling data to identify the ROIs for simulation. Subsequently, the second execution simulates the selected ROIs. Profile-based sample selection methodologies assume identical program behavior across executions, which is difficult to guarantee, especially for multi-threaded programs. Heterogeneity in hardware environments (for instance, varying ISA support leading to scalar vs. vectorized runs), inconsistencies in system libraries, and timing-dependent control flow (for example, work stealing in parallel applications) can all introduce discrepancies between profiling and simulation runs.

Several works have been proposed to ensure repeatable program execution during profiling and simulation. PinPlay [77] utilizes a record-and-replay framework, guaranteeing identical behavior across analyses by capturing the entire program execution and then performing profiling/simulation using a deterministic replay. However, PinPlay's replay incurs significant overhead ($\approx 50\times$ slowdown), rendering performance counter-based evaluation inaccurate. Other efforts to improve repeatability include using static binaries, checkpoints [249], or ELFies [47]. However, none of these techniques guarantee fully repeatable execution, particularly in multi-threaded scenarios where timing-dependent control flow and the resulting execution divergence happen more often [73, 124].

While ROIperf leverages native program execution to validate the samples or ROI, we acknowledge the inherent challenge of guaranteeing perfect repeatability across runs. However, the effects of this challenge can be minimized by executing both the sample selection and ROIperf measurements in a strictly controlled environment. We describe tests for the applicability of ROIperf prior to the measurement in Section 7.5. In our evaluations, we find that ROIperf is effective in identifying the regions accurately in most cases.

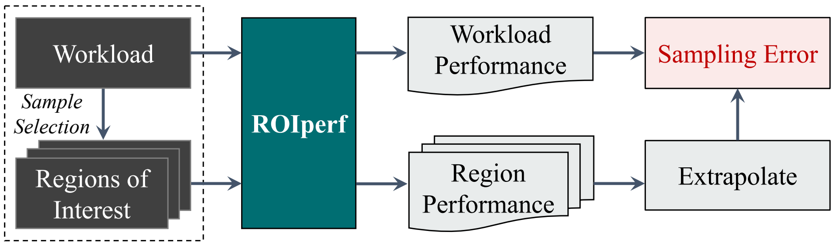


Figure 7.2: An overview of the working of ROIperf framework to validate the regions of interest (ROIs). The performance of the full workload and the ROIs are measured on the native hardware. The extrapolated performance is compared with the performance of the full runs to quantify the sampling error.

Contributions and Organization of the Chapter

To summarize, the chapter makes the following contributions:

1. We introduce ROIperf, a framework designed to rapidly evaluate the effectiveness of workload sampling methodologies. We demonstrate its application in validating regions of interest (ROIs) for long-running single-threaded, and multi-threaded workloads.
2. We leverage previously proposed PinPoints [24] (for single-threaded programs) and LoopPoint [8] (for multi-threaded programs) methodologies for ROI selection. Both methodologies rely on profiling based on deterministic replay [77] for region selection.
3. We also introduce sanity tests (detailed in Section 7.5.1) to assess reproducibility and identify extreme deviations in control flow in program execution. For accurate ROI validation, achieving identical control flow between the ROIperf run and the profiling run used for ROI selection is essential.
4. We will open-source the ROIperf infrastructure for use in the community, enabling the sample validation of large, realistic applications at near-native speeds, which was not possible before.

The rest of the chapter is organized as follows. In Section 7.2, we discuss the background on workload sampling methodologies and techniques for sample validation. Section 7.3 presents the ROIperf methodology and implementation details. We then describe the experimental infrastructure in Section 7.4, followed by an extensive evaluation of ROIperf in Section 7.5 to demonstrate the applicability of the proposed methodology. Finally, we present the related work in Section 7.6 and conclude the chapter in Section 7.7 with some possible future directions.

7.2 Background

In this section, we provide a background on the existing workload sampling techniques, particularly those leveraged in this chapter for sample selection. We also provide an overview of previously proposed sample validation techniques.

7.2.1 Sample Selection Methodologies

SimPoint [20] and SMARTS [2] are two well-established techniques for sampling single-threaded applications to accelerate simulation. SimPoint is a profile-driven methodology that identifies representative regions for simulation, whereas SMARTS employs statistical sampling for fast simulation. We employ SimPoint to select representative regions of single-threaded applications.

Accurately sampling multi-threaded workloads presents a significant challenge. Applying naive extensions of single-threaded techniques directly proves ineffective due to factors like thread interactions and spin-loops [31]. There are several techniques proposed to sample multi-threaded workloads [8, 30, 32, 33, 34], each having its own limitations, as discussed in Chapter 2. In this chapter, we evaluate LoopPoint [8] methodology that applies to generic multi-threaded workloads. LoopPoint identifies regions bounded by loop entries and can achieve high simulation speedups without compromising on sampling accuracy.

7.2.2 Sample Validation

Validating the representative regions identified for a large application can be tedious. This typically requires the full simulation of the application and the representative regions. While FPGA-based simulation infrastructures like FireSim [50] offer faster execution compared to traditional cycle-level software simulators, their turnaround time is still not negligible. Moreover, the hardware implementation of each component within the limited memory of the FPGAs is challenging.

Perelman et al. [21] propose a technique to select statistically valid representative regions early in the application to reduce the fast-forward time to reach the simulation regions. Gottschall et al. [250] propose SimPoint validation with TraceDoctor, an instrumentation framework attached to FireSim. This technique can be used to validate SimPoints for a RISC-V model running on FPGAs at high speeds. While significantly faster than detailed simulation, it remains slower than hardware validation and is limited to FPGA-based models.

7.2.3 Hardware Performance Counters

Modern microprocessors have special hardware registers called hardware performance counters for monitoring various performance-related metrics [66]. These counters provide the ability to measure performance in real-time and are typically used by software performance tools to measure metrics like the number of instructions executed, cache misses, page faults, etc [251]. By measuring these metrics, performance tools can help developers identify bottlenecks and other performance issues in their software. Because these counters are built into the hardware, they are able to provide measurements of performance-related metrics with very low overhead.

7.2.4 Instrumentation using Pin

Pin [116] is a well-known dynamic instrumentation and analysis framework for x86 applications. It offers an application programming interface (API) for adding extra code at various points within a program. The API differs based on the mode specified during Pin initialization. Pin supports two modes: (a) a just-in-time (JIT) mode which translates the test program in memory and adds instrumentation during the translation, and (b) a probe mode which merely patches an in-memory copy of the program with extra code. The JIT mode API allows for sophisticated run-time analyses but at the cost of translation overhead. The probe mode API is limited to adding extra code only at specific program points, although the overhead of such an addition is very low. ROIperf leverages Pin in probe mode due to its low overhead, which minimizes perturbation during performance measurement of the target application.

7.3 Methodology and Implementation Details

This section describes the implementation details of the ROIperf methodology. We will further present and compare the usage models of the methodology for single-threaded and multi-threaded applications.

7.3.1 ROI Selection using Sampling

To select representative regions of interest (ROIs), we employ phase-based and loop-based approaches. For single-threaded applications, we leverage the PinPoints methodology [24]. This method builds upon SimPoint methodology [20] where an application is profiled to generate basic block vectors at every execution slice (indicating *unit of work*), and the resulting vectors are clustered to identify multiple phases in the application. A representative ROI is chosen for each phase, weighted proportional to the size of the phase it represents. Similarly, we use LoopPoint methodology [8] to identify the representative ROIs of multi-threaded applications. LoopPoint demarcates application regions based on loop iterations (instead of instruction counts) and clusters these regions to select ROIs. The ROIs are then used to guide architectural simulations. However, this approach relies on the assumption that the execution of ROIs can be reproduced precisely during the simulation as they were during profiling.

7.3.2 ROI Specification

The evaluation using ROIperf considers program repeatability to ensure ROIs remain representative. However, single-threaded programs can often exhibit non-repeatable behavior [77]. One of the main reasons for this behavior is the differences in the microarchitecture that are used for profiling and performance measurements. Other reasons include changes in shared library versions and memory allocation patterns (load and stack locations). To address this and maintain ROI validity, ROIperf enforces identical shared libraries and memory allocation during measurement as observed during profiling. For example, the loading addresses of the shared libraries and the starting address of their stacks should remain the same. On Linux, this can be achieved by temporarily disabling Address Space Layout Randomization (ASLR).

A single-threaded ROI can be simply represented by the retired instruction count at the beginning and the end of the region. As long as regions are repeatable, ensured by using fixed shared libraries and by turning off ASLR, capturing hardware performance counter values at ROI boundaries is sufficient for performance projection (Figure 7.3) of single-threaded programs. For single-threaded programs, an ROI can be represented by the retired instruction count at the beginning and end of the region. Assuming repeatable program execution (achieved through fixed shared libraries and similar microarchitecture), capturing hardware performance counter values at ROI boundaries suffices for

This can typically be done globally by modifying `/proc/sys/kernel/randomize_va_space`, or on a per-process basis by prepending the command line with `setarch x86_64 --addr-no-randomize`.

performance measurements (as shown in Figure 7.3). For multi-threaded programs, a major source of non-repeatability is the timing and behavior of thread synchronization [31, 73, 77]. Instruction counts are, therefore, not a reliable way to specify ROI boundaries. The LoopPoint methodology [8] guarantees ROI repeatability by selecting units of work that begin and end at *worker* loop entries to avoid synchronization overhead and ensure consistent behavior across executions. LoopPoint defines ROIs using pairs of (PC, count), where PC represents the program counter address of the corresponding worker loop entry and count signifies the number of loop iterations. This approach ensures the invariant nature of worker loops to establish reliable ROI boundaries across executions.

7.3.3 ROI Handling in ROIperf

ROIperf aims to capture relevant hardware performance counter values at the boundaries of each region of interest (ROI). It achieves this by leveraging the Linux function `perf_event_open()` to program specific hardware performance counters. The required performance counters can be specified through an environment variable *ROIPERF_LIST*. This variable expects a comma-separated list of number pairs in the format *perftype:counter*. Here, *perftype* indicates the counter type (0 for hardware, 1 for software). The specific counter selection is based on values defined within the Linux header file `/usr/include/linux/perf_event.h`. For example, the *perftype:counter* pair `0:0` corresponds to *hw_cpu_cycles* (hardware counter for CPU cycles), while `1:2` refers to *sw_page_faults* (software counter for page faults).

ROIperf operates on an application along with its designated ROIs, as detailed in Figure 7.2. ROIperf utilizes two primary methods to program hardware performance counters: (a) sampled counting of retired instructions or program counters and (b) continuous monitoring of performance counters specified with *ROIPERF_LIST*. The sampled counting is programmed using an overflow value and a callback function. When using

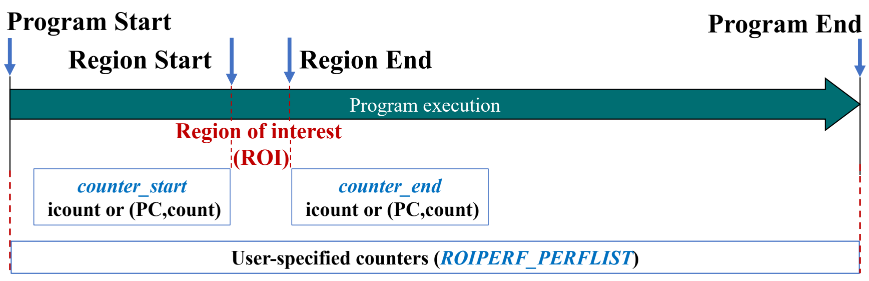


Figure 7.3: The high-level execution flow of an application using the ROIperf tool. Upon program start, user-defined performance counters are initialized. Measurements are then activated at the start of ROI and remain active until the end of ROI. Hardware instruction counts or address (PC) counts are employed to identify the ROI.

instruction count-based ROIs, two counters monitor user-mode *PERF_COUNT_HW_-INSTRUCTIONS*, with overflow values set to the start and end instruction counts of the ROI. Upon overflow, the callback function triggers, capturing the current system-wide time using the Read Time-Stamp Counter (RDTSC) and the values of all the performance counters programmed for continuous monitoring (defined by the *ROIPERF_LIST* environment variable). This continuous monitoring mode allows tracking performance counters specified in *ROIPERF_LIST* alongside sampled counting. An illustration of these various actions taken by ROIperf can be found in Figure 7.3.

For ROIs defined by program counter (`PC`) and `count` values, a different approach is employed. Here, two counters with perftype set to *PERF_TYPE_BREAKPOINT* target the start and end PCs of the ROI. The overflow values are set to the corresponding `count` values specified for the ROI boundaries. Similar to sampled counting, the callback function upon overflow outputs RDTSC values and the values of performance counters from *ROIPERF_LIST*.

Our experiments revealed a significant performance difference between the techniques for programming performance counters used in ROIperf. While *PERF_COUNT_HW_-INSTRUCTIONS* with overflow handling proved highly efficient across all tested x86 processors, *PERF_TYPE_BREAKPOINT* exhibited higher overhead. This overhead derives from the operating system trapping into the kernel on every execution of the programmed PC to check for overflow using a software counter. This frequent trapping can significantly perturb performance measurements, especially for ROIs with frequently executed PCs.

To address this trade-off, we propose a hybrid approach for ROI specification. We recommend using *PERF_TYPE_BREAKPOINT* only for the ROI start, leveraging its precise triggering mechanism. For the ROI end, we suggest employing a relative instruction count-based *PERF_COUNT_HW_INSTRUCTIONS* approach. This combination prioritizes a precise start point while achieving faster monitoring for the ROI end (albeit slightly imprecise, particularly for multi-threaded scenarios). Since ROIperf ultimately focuses on the performance measurements between the start and end of the ROI, this approach offers an acceptable solution.

ROIperf exhibits limitations when dealing with multi-threaded programs, as it focuses on monitoring only the main thread (thread 0). Hence ROIperf starts hardware performance counters for the core/processor where the main thread is running. Pin in probe mode cannot monitor thread creation events. Consequently, there is no callback to ROIperf when child threads are spawned during program execution. Therefore ROIperf cannot monitor any children threads in a multi-threaded program. While ROIperf cannot directly monitor child threads, the captured RDTSC values still reflect the total execution time for the entire ROI, including the work done by child threads. This approach hinges on the assumption that the main thread remains active throughout the ROIs, which means the counters specified using *ROIPERF_PERFLIST* will be counted for the core/processor where the main thread is active.

7.4 Experimental Setup

7.4.1 Workloads Used

We use two benchmarks for our evaluation, SPEC CPU2017 and NAS Parallel Benchmarks (NPB). For our single-threaded evaluations, we use the *rate* version of SPEC CPU2017 benchmarks using training (train) inputs and reference (ref) inputs. For our multi-threaded evaluations, we use the multi-threaded subset of SPEC CPU2017 benchmarks (*speed* version). These benchmarks can spawn several threads that synchronize and share memory. We configure the benchmarks with eight OpenMP threads. We also use NPB version 3.3 (OpenMP-based) for our multi-threaded evaluations that are configured to Class C inputs with eight threads. We present the evaluation results for all but `dc` (data cube) benchmark in the NPB benchmark suite as it generates a huge amount of data. We use *active* thread wait-policy for evaluating the SPEC CPU2017 benchmarks, which means that the threads spin (user-level) at the synchronization point, whereas *passive* policy is used for the NPB benchmarks for which the threads go to sleep while waiting for the other threads at a synchronization point.

7.4.2 Sample Selection

For single-threaded sampling, we use PinPlay-based profiling methodologies involving the PinPoint [24] tool, derived from the SimPoint [20] methodology. We split the application every 200 million instructions. We also use a maxk of 50 for k-means clustering. For sampling multi-threaded applications that use eight threads, we use the LoopPoint methodology [149] with default settings. We split the applications targeting multi-threaded regions of size 800 million global (all-threads) instructions, always aligning with a loop entry. The regions are represented as basic block vectors (BBVs), clustered using k-means clustering with a maxk of 50. PinPlay processing, especially logging, is quite expensive, and therefore, running region selection in a controlled environment was not practical. Instead, region selection was done on machines with varying microarchitectures and run-time libraries. But, in an ideal case, we are required to (a) run all the experiments (region selection, simulation, ROIperf validation, etc.) on the same microarchitecture and (b) package and reuse the system libraries so that we are sure we control the simulation. For ROIperf-based evaluations, we chose two machines with Broadwell and Skylake microarchitectures.

7.4.3 Simulators Used

We compare the effectiveness of ROIperf in sample validation against performance evaluation using simulators. For our experiments with the SPEC CPU2017 benchmarks, we use an in-house simulator derived from Sniper [14], called CoreSim, for evaluations. CoreSim allows for rapid yet fairly accurate simulation of x86 many-core systems that use Intel SDE [227] as the simulation front-end. We configured CoreSim to simulate both Intel Skylake [157] and Intel Cascade Lake [220] microarchitectures. We also use the Sniper multi-core simulator [14, 252] version 8.0 (using Pin [116] front-end) for our evaluations with NPB benchmarks. We configured Sniper to simulate the Intel Gainestown microarchitecture.

7.5 Evaluation

In this section, we aim to demonstrate the effectiveness of the ROIperf methodology across different benchmarks.

7.5.1 Testing ROIperf Applicability

As discussed in Section 7.1, the repeatability of application results can be an issue for a number of applications, both single-threaded and multi-threaded. For our evaluations, we selected the applications that were not prone to this issue. We devised a pre-test for the applications for repeatability, which is two-fold:

1. *Do the thread-0 instruction counts from the region selection and ROIperf runs exhibit a close match?*

   The cases where we found a difference of more than 10% were ruled out from ROIperf evaluations. This test works well for single-threaded applications. For multi-threaded programs, where run-to-run variation is expected due to different amounts of synchronization code, the instruction count test may not be adequate.

2. *Are the regions described using (PC, count) specifications executed on the test machine?*

   We tested this with a Pin-based tool to report ROI start and end events based on (PC, count) ROI specification. If the ROIs are not being executed, this implies subtle control flow diversion between the region selection and ROIperf runs. Any cases with a substantial number of ROIs missed were ruled out.

We demonstrate the ROIperf methodology using simulation-based region validation as the base case and compare the prediction errors reported by the simulation to those reported by ROIperf. ROIperf enables quick fine-tuning of sampling parameters for existing sampled simulation techniques like SimPoint or LoopPoint. We perform this study for relatively shorter train input for SPEC CPU2017 runs as the simulation of ref inputs is otherwise not practical.

7.5.2 Evaluation of Single-threaded Applications

We use the *rate* setup from SPEC CPU2017 benchmarks. The binaries used were compiled using GCC to use the AVX vector instructions. The simulator used was, CoreSim, an SDE-based simulator modeling an Intel Skylake processor. ROIperf evaluations were

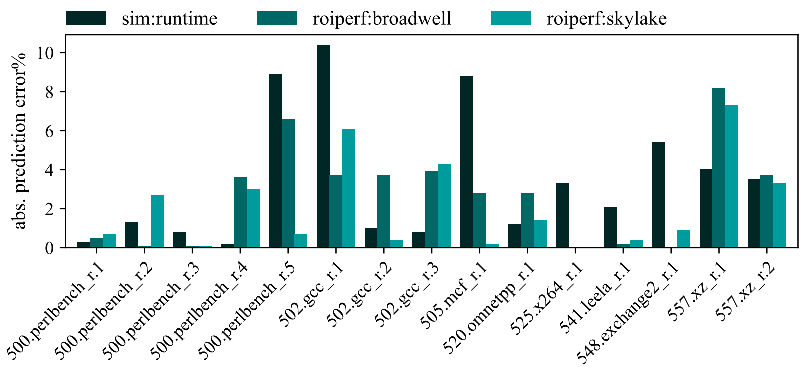


Figure 7.4: Sampling error in predicting cycles-per-instructions (CPI) for single-threaded workloads from the SPEC CPU2017 suite using train inputs. The errors were measured using both a cycle-level simulator and the ROIperf tool running on Broadwell and Skylake hardware platforms.

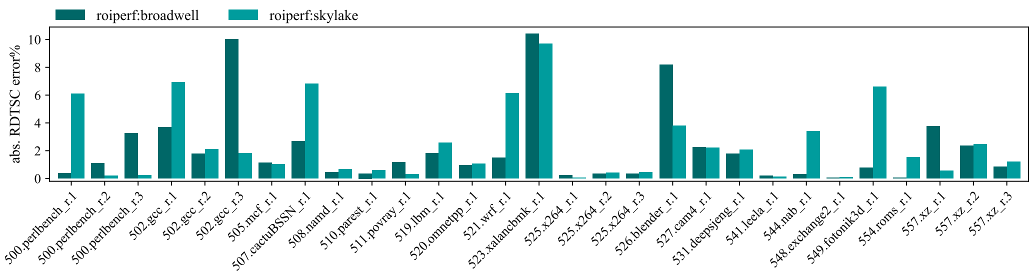


Figure 7.5: Sampling error in predicting the RDTSC values of the single-threaded SPEC benchmarks using ref input.

done on two test machines, one with a Broadwell processor and another with a Skylake processor. The region selection was done using the PinPoints methodology with a slice-size of 200 million instructions and a maximum cluster count (maxk) of 50.

7.5.2.1 SPEC CPU2017 with train input

We first simulated the binaries running train input with CoreSim in two ways: (a) for the entire program execution and (b) once each for each ROI selected by PinPoints (specification based on instruction count). Prediction error for each benchmark was computed using the simulated runtime, full program, and region projected. The longest-running full-program simulation took five weeks to finish. We then used ROIperf using the exact region specification and found prediction errors on two different test machines, one with a Broadwell x86 processor and another with a Skylake x86 processor. We evaluated ROIperf with the full-program and each region and computed prediction error based on cycles-per-instruction (CPI) values reported as shown in Figure 7.2. The measurement was repeated several times, and the average values were considered. The entire evaluation took a few hours, which is a significant improvement over the simulation-based validation methodology. Figure 7.4 reports the prediction errors for simulation and ROIperf-based validation. We see that while the absolute prediction error values differ, the trends in prediction errors are the same between simulation-based and ROIperf-based validation. This gives us confidence in using ROIperf as a much faster alternative to simulation-based ROI validation.

7.5.2.2 SPEC CPU2017 with ref input

SPEC CPU2017 runs with *ref* input are much longer running compared to train input runs. Simulation-based validation for ref input is therefore not practical as it would take a number of months to finish full-program ref runs simulations with CoreSim. This is where ROIperf-based simulation adds value. Since we are using native hardware as the simulator, the evaluation times are much shorter. Figure 7.5 reports the prediction errors for ROIperf-based validation of SPEC CPU2017 ref input runs on running Broadwell and Skylake servers. ROIperf applicability testing (Section 7.5.1) revealed a significant variation (>15%) in the instruction count between the native run on the test machine and the profiling run. On the Skylake machine, all runs of `503.bwaves_r` showed more than 50% difference between instruction count during profiling and during ROIperf run. We observed the Skylake machine happened to have a different version of the math library than the Broadwell machine, and the code executed on the two machines was quite different, as measured by the instruction mixes on both machines. Security updates often add input checks that can lead to significant slowdowns. For example, the *libm* library on the Broadwell machine uses an optimization that removed the canonical input check from `pow()`, which led to a 2.4× reduction in the instruction count for `503.bwaves_r`.

7.5.3 Evaluation of Multi-threaded Applications

Evaluating synchronizing multi-threaded applications can be quite challenging [31]. Tools like PinPlay [3] offer deterministic analysis of multi-threaded applications. While using ROIperf, we estimate the performance using native hardware. For multi-threaded evaluation, we used the OpenMP subset of the speed version of SPEC CPU2017 benchmarks. The regions of interest (ROIs) of the benchmarks were selected using LoopPoint methodology using the settings as described in Section 7.4.2.

7.5.3.1 SPEC CPU2017 with train input

Figure 7.6 shows a comparison between the RDTSC prediction error using ROIperf and runtime prediction error using CoreSim. The ROIs were simulated on CoreSim with Cascade Lake microarchitecture specifications. We use 8-threaded SPEC CPU2017 benchmarks that use train inputs for this evaluation. The benchmarks use active thread wait policy. We can observe very similar trends in the estimation errors, especially for applications like `627.cam4_s.1`.

7.5.3.2 NPB using Class C inputs

We repeat the comparison of prediction errors from ROIperf and simulation for NAS Parallel Benchmarks (NPB) Class C input size. Figure 7.7 shows the runtime prediction errors obtained from simulation (Sniper:Gainestown), and prediction errors for user-level hardware CPU cycles and RDTSC using ROIperf. Again the error bars show similar

The results shown here have been filtered to exclude these specific cases.

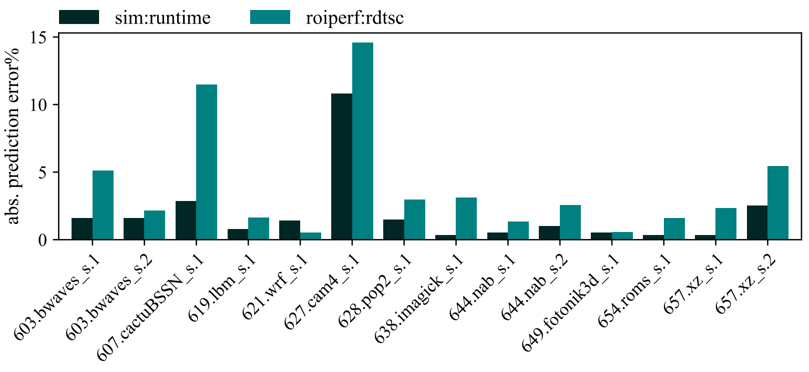


Figure 7.6: A comparison of RDTSC estimation error using ROIperf and runtime estimation error using CoreSim simulator. The benchmark suite is SPEC CPU2017, and the benchmarks use 8 threads, train inputs, and active wait policy. The ROIs are identified using LoopPoint methodology.

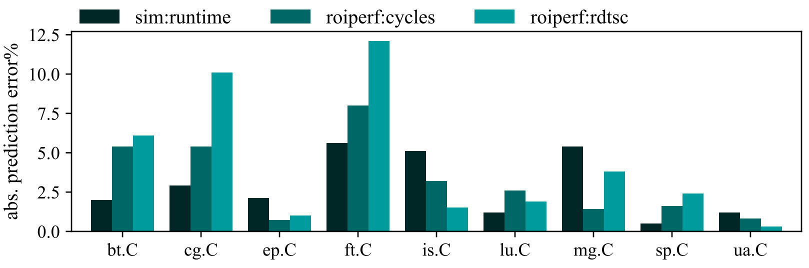


Figure 7.7: A comparison of simulation-based prediction errors with ROIperf results for both HW_CPU_CYCLES and RDTSC projections on a Skylake Server. We use NPB benchmarks that use Class C inputs, 8 threads and passive wait policy.

trends which signify the reliability of the results obtained using ROIperf.

7.6 Related Work

The overhead of the Linux perf_event counter interface that ROIperf uses is described in prior works [253]. ROIperf uses the self-monitoring interface as described earlier and hence is prone to various overheads, namely overheads for performance counter starting, reading, reading multiple times, and stopping. The paper suggests turning off dynamic frequency scaling to avoid affecting the RDTSC instruction results. We did that for our test machines. They also suggest using static linking to avoid dynamic link overhead of the `read()` system call used to read performance counters.

In PinPoints [24], simulation regions selected for SPEC2000 Itanium programs using SimPoint [20] were validated against hardware performance metrics. For evaluation, a JIT-mode Pin tool was used to run until the start of the ROI, determined by instruction count. After detaching from the application, the remaining execution was profiled using a performance monitoring Linux tool, sampling hardware performance counters at slice-size intervals. Unlike PinPoints, ROIperf does not detach from the application. This allows for more precise ROI boundary monitoring, especially when using (PC, count) specifications. ROIperf also supports variable-sized ROIs for both single-threaded and multi-threaded programs.

7.7 Conclusion

We introduce ROIperf, a technique to validate workload sampling methodologies. ROIperf leverages hardware performance counters to validate the representativeness of chosen samples, which can be used in several ways to study the workload characteristics, core interactions, cache behavior, etc., without requiring a simulator. ROIperf facilitates the validation of workload sampling methodologies for large-scale workloads like the SPEC CPU2017 benchmarks with reference inputs. Our analyses show that the reproducibility of program behavior across multiple executions is a prerequisite for obtaining stable measurements. Simulators provide a controlled environment for performance estimation and, especially in the case of multi-threaded applications, control the thread progress. ROIperf could be extended to support compatibility beyond specific hardware platforms, particularly towards the increasingly heterogeneous nature of modern applications.

Chapter 8

Conclusion and Future Work

In this chapter, we present a summary of the contributions of this thesis and lay out the potential directions for future work.

8.1 Conclusion

There's no real ending. It's just the place where you stop the story.

— Frank Herbert

The thesis explores novel techniques for efficiently evaluating the performance of post-Dennard systems using sampling. We first proposed LoopPoint, a sampled simulation methodology that significantly reduces the simulation time of large general-purpose multi-threaded workloads. LoopPoint methodology is integrated with widely used microarchitectural simulators like gem5 and Sniper. A follow-up work, Viper, enhanced the accuracy and speed of LoopPoint by considering the hierarchical structure of program execution. LoopPoint is effective for microarchitecture-level simulations, while Viper is suitable for finer granularity in RTL-level simulations. Both these methodologies are applicable only to static workloads. Existing sampled simulation methodologies are considered insufficient for assessing the dynamic nature of evolving architectures. These architectures integrate several runtime optimization techniques at both hardware and software levels to enhance system performance. We proposed Pac-Sim, which is designed for dynamically optimized software and hardware by performing region selection and analysis online. Pac-Sim accurately evaluates dynamically scheduled multi-threaded applications, accounting for runtime performance variability. The growing need for high-performance computing (HPC) and artificial intelligence (AI) has driven the adoption of heterogeneous computing systems that integrate diverse processing cores like CPUs and GPUs. However, evaluating the performance of these systems remains a significant challenge, often requiring substantial time and resources. To address this, we introduce XPU-Point, a novel methodology designed to identify representative regions within heterogeneous CPU-GPU applications. While workload sampling techniques identify regions of interest within applications, their performance is typically validated using simulations, which can still be time-consuming. However, validating the performance of a selected sample against the full application is crucial. To address this, we proposed ROIperf, which leverages native hardware performance counters, providing a quick and accurate method to validate the representativeness of the regions of interest selected for long-running workloads.

8.2 Future Work

We can only see a short distance ahead, but we can see plenty there that needs to be done.

— Alan Turing

In this thesis, we addressed the significant challenge that arises due to the performance disparity between hardware and its corresponding simulators. This bottleneck significantly hinders the design space exploration for increasingly complex systems.

While workload sampling, explored in this thesis, offers a solution by focusing on representative subsets of the workload, it represents just one direction for further research. Future investigations to explore complementary techniques for faster simulation include hardware emulation techniques, GPU-based simulations, analytical models, etc. We outline a few such directions below:

1. The characterization of emerging real-world workloads presents a significant challenge. With the increasing complexity of these workloads, like those found in mobile and data center environments, the methodologies proposed in this thesis may be inadequate. While this thesis highlights the potential of Pac-Sim in evaluating compute-intensive general-purpose workloads, its applicability to other domains, such as cloud or mobile workload classes, is a promising area of research. Current approaches to GPU and heterogeneous CPU-GPU sampling, while effective in controlled settings, exhibit limitations when dealing with real-world scenarios, such as over-subscribed GPUs, dynamic kernels, and multi-GPU configurations. Leveraging neural networks for program phase identification presents an alternative for BBVs or other signature vectors. Their ability to learn from program structure and runtime states allows for the effective characterization of application regions independent of the underlying ISA.

2. Accelerating cycle-accurate simulations remains a crucial area of research. The traditional simulation takes an extremely long time by simulating the entire workload on the CPU alone. One approach to speed up simulation involves leveraging profiling tools to understand the control flow of the workload, which can be used to optimize the datapath and parallelize the simulation of independent components across GPUs. In a similar direction, simulating CPU-GPU workloads can be accelerated by offloading the GPU kernel simulation onto actual GPUs. However, this approach may necessitate frequent CPU-GPU synchronization. Additionally, com-

bining simulators with analytical models of traditional microarchitectural structures offers another potential for simulation speedups.

3. While sampled simulation addresses the problem of long simulation times, this introduces the challenge of warming up the microarchitectural state. Traditional CPU warmup techniques, such as statistical warming and checkpointing, are unavailable for GPU systems and heterogeneous CPU-GPU systems, whereas functional warming would incur a significant amount of time to be spent on simulations just for microarchitectural state reconstruction. Leveraging machine learning techniques on statistical profiles of relevant memory access patterns, branch predictor behavior, and prefetcher accesses could enable the development of sophisticated microarchitectural warmup methods.

4. While application checkpointing and deterministic replay have been established for CPU workloads, extending these techniques to heterogeneous CPU-GPU environments presents a significant research opportunity. For CPU workloads, ELFies offer a widely adopted solution for capturing application state through executable checkpoints. However, ELFies do not capture the system state, and system-level checkpointing techniques using QEMU are essential to capture the entire system state for accurate full-system simulation.

He thought he kept the universe alone;

For all the voice in answer he could wake

Was but the mocking echo of his own

From some tree-hidden cliff across the lake.

Some morning from the boulder-broken beach

He would cry out on life, that what it wants

Is not its own love back in copy speech,

But counter-love, original response.

Robert Frost

Bibliography

[1] SPEC CPU®2017 documentation index. `http://spec.org/cpu2017/Docs/index.html`.

[2] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *International Symposium on Computer Architecture (ISCA)*, pages 84–97, June 2003.

[3] Harish Patil and Trevor E. Carlson. Pinballs: portable and shareable user-level checkpoints for reproducible analysis and simulation. In *Workshop on Reproducible Research Methodologies (REPRODUCE)*, February 2014.

[4] Ferenc Bodon and Lajos Rónyai. Trie: an alternative data structure for data mining algorithms. *Mathematical and Computer Modelling*, 38(7-9):739–751, 2003.

[5] Kenneth C Barr, Heidi Pan, Michael Zhang, and Krste Asanovic. Accelerating multiprocessor simulation with a memory timestamp record. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 66–77, March 2005.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D.

Hill, and David A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, August 2011.

[7] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *International Symposium on Computer Architecture (ISCA)*, pages 473–486. IEEE, 2020.

[8] Alen Sabu, Harish Patil, Wim Heirman, and Trevor E. Carlson. LoopPoint: Checkpoint-driven sampled simulation for multi-threaded applications. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 604–618, 2022.

[9] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 2006.

[10] Shekhar Borkar. Thousand core chips: a technology perspective. In *Design Automation Conference (DAC)*, pages 746–749, 2007.

[11] Anant Agarwal and Markus Levy. The kill rule for multicore. In *Proceedings of the 44th annual Design Automation Conference*, pages 750–753, 2007.

[12] Eric S Chung, Peter A Milder, James C Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *International Symposium on Microarchitecture (MICRO)*, pages 225–236. IEEE, 2010.

[13] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. Theres plenty of room at the top: What will drive computer performance after moores law? *Science*, 368(6495):eaam9744, 2020.

[14] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *International*

Conference for High Performance Computing, Networking, Storage and Analysis (SC), pages 52:1–52:12, November 2011.

[15] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 1974.

[16] Changxi Liu, Yifan Sun, and Trevor E. Carlson. Photon: A fine-grained sampled simulation methodology for gpu workloads. In *International Symposium on Microarchitecture (MICRO)*, page 12271241, 2023.

[17] Murali Annavaram, Ryan Rakvic, Marzia Polito, J-Y Bouguet, Richard Hankins, and Bob Davies. The fuzzy correlation between code and performance predictability. In *International Symposium on Microarchitecture (MICRO)*, pages 93–104, 2004.

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

[19] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, 2001.

[20] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, October 2002.

[21] Erez Perelman, Greg Hamerly, and Brad Calder. Picking statistically valid and early simulation points. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 244–255, 2003.

[22] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and pre-

diction. In *International Symposium on Computer Architecture (ISCA)*, pages 336–349, 2003.

[23] Greg Hamerly, Erez Perelman, and Brad Calder. How to use SimPoint to pick simulation points. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):25–30, March 2004.

[24] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *International Symposium on Microarchitecture (MICRO)*, pages 81–92, December 2004.

[25] Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood, and Brad Calder. Motivation for variable length intervals and hierarchical phase behavior. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 135–146, March 2005.

[26] Jeremy Lau, Erez Perelman, and Brad Calder. Selecting software phase markers with code structure analysis. In *International Symposium on Code Generation and Optimization (CGO)*, pages 135–146, March 2006.

[27] Sina Hassani, Gabriel Southern, and Jose Renau. LiveSim: Going live with microarchitecture simulation. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 606–617, March 2016.

[28] M. Van Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software, 2004*, March 2004.

[29] Thomas F. Wenisch, Roland E. Wunderlich, Babak Falsafi, and James C. Hoe. TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. ACM, 2005.

[30] 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*, 26(4):18–31, 2006.

[31] A.R. Alameldeen and D.A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, 2006.

[32] T. E. Carlson, W. Heirman, and L. Eeckhout. Sampled simulation of multi-threaded applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, April 2013.

[33] E. K. Ardestani and J. Renau. ESESC: A fast multicore simulator using time-based sampling. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 448–459, February 2013.

[34] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout. BarrierPoint: Sampled simulation of multi-threaded applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, March 2014.

[35] Stijn Eyerman and Lieven Eeckhout. Fine-grained dvfs using on-chip regulators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(1):1–24, 2011.

[36] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *International Symposium on Microarchitecture (MICRO)*, pages 347–358, 2006.

[37] Wonyoung Kim, Meeta S Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 123–134, 2008.

[38] Sparsh Mittal, Zhao Zhang, and Jeffrey S Vetter. Flexiway: A cache energy saving

technique using fine-grained cache reconfiguration. In *International conference on computer design (ICCD)*, pages 100–107. IEEE, 2013.

[39] Sparsh Mittal, Yanan Cao, and Zhao Zhang. Master: A multicore cache energy-saving technique using dynamic cache reconfiguration. *IEEE Transactions on very large scale integration (VLSI) systems*, 22(8):1653–1665, 2013.

[40] D.H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *International Symposium on Microarchitecture (MICRO)*, pages 248–259, 1999.

[41] James Charles, Preet Jassi, Narayan S Ananth, Abbas Sadat, and Alexandra Fedorova. Evaluation of the intel® core i7 turbo boost feature. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 188–197. IEEE, 2009.

[42] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters*, 21(02):173–193, 2011.

[43] T. Grass, A. Rico, M. Casas, M. Moreto, and E. Ayguadé. TaskPoint: Sampled simulation of task-based programs. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 296–306, April 2016.

[44] T. Grass, T. E. Carlson, A. Rico, G. Ceballos, E. Ayguadé, M. Casas, and M. Moreto. Sampled simulation of task-based programs. *Transactions on Computers (TC)*, 68(2):255–269, 2019.

[45] Jen-Cheng Huang, Lifeng Nai, Hyesoon Kim, and Hsien-Hsin S Lee. Tbpoint: Reducing simulation time for large-scale gpgpu kernels. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 437–446. IEEE, 2014.

[46] Cesar Avalos Baddouh, Mahmoud Khairy, Roland N Green, Mathias Payer, and Timothy G Rogers. Principal kernel analysis: A tractable methodology to simulate scaled gpu workloads. In *International Symposium on Microarchitecture (MICRO)*, pages 724–737, 2021.

[47] Harish Patil, Alexander Isaev, Wim Heirman, Alen Sabu, Ali Hajiabadi, and

Trevor E Carlson. ELFies: Executable region checkpoints for performance analysis and simulation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 126–136, February/March 2021.

[48] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, et al. Towards developing high performance risc-v processors using agile methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1178–1199. IEEE, 2022.

[49] W Snyder. Verilator: the fast free verilog simulator. *URL: http://www. veripool. org*, 2012.

[50] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *International Symposium on Computer Architecture (ISCA)*, pages 29–42, June 2018.

[51] Jeremy Lau, Stefan Schoemackers, and Brad Calder. Structures for phase classification. In *IEEE International Symposium on-ISPASS Performance Analysis of Systems and Software, 2004*, pages 57–67. IEEE, 2004.

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

[53] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.

[54] TOP500. Top500 supercomputer sites. `https://www.top500.org/`, 2022. Accessed on November 16, 2022.

[55] Cen Chen, Kenli Li, Aijia Ouyang, Zhuo Tang, and Keqin Li. Gpu-accelerated parallel hierarchical extreme learning machine on flink for big data. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47(10):2740–2753, 2017.

[56] Hai Jiang, Yi Chen, Zhi Qiao, Tien-Hsiung Weng, and Kuan-Ching Li. Scaling up mapreduce-based big data processing on multi-gpu systems. *Cluster Computing*, 18:369–383, 2015.

[57] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *Symposium on Operating Systems Design and Implementation (OSDI)*, volume 16, pages 265–283. Savannah, GA, USA, 2016.

[58] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 578–594, 2018.

[59] Melanie Kambadur, Sunpyo Hong, Juan Cabral, Harish Patil, Chi-Keung Luk, Sohaib Sajid, and Martha A Kim. Fast computational gpu design with gt-pin. In *2015 IEEE International Symposium on Workload Characterization*, pages 76–86. IEEE, 2015.

[60] Mahmood Naderan-Tahan, Hossein SeyyedAghaei, and Lieven Eeckhout. Sieve: Stratified gpu-compute workload sampling. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 224–234. IEEE, 2023.

[61] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th annual*

international symposium on Computer architecture, pages 451–460, 2010.

[62] Humayun Khalid. Validating trace-driven microarchitectural simulations. *IEEE Micro*, 20(6):76–82, 2000.

[63] Qinzhe Wu, Steven Flolid, Shuang Song, Junyong Deng, and Lizy K John. Invited paper for the hot workloads special session hot regions in spec cpu2017. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 71–77. IEEE, 2018.

[64] Haiyang Han and Nikos Hardavellas. Public release and validation of spec cpu2017 pinpoints. *arXiv preprint arXiv:2112.06981*, 2021.

[65] Rajat Todi. Speclite: using representative samples to reduce spec cpu2000 workload. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization (WWC-4)*, pages 15–23. IEEE, 2001.

[66] Performance monitoring in the intel 64 and ia-32 architectures software developers manual, volume 3b. `https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html`.

[67] Arun A Nair and Lizy K John. Simulation points for spec cpu 2006. In *2008 IEEE International Conference on Computer Design*, pages 397–403. IEEE, 2008.

[68] Q. Wu, S. Flolid, S. Song, J. Deng, and L. K. John. Invited paper for the hot workloads special session hot regions in SPEC CPU2017. In *International Symposium on Workload Characterization (IISWC)*, pages 71–77, 2018.

[69] Reena Panda, Shuang Song, Joseph Dean, and Lizy K John. Wait of a decade: Did spec cpu 2017 broaden the performance horizon? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282. IEEE, 2018.

[70] Kenneth Hoste and Lieven Eeckhout. Microarchitecture-independent workload characterization. *IEEE micro*, 27(3):63–72, 2007.

[71] Yakun Sophia Shao and David Brooks. Isa-independent workload characterization and its implications for specialized architectures. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 245–255. IEEE, 2013.

[72] Alaa R Alameldeen, Carl J Mauer, Min Xu, Pacia J Harper, Milo MK Martin, Daniel J Sorin, Mark D Hill, and David A Wood. Evaluating non-deterministic multi-threaded commercial workloads. In *Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, February 2002.

[73] A.R. Alameldeen and D.A. Wood. Variability in architectural simulations of multi-threaded workloads. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 7–18, February 2003.

[74] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[75] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. *ACM SIGPLAN Notices*, 39(11):165–176, 2004.

[76] Edward W Forgy. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *biometrics*, 21:768–769, 1965.

[77] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *International Symposium on Code Generation and Optimization (CGO)*, pages 2–11, April 2010.

[78] Andreas Sandberg, Nikos Nikoleris, Trevor E. Carlson, Erik Hagersten, Stefanos Kaxiras, and David Black-Schaffer. Full speed ahead: Detailed architectural simulation at near-native speed. In *2015 IEEE International Symposium on Workload Characterization*, pages 183–192, 2015.

[79] M. Ekman and P. Stenstrom. Enhancing multiprocessor architecture simulation

speed using matched-pair comparison. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 89–99, March 2005.

[80] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. Cotson: infrastructure for full system simulation. *ACM SIGOPS Operating Systems Review*, 43(1):52–61, 2009.

[81] Alex Skaletsky, Konstantin Levit-Gurevich, Michael Berezalsky, Yulia Kuznetcova, and Hila Yakov. Flexible binary instrumentation framework to profile code running on intel gpus. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 109–120. IEEE, 2022.

[82] Zhibin Yu, Lieven Eeckhout, Nilanjan Goswami, Tao Li, Lizy John, Hai Jin, and Chengzhong Xu. Accelerating gpgpu architecture simulation. In *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems*, pages 331–332, 2013.

[83] Zhibin Yu, Lieven Eeckhout, Nilanjan Goswami, Tao Li, Lizy K John, Hai Jin, Chengzhong Xu, and Junmin Wu. Gpgpu-minibench: accelerating gpgpu microarchitecture simulation. *IEEE Transactions on Computers*, 64(11):3153–3166, 2015.

[84] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems (TOCS)*, 27(2):1–37, 2009.

[85] S. De Pestel, S. Van den Steen, S. Akram, and L. Eeckhout. RPPM: Rapid performance prediction of multithreaded workloads on multicore processors. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 257–267, March 2019.

[86] David Eklov and Erik Hagersten. Statstack: Efficient modeling of lru caches. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 55–65. IEEE, 2010.

[87] Sander De Pestel, Stijn Eyerman, and Lieven Eeckhout. Micro-architecture independent branch behavior characterization. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 135–144. IEEE, 2015.

[88] John W Haskins and Kevin Skadron. Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation. In *2003 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS 2003.*, pages 195–203. IEEE, 2003.

[89] Lieven Eeckhout, Yue Luo, Koen De Bosschere, and Lizy K John. Blrl: Accurate and efficient warmup for sampled processor simulation. *The Computer Journal*, 48(4):451–459, 2005.

[90] Nikos Nikoleris, Lieven Eeckhout, Erik Hagersten, and Trevor E. Carlson. Directed statistical warming through time traveling. In *International Symposium on Microarchitecture (MICRO)*, pages 1037–1049, October 2019.

[91] Nikos Nikoleris, Andreas Sandberg, Erik Hagersten, and Trevor E Carlson. Coolsim: Statistical techniques to replace cache warming with efficient, virtualized profiling. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 106–115. IEEE, 2016.

[92] Michael Van Biesbrouck, Brad Calder, and Lieven Eeckhout. Efficient sampling startup for simpoint. *IEEE Micro*, 26(4):32–42, 2006.

[93] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *International Symposium on Computer Architecture (ISCA)*, pages 475–486, June 2013.

[94] I Synopsys. Vcs–functional verification solution, 2014.

[95] Prasun Gera, Hyojong Kim, Hyesoon Kim, Sunpyo Hong, Vinod George, and Chi-Keung Luk. Performance characterisation and simulation of intel's integrated gpu

architecture. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 139–148. IEEE, 2018.

[96] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: A simulation framework for cpu-gpu computing. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 335–344. IEEE, 2012.

[97] Jason Power, Joel Hestness, Marc S Orr, Mark D Hill, and David A Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, 2014.

[98] Yifan Sun, Trinayan Baruah, Saiful A Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, et al. Mgpusim: enabling multi-gpu performance modeling and optimization. In *International Symposium on Computer Architecture (ISCA)*, pages 197–209, 2019.

[99] Bradford M Beckmann and Anthony Gutierrez. The amd gem5 apu simulator: Modeling heterogeneous systems in gem5. In *Tutorial at the International Symposium on Microarchitecture (MICRO)*, 2015.

[100] Anthony Gutierrez, Bradford M Beckmann, Alexandru Dutu, Joseph Gross, Michael LeBeane, John Kalamatianos, Onur Kayiran, Matthew Poremba, Brandon Potter, Sooraj Puthoor, et al. Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 608–619. IEEE, 2018.

[101] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE international symposium on performance analysis of systems and software*, pages 163–174. IEEE, 2009.

[102] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladrish Chatterjee, Nan Jiang, and David Nellans. Need for speed: Experiences building a

trustworthy system-level gpu simulator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 868–880. IEEE, 2021.

[103] Karthik Ganesan and Lizy K John. Maximum multicore power (mampo) an automatic multithreaded synthetic power virus generation framework for multicore systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[104] Siddharth Nilakantan, Karthik Sangaiah, Ankit More, Giordano Salvadory, Baris Taskin, and Mark Hempstead. Synchrotrace: synchronization-aware architecture-agnostic traces for light-weight multicore simulation. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 278–287. IEEE, 2015.

[105] Reena Panda, Xinnian Zheng, Jiajun Wang, Andreas Gerstlauer, and Lizy K John. Statistical pattern based modeling of gpu memory access streams. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[106] Mingyu Liang, Wenyin Fu, Louis Feng, Zhongyi Lin, Pavani Panakanti, Shengbao Zheng, Srinivas Sridharan, and Christina Delimitrou. Mystique: Enabling accurate and scalable generation of production ai benchmarks. In *International Symposium on Computer Architecture (ISCA)*, pages 1–13, 2023.

[107] Mingyu Liang, Yu Gan, Yueying Li, Carlos Torres, Abhishek Dhanotia, Mahesh Ketkar, and Christina Delimitrou. Ditto: End-to-end application cloning for networked cloud services. In *International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 222–236, 2023.

[108] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, page 41, 2005.

[109] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[110] A. J. KleinOsowski and D. J. Lilja. Minnespec: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters (CAL)*, 1(1):7–7, 2002.

[111] Robert H. Bell and Lizy K. John. Improved automatic testcase synthesis for performance model validation. In *International Conference on Supercomputing (SC)*, pages 111–120, June 2005.

[112] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical report, NAS-95-020, NASA Ames Research Center, 1995.

[113] Edward W Forgy. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *Biometrics*, 21:768–769, 1965.

[114] Gideon Schwarz. Estimating the dimension of a model. *The Annals of Statistics*, pages 461–464, 1978.

[115] Thomas F Wenisch, Roland E Wunderlich, Babak Falsafi, and James C Hoe. Simulation sampling with live-points. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, March 2006.

[116] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, June 2005.

[117] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-generation compute benchmark. In *International Conference on Performance Engineering (ICPE)*, pages 41–42, April 2018.

[118] Ankur Limaye and Tosiron Adegbija. A workload characterization of the SPEC CPU2017 benchmark suite. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 190–200, April 2018.

[119] E Barszcz, J Barton, L Dagum, P Frederickson, T Lasinski, R Schreiber, V Venkatakrishnan, S Weeratunga, D Bailey, D Browning, et al. The NAS parallel benchmarks. In *International Journal of Supercomputer Applications*, 1991.

[120] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks summary and preliminary results. In *Conference on Supercomputing (SC)*, pages 158–165, 1991.

[121] Haoqiang Jin, Michael Frumkin, and Jerry Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, NAS-99-011, NASA Ames Research Center, October 1999.

[122] DCFG generation with PinPlay. `https://software.intel.com/content/www/us/en/develop/articles/pintool-dcfg.html`.

[123] C. Yount, H. Patil, and M. S. Islam. Graph-matching-based simulation-region selection for multiple binaries. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 52–61, March 2015.

[124] C. Pereira, H. Patil, and B. Calder. Reproducible simulation of multi-threaded workloads for architecture design exploration. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 173–182, September 2008.

[125] OpenMP 3.1 API C/C++ Syntax Quick Reference Card. `https://www.openmp.org/wp-content/uploads/OpenMP3.1-CCard.pdf`.

[126] Tong Li, Alvin R Lebeck, and Daniel J Sorin. Spin detection hardware for improved management of multithreaded systems. *Transactions on Parallel and Distributed Systems (TPDS)*, 17(6):508–521, 2006.

[127] S. Van den Steen, S. Eyerman, S. De Pestel, M. Mechri, T. E. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout. Analytical processor performance and

power modeling using micro-architecture independent characteristics. *Transactions on Computers (TC)*, 65(12):3537–3551, 2016.

[128] S. Van den Steen, S. De Pestel, M. Mechri, S. Eyerman, T. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout. Micro-architecture independent analytical processor performance and power modeling. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 32–41, March 2015.

[129] A. A. Nair and L. K. John. Simulation points for SPEC CPU 2006. In *International Conference on Computer Design (ICCD)*, pages 397–403, October 2008.

[130] Xinnian Zheng, Haris Vikalo, Shuang Song, Lizy K John, and Andreas Gerstlauer. Sampling-based binary-level cross-platform performance estimation. In *DATE*, 2017.

[131] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28, 2005.

[132] Timothy Sherwood and Brad Calder. Time varying behavior of programs. *In UC San Diego*, 1999.

[133] Checkpoint/restore in userspace.

[134] CRIU integration with docker.

[135] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza

Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

[136] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. From RTL to CUDA: A GPU acceleration flow for RTL simulation with batch stimulus. In *International Conference on Parallel Processing (ICPP)*, pages 1–12, 2022.

[137] Fares Elsabbagh, Shabnam Sheikhha, Victor A Ying, Quan M Nguyen, Joel S Emer, and Daniel Sanchez. Accelerating rtl simulation with hardware-software co-design. In *Symposium on Microarchitecture (MICRO23)*, 2023.

[138] Mahyar Emami, Sahand Kashani, Keisuke Kamahori, Mohammad Sepehr Pourghannad, Ritik Raj, and James R Larus. Manticore: Hardware-accelerated RTL simulation with static bulk-synchronous parallelism. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 219–237, 2023.

[139] Zhangxi Tan, Zhenghao Qian, Xi Chen, Krste Asanovic, and David Patterson. DIABLO: A warehouse-scale computer network simulator using FPGAs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 207–221, 2015.

[140] William Lloyd Bircher and Lizy John. Predictive power management for multi-core

processors. In *International Symposium on Computer Architecture*, pages 243–255. Springer, 2010.

[141] Andreas Diavastos and Pedro Trancoso. Switches: A lightweight runtime for dataflow execution of tasks on many-cores. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):1–23, 2017.

[142] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, pages 81–90, 2000.

[143] Neeraj Kulkarni, Feng Qi, and Christina Delimitrou. Pliant: Leveraging approximation to improve datacenter resource efficiency. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 159–171. IEEE, 2019.

[144] Michael J Voss and Rudolf Eigemann. High-level adaptive program optimization with adapt. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 93–102, 2001.

[145] Xin You, Changxi Liu, Hailong Yang, Pengbo Wang, Zhongzhi Luan, and Depei Qian. Vectorizing spmv by exploiting dynamic regular patterns. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–12, 2022.

[146] Chuntao Jiang, Zhibin Yu, Hai Jin, Chengzhong Xu, Lieven Eeckhout, Wim Heirman, Trevor E. Carlson, and Xiaofei Liao. Pcantorsim: Accelerating parallel architecture simulation through fractal-based sampling. *ACM Trans. Archit. Code Optim.*, 10(4), dec 2013.

[147] R.L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[148] Ayose Falcón, Paolo Faraboschi, and Daniel Ortega. Combining simulation and virtualization through dynamic sampling. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 72–83. IEEE, 2007.

[149] LoopPoint source code. `https://github.com/nus-comparch/looppoint`.

[150] Anastasiia Butko, Rafael Garibotti, Luciano Ost, Vianney Lapotre, Abdoulaye Gamatie, Gilles Sassatelli, and Chris Adeniyi-Jones. A trace-driven approach for fast and accurate simulation of manycore architectures. In *The 20th Asia and South Pacific Design Automation Conference*, pages 707–712. IEEE, 2015.

[151] Joshua L Kihm, Samuel D Strom, and Daniel A Connors. Phase-guided small-sample simulation. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 84–93. IEEE, 2007.

[152] Marc Casas, Harald Servat, Rosa M. Badia, and Jesús Labarta. Extracting the optimal sampling frequency of applications spectral analysis. *Concurrency and Computation: Practice and Experience*, 24:237–259, 03 2012.

[153] Sanjoy Dasgupta. Experiments with random projection. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pages 143–151, 2000.

[154] Charles Elkan. Using the triangle inequality to accelerate k-means. In *Proceedings of the 20th international conference on Machine Learning (ICML-03)*, pages 147–153, 2003.

[155] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[156] Trevor E Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):1–25, 2014.

[157] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro*, 37(2):52–62, 2017.

[158] Irma Esmer Papazian. New 3rd gen intelő xeonő scalable processor (codename: Ice lake-sp). In *IEEE Hot Chips Symposium (HCS)*, pages 1–22, 2020.

[159] Dimitrios Chasapis, Marc Casas, Miquel Moretó, Raul Vidal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Parsecss: Evaluating the impact of task parallelism in the parsec benchmark suite. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):1–22, 2015.

[160] Ali Hajiabadi, Andreas Diavastos, and Trevor E. Carlson. Noreba: A compiler-informed non-speculative out-of-order commit processor. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 182193, New York, NY, USA, 2021. Association for Computing Machinery.

[161] Jianping Zeng, Hongjune Kim, Jaejin Lee, and Changhee Jung. Turnpike: Lightweight soft error resilience for in-order cores. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 654666, New York, NY, USA, 2021. Association for Computing Machinery.

[162] Stijn Eyerman, Wim Heirman, Sam Van den Steen, and Ibrahim Hur. Enabling branch-mispredict level parallelism by selectively flushing instructions. In *EEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 767–778, 2021.

[163] M Deilmann et al. A guide to vectorization with intel c++ compilers. *Intel Corporation*, pages 20–21, 2012.

[164] Alen Sabu, Changxi Liu, and Trevor E. Carlson. Viper: Utilizing hierarchical program structure to accelerate multi-core simulation. *IEEE Access*, 12:17669–17678, 2024.

[165] Shoaib Akram, Jennifer B Sartor, and Lieven Eeckhout. Dvfs performance prediction for managed multithreaded applications. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 12–23. IEEE, 2016.

[166] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn,

and Pablo Villalobos. Compute trends across three eras of machine learning. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2022.

[167] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *International Symposium on Computer Architecture*, pages 365–376, 2011.

[168] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[169] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM transactions on graphics (TOG)*, 23(3):777–786, 2004.

[170] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[171] William S. Carter, Ic Duong, R. R. Freman, Henry Hsieh, Jason Y. Ja, John E. Mahoney, N. T. Ngo, and S. L. Sac. A user programmable reconfigurable logic array. In *Proc. Custom Integrated Circuits Conf.*, pages 515–521, 1986.

[172] Vıctor Garcıa, Juan Gomez-Luna, Thomas Grass, Alejandro Rico, Eduard Ayguade, and Antonio J Pena. Evaluating the effect of last-level cache sharing on integrated gpu-cpu systems with heterogeneous applications. In *International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[173] Mark D Hill and Vijay Janapa Reddi. Accelerator-level parallelism. *Communications of the ACM*, 64(12):36–38, 2021.

[174] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[175] Manish Arora, Siddhartha Nath, Subhra Mazumdar, Scott B Baden, and Dean M Tullsen. Redefining the role of the cpu in the era of cpu-gpu integration. *IEEE Micro*, 32(6):4–16, 2012.

[176] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, page 112, New York, NY, USA, 2000. Association for Computing Machinery.

[177] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM workshop on feedback-directed and dynamic optimization (FDDO-4)*, page 20, 2001.

[178] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 372–383, 2019.

[179] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[180] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[181] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[182] Jack Choquette. Nvidia hopper h100 gpu: Scaling performance. *IEEE Micro*, 2023.

[183] Kai Yuan, Christoph Bauinger, Xiangyi Zhang, Pascal Baehr, Matthias Kirch-

hart, Darius Dabert, Adrien Tousnakhoff, Pierre Boudier, and Michael Paulitsch. Fully-fused multi-layer perceptrons on intel data center gpus. *arXiv preprint arXiv:2403.17607*, 2024.

[184] XPU-Point source code. `https://github.com/nus-comparch/xpupoint`, 2025.

[185] Moshe Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. Analyzing parallel programs with pin. *Computer*, 43(3):34–41, 2010.

[186] Linux. Linux programmers manual. `https://man7.org/linux/man-pages/man8/ld.so.8.html`, 2024.

[187] Gordon E Moore et al. Progress in digital integrated electronics. In *Electron devices meeting*, volume 21, pages 11–13. Washington, DC, 1975.

[188] Herb Sutter et al. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs journal*, 30(3):202–210, 2005.

[189] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T Kandemir, Gabriel H Loh, Onur Mutlu, and Chita R Das. Managing gpu concurrency in heterogeneous architectures. In *IEEE/ACM international symposium on microarchitecture*, pages 114–126. IEEE, 2014.

[190] Joel Hestness, Stephen W Keckler, and David A Wood. Gpu computing pipeline inefficiencies and optimization opportunities in heterogeneous cpu-gpu processors. In *2015 IEEE International Symposium on Workload Characterization*, pages 87–97. IEEE, 2015.

[191] Nvidia gh200 grace hopper superchip architecture. `https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper/`, 2023.

[192] Debendra Das Sharma. Compute express link. *CXL Consortium White Paper*, 2019.

[193] Denis Foley and John Danskin. Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17, 2017.

[194] DARPA. Common heterogeneous integration and ip reuse strategies (chips). `https://www.darpa.mil/program/common-heterogeneous-integration-and-ip-reuse-strategies`, 2024.

[195] CP Wong and Michelle M Wong. Recent advances in plastic packaging of flip-chip and multichip modules (mcm) of microelectronics. *IEEE Transactions on Components and Packaging Technologies*, 22(1):21–25, 1999.

[196] Arik Gihon. Lunar lake architecture session. In *2024 IEEE Hot Chips 36 Symposium (HCS)*, pages 1–49. IEEE Computer Society, 2024.

[197] Gabriel H Loh, Michael J Schulte, Mike Ignatowski, Vignesh Adhinarayanan, Shaizeen Aga, Derrick Aguren, Varun Agrawal, Ashwin M Aji, Johnathan Alsop, Paul Bauman, et al. A research retrospective on amd's exascale computing journey. In *International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2023.

[198] Alan Smith, Gabriel H Loh, Michael J Schulte, Mike Ignatowski, Samuel Naffziger, Mike Mantor, Mark Fowler, Nathan Kalyanasundharam, Vamsi Alla, Nicholas Malaya, Joseph L. Greathouse, Eric Chapman, and Raja Swaminathan. Realizing the amd exascale heterogeneous processor vision. In *International Symposium on Computer Architecture (ISCA)*, 2024.

[199] Boris Krasnopolsky and Alexey Medvedev. Acceleration of large scale openfoam simulations on distributed systems with multicore cpus and gpus. In *Parallel Computing: On the Road to Exascale*, pages 93–102. IOS Press, 2016.

[200] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[201] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla:

A unified graphics and computing architecture. *IEEE micro*, 28(2):39–55, 2008.

[202] David Blythe. The xe gpu architecture. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–27. IEEE Computer Society, 2020.

[203] Michael J Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[204] Guei-Yuan Lueh, Kaiyu Chen, Gang Chen, Joel Fuentes, Wei-Yu Chen, Fangwen Fu, Hong Jiang, Hongzheng Li, and Daniel Rhee. C-for-metal: High performance simd programming on intel gpus. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 289–300. IEEE, 2021.

[205] Kenneth L Clarkson. An algorithm for approximate closest-point queries. In *Symposium on Computational Geometry*, pages 160–164, 1994.

[206] Amir Globerson and Naftali Tishby. Sufficient dimensionality reduction. *Journal of Machine Learning Research*, 3(Mar):1307–1331, 2003.

[207] Yuanwei Fang, Zihao Liu, Yanheng Lu, Jiawei Liu, Jiajie Li, Yi Jin, Jian Chen, Yenkuang Chen, Hongzhong Zheng, and Yuan Xie. Nps: A framework for accurate program sampling using graph neural network. *arXiv preprint arXiv:2304.08880*, 2023.

[208] Junyuan Xie, Ross Girshick, and Ali Farhadi. Unsupervised deep embedding for clustering analysis. In *International conference on machine learning*, pages 478–487. PMLR, 2016.

[209] Standard Performance Evaluation Corporation (SPEC). Specaccel® 2023 benchmark. `https://www.spec.org/accel2023/`, 2023.

[210] Junjie Li, Alexander Bobyr, Swen Boehm, William Brantley, Holger Brunst, Aurelien Cavelan, Sunita Chandrasekaran, Jimmy Cheng, Florina M. Ciorba, Mathew Colgrove, Tony Curtis, Christopher Daley, Mauricio Ferrato, Mayara Gimenes de Souza, Nick Hagerty, Robert Henschel, Guido Juckeland, Jeffrey Kelling, Kelvin Li, Ron Lieberman, Kevin McMahon, Egor Melnichenko, Mohamed Ayoub Neggaz,

Hiroshi Ono, Carl Ponder, Dave Raddatz, Severin Schueller, Robert Searles, Fedor Vasilev, Veronica Melesse Vergara, Bo Wang, Bert Wesarg, Sandra Wienke, and Miguel Zavala. Spechpc 2021 benchmark suites for modern hpc systems. In *ACM/SPEC International Conference on Performance Engineering*, ICPE '22, page 1516, New York, NY, USA, 2022. Association for Computing Machinery.

[211] Garrett M Morris, David S Goodsell, Ruth Huey, William E Hart, Scott Halliday, Rik Belew, and Arthur J Olson. Autodock. *Automated docking of flexible ligands to receptor-User Guide*, 2001.

[212] Leonardo Solis-Vasquez, Edward Mascarenhas, and Andreas Koch. Experiences migrating CUDA to SYCL: A molecular docking case study. In *International Workshop on OpenCL (IWOCL)*. ACM, 2023.

[213] Diogo Santos-Martins, Leonardo Solis-Vasquez, Andreas F Tillack, Michel F Sanner, Andreas Koch, and Stefano Forli. Accelerating autodock4 with gpus and gradient-based local search. *Journal of chemical theory and computation*, 17(2):1060–1073, 2021.

[214] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C Smith, Berk Hess, and Erik Lindahl. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1:19–25, 2015.

[215] Intel extension for PyTorch sources. `https://github.com/intel/intel-extension-for-pytorch/`.

[216] Efraim Rotem, Adi Yoaz, Lihu Rappoport, Stephen J Robinson, Julius Yuli Mandelblat, Arik Gihon, Eliezer Weissmann, Rajshree Chabukswar, Vadim Basin, Russell Fenger, et al. Intel alder lake cpu architectures. *IEEE Micro*, 42(3):13–19, 2022.

[217] Irma Esmer Papazian. New 3rd gen intel® xeon® scalable processor (codename: Ice lake-sp). In *Hot Chips Symposium*, pages 1–22, 2020.

[218] Hong Jiang. Intel's ponte vecchio gpu: Architecture, systems & software. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–29. IEEE Computer Society, 2022.

[219] Nevine Nassif, Ashley O Munch, Carleton L Molnar, Gerald Pasdast, Sitaraman V Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, et al. Sapphire rapids: The next-generation intel xeon scalable processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 44–46. IEEE, 2022.

[220] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily P Looi, Sreenivas Mandava, Andy Rudoff, Ian M Steiner, Bob Valentine, Geetha Vedaraman, et al. Cascade lake: Next generation intel xeon scalable processor. *IEEE Micro*, 39(2):29–36, 2019.

[221] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[222] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro*, 37(2):52–62, 2017.

[223] *oneAPI Specification.*

[224] Intel. Intel oneapi. `https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html`, 2023.

[225] NVIDIA. *NVIDIA CUDA Toolkit Documentation*, 2024.

[226] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W Hwu, et al. Spec accel: A standard application suite for measuring hardware accelerator performance. In *International Workshop on Performance*

Modeling, Benchmarking and Simulation of High Performance Computer Systems, pages 46–67. Springer, 2014.

[227] Intel Software Development Emulator (Intel SDE). `https://www.intel.com/software/sde`.

[228] Herman JC Berendsen, David van der Spoel, and Rudi van Drunen. Gromacs: A message-passing parallel molecular dynamics implementation. *Computer physics communications*, 91(1-3):43–56, 1995.

[229] The GROMACS molecular simulation toolkit. `https://github.com/gromacs/gromacs`.

[230] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems (NeurIPS)*, 32, 2019.

[231] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[232] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[233] Vitaly Zakharenko, Tor Aamodt, and Andreas Moshovos. Characterizing the performance benefits of fused cpu/gpu systems using fusionsim. In *Design, Automation & Test in Europe Conference (DATE)*, pages 685–688. IEEE, 2013.

[234] Richard L Graham, Galen M Shipman, Brian W Barrett, Ralph H Castain, George Bosilca, and Andrew Lumsdaine. Open mpi: A high-performance, heterogeneous mpi. In *International Conference on Cluster Computing*, pages 1–9. IEEE, 2006.

[235] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. In *International Euro-Par Conference*, pages 863–874. Springer, 2009.

[236] Aaftab Munshi. The OpenCL specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[237] James C Beyer, Eric J Stotzer, Alistair Hart, and Bronis R de Supinski. Openmp for accelerators. In *OpenMP in the Petascale Era: 7th International Workshop on OpenMP (IWOMP)*, pages 108–121. Springer, 2011.

[238] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, 2008.

[239] AMD ROCm. Hip: C++ heterogeneous-compute interface for portability. `https://github.com/ROCm/HIP`, 2024.

[240] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of CUDA and OpenCL. In *International Conference on Parallel Processing (ICPP)*, pages 216–225. IEEE, 2011.

[241] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391–407, 2012.

[242] Junghyun Kim, Thanh Tuan Dao, Jaehoon Jung, Jinyoung Joo, and Jaejin Lee. Bridging OpenCL and CUDA: a comparative analysis and translation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2015.

[243] Mayank Daga, Zachary S Tschirhart, and Chip Freitag. Exploring parallel programming models for heterogeneous computing systems. In *IEEE international symposium on workload characterization*, pages 98–107. IEEE, 2015.

[244] Lee Howes and Maria Rovatsou. *SYCL Specification – SYCL integrates OpenCL devices with modern C++*, 2015.

[245] Zhiming Wang, Yury Plyakhin, Chenwei Sun, Ziran Zhang, Zhiwei Jiang, Andy Huang, and Hao Wang. A source-to-source CUDA to SYCL code migration tool: Intel® DPC++ compatibility tool. In *International Workshop on OpenCL*, pages 1–2, 2022.

[246] Intel project for LLVM technology. `https://github.com/intel/llvm`.

[247] perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/`, 2012.

[248] Andi Kleen and Beeman Strong. Intel processor trace on linux. *Tracing Summit*, 2015, 2015.

[249] Trevor E. Carlson, Wim Heirman, Harish Patil, and Lieven Eeckout. Efficient, accurate and reproducible simulation of multi-threaded workloads. In *Workshop on Reproducible Research Methodologies (REPRODUCE)*, February 2014.

[250] Björn Gottschall, Silvio Campelo de Santana, and Magnus Jahre. Balancing accuracy and evaluation overhead in simulation point selection. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 43–53. IEEE, 2023.

[251] Brinkley Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, 2002.

[252] The sniper multi-core simulator. `https://snipersim.org`.

[253] Vincent M. Weaver. Self-monitoring overhead of the linux perf_event performance counter interface. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 102–111, 2015.