Distribution Agreement

In presenting this dissertation as a partial fulfillment of the requirements for an advanced degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my dissertation in whole or in part in all forms of media, now or hereafter known, including display on the world wide web. I understand that I may select some access restrictions as part of the online submission of this dissertation. I retain all ownership rights to the copyright of the dissertation. I also retain the right to use in future works (such as articles or books) all or part of this dissertation.

_____ _____

Si Chen Date

Efficiently Optimizing HPC Application Design
Across a Heterogeneous Hardware Environment

By

Si Chen
Doctor of Philosophy

Computer Science and Informatics

Avani Wildani, Ph.D.
Co-Advisor

Dorian Arnold, Ph.D.
Co-Advisor

Ymir Vigfusson, Ph.D.
Committee Member

Simon Garcia De Gonzalo, Ph.D.
Committee Member

Accepted:

Kimberly Jacob Arriola
Dean of the James T. Laney School of Graduate Studies

Date

Efficiently Optimizing HPC Application Design Across a Heterogeneous Hardware Environment

By

Si Chen
B.S., Huazhong University of Science and Technology, China, 2004
M.S., Huazhong University of Science and Technology, China, 2006

Advisors: Avani Wildani, Ph.D. Dorian Arnold, Ph.D.

An abstract of
A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Informatics
2024

Abstract

Efficiently Optimizing HPC Application Design
Across a Heterogeneous Hardware Environment
By Si Chen

Efficiently developing high-performance computing (HPC) applications is essential for optimizing performance and reducing economic costs. However, the inherent complexity of these applications, along with diverse heterogeneous hardware environments, poses significant challenges in application optimization. Heterogeneity complicates optimization due to differing memory architectures, processing capabilities, and communication patterns. Researchers often rely on proxy applications and simulations to estimate performance on various hardware platforms, but proxy applications often lack rigorous quantitative evaluation of their fidelity, and cycle-level accurate simulation remains inefficient, even with acceleration tools.

This dissertation addresses these challenges through three contributions. First, we develop a robust toolkit for characterizing and quantifying behavior similarities between HPC proxy applications and their corresponding parent applications. This ensures high fidelity in performance estimation and enhances the reliability of proxy applications in representing complex HPC applications. By identifying the most important features, we reduce data collection time by up to 95% while maintaining accuracy in representation.

Second, we improve a widely used simulation acceleration tool by integrating advanced clustering methods. This achieves a 5× speed up in simulation time while maintaining accuracy, enabling efficient exploration of design spaces for HPC applications across various hardware configurations. The enhanced simulation capabilities provide researchers with faster and more reliable means to evaluate application performance.

Third, we introduce a generalized model that combines meta-learning with architecture simulation to predict runtime across various applications and hardware systems. This approach facilitates rapid performance assessments and informed decision-making in HPC application design, achieving a 127× speedup in training time for additional tasks compared to traditional machine learning methods. The model's predictions can be practically applied to inform resource allocation and guide design choices in actual HPC workflows.

These three components address representation accuracy, simulation efficiency, and performance prediction. Together, they form a comprehensive framework for optimizing HPC application design process, making it faster, more cost-effective, and more adaptable to heterogeneous computing environments. This framework is designed to evolve alongside advancements in hardware, supporting new architectures and adapting to shifts in HPC workloads, ensuring its continued relevance in future HPC ecosystems.

Efficiently Optimizing HPC Application Design Across a Heterogeneous Hardware Environment

By

Si Chen
B.S., Huazhong University of Science and Technology, China, 2004
M.S., Huazhong University of Science and Technology, China, 2006

Advisors: Avani Wildani, Ph.D. Dorian Arnold, Ph.D.

A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Informatics
2024

Acknowledgments

I would like to express my deepest gratitude to my advisor, Dr. Avani Wildani. She has been an exceptional mentor and leader throughout my research journey, guiding me through every step——from reading papers and formulating research questions to job hunting. Her insightful advice has been indispensable to my growth as a researcher. I also sincerely appreciate my co-advisor, Dr. Dorian Arnold, for his considerate assistance and supervision of my doctoral studies.

I am immensely thankful to my dissertation committee, especially Dr. Simon Garcia De Gonzalo, whose constructive feedback and thoughtful suggestions have greatly improved the quality of my research. Meanwhile, I would also like to express my gratitude to my other collaborators at Sandia National Laboratory, Dr. Jeanine Cook and Dr. Omar Aaziz, who led me to the world of HPC.

I would like to extend my sincere appreciation to my internship mentors: Dr. Alma Dimnaku, Dr. Zhichao Li, and Haiying Xu, who selflessly shared their knowledge and expertise. They provided me with invaluable opportunities to learn practical techniques and apply my skills to real-world products and systems, enriching my understanding of the field.

I am deeply grateful to the members of the Emory Simbiosys Lab, led by Dr. Ymir Vigfusson. Our weekly meetings' friendly discussions, collaborative spirit, and mutual care have made this journey both enjoyable and inspiring. I cherish the friendships with Yazhuo, Shrey, Vish, and all my fellow PhD friends at Emory, who made my life so memorable.

Finally, I wish to express my profound gratitude to my family. My husband, kids, and parents have been my unwavering pillar of support through every challenge. Their love, encouragement, and understanding have given me the strength to persevere, and I could not have accomplished this without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

High-Performance Computing (HPC) plays an indispensable role in modern computational science, serving large-scale computationally intensive tasks across fields such as science, engineering, and economics. Built on intricate algorithms and massive datasets, HPC applications demand enormous computing resources. Their complexity significantly increases as expanding from traditional scientific simulations to artificial intelligence (AI) and machine learning (ML) workloads. For example, climate modeling now incorporates more variables and higher resolutions, and genomics research processes ever-larger datasets. This growing complexity demands more powerful and efficient computing solutions for diverse workloads.

To meet these escalating demands, HPC hardware has evolved significantly. Traditional CPU-based systems have reached their limits in performance scaling and energy efficiency, leading to the adoption of **heterogeneous computing environments**. These new systems combine a variety of hardware components. For instance, various processing units address different workload requirements: Graphical Processing Units (GPUs) excel at parallel processing for AI/ML tasks; Field-Programmable Gate

Arrays (FPGAs) provide customizable hardware acceleration for specific algorithms, Processing-in-Memory (PIM) architectures reduce data movement overhead, and even quantum processors. Additionally, diverse memory technologies are available, such as Double Data Rate (DDR) RAM, High Bandwidth Memory (HBM) for rapid access to data, and non-volatile memory (NVM) for fast data retrieval. Storage options are also varied, with Solid State Drives (SSDs) providing speed, Hard Disk Drives (HDDs) offering greater capacity, and network-attached storage facilitating data sharing across networks. Advanced networking hardware, such as Ethernet, InfiniBand, or custom interconnects, ensures efficient data transfer between components.

While this shift towards heterogeneity brings unprecedented computational power, flexibility, and scalability, it also introduces new challenges in efficiently using these diverse architectures [84]. Integrating multiple types of hardware adds complexity to optimizing performance, managing data movement, and controlling energy consumption. Currently, the world's most powerful supercomputer, Frontier, consumes over 22 megawatts (MW) of power [2], equivalent to the energy usage of approximately 18,000 American households. Besides energy consumption, suboptimal use of these diverse hardware components can lead to performance degradation and wasted resources. To fully leverage the potential of new hardware systems, applications often require substantial algorithmic modifications or complete redesigns. This highlights the importance of **co-design**, a collaborative approach where hardware designers and software developers work together to optimize both hardware and applications. Co-design ensures that applications are fine-tuned to the capabilities of heterogeneous systems, maximizing their performance and efficiency while minimizing energy costs.

Given the ever-increasing complexities in both software and hardware architecture, researchers have actively explored two main directions to facilitate co-design and optimization [3]: **proxy applications** and **simulations**.

Proxy applications are simplified versions of the actual, or parent applications.

While parent applications are large and have complicated dependencies, proxy applications share some important behaviors or characteristics of a parent application [31], facilitating easier scalability testing, algorithm optimization, and runtime performance evaluation. Researchers, vendors, and developers experiment with proxy applications using different optimization strategies without the heavy overhead of a large and complex code base.

Simulations model applications behavior across various hardware configurations without physical implementation. Simulation aids in workload characterization and performance tuning, providing valuable insights into which configurations best suit a given application. These insights can serve as a foundation for application optimization and vendor guidance. There's a trade-off between simulation time and accuracy, and detailed simulation provides more accurate results with longer simulation time. To address this, researchers often employ simulation acceleration techniques to reduce simulation time. One common method is to focus only on simulating the **region of interest (ROI)**—the part of the program most critical to performance analysis—while ignoring less impactful portions. This expedites simulation without sacrificing the accuracy of key insights. Other techniques include reducing the problem size or using sampling methods to approximate the program's behavior more efficiently.

Gap and Challenge Both approaches aim to gain a deeper understanding of application behavior, optimize code, and ultimately achieve better performance. However, optimizing application design across a heterogeneous hardware environment presents several challenges.

Challenge 1: Difficulty in Defining the Similarity Between Proxy and Parent Applications While proxy applications are designed to enhance the interaction between developers and parent applications by serving as simplified representations of the latter, accurately quantifying this representation, is challenging. **Fidelity** refers

to the degree of accuracy with which a proxy application mimics the behavior and performance characteristics of its parent application. An inaccurate proxy, or one with low fidelity, may lead to suboptimal system designs. For example, if a proxy application underestimates the communication patterns of its parent, it may result in a system with insufficient network bandwidth, severely degrading the parent application's performance. Several factors can affect the similarity between the proxy and parent application, including biases in programming language implementation, differences in data layouts, and varying parallelization approaches [80]. High fidelity ensures that insights gained from the proxy are directly applicable to the parent application, while low fidelity may lead to misleading conclusions. Moreover, lacking general guidelines or standards for selecting evaluation metrics and criteria further complicates the process. Additionally, the evaluation process is complicated by performance portability (*e.g.*, Code Portability,Performance Maintenance,Programming Model Compatibility) across different platforms [63].

Challenge 2: Slow Simulation Conventional simulation methods for runtime prediction are time-consuming [15] and often lack general applicability [78]. The slow pace of these simulations hinders rapid design space exploration and delays timely optimization decisions, potentially impeding the development of efficient HPC systems. Although researchers have attempted to address this timing issue through various simulation acceleration techniques, such as selecting representative regions for analysis, the effectiveness of these techniques remains a subject of ongoing research and debate, particularly concerning the representativeness of the chosen regions and the optimal number of regions required to balance accuracy and simulation time.

As new application requirements emerge at an unprecedented rate, traditional expertise-based heuristic approaches, which have historically ensured accuracy, are struggling to keep pace with the speed of hardware upgrades. Conventional analytical

models are increasingly unsuitable for evaluating system architectural designs due to the difficulty of managing a vast array of system configurations, as performance varies with minor configuration changes. In recent years, ML has begun to gradually transform the way computer architecture and systems are designed.

Challenge 3: Generalization in ML Models for Performance Prediction ML models, while often highly accurate (*e.g.*, 5% error) on specific tasks [55], often have a significant data requirement for training robust models, leading to a lack of model generalizability or ability to model novel architectures [121, 23]. Variable HPC application performance based on different inputs complicates performance extrapolation from small to large inputs. Adapting to a new application (with various inputs) necessitates retraining the model from scratch, which is time-consuming. Therefore, Application-specific ML models are impractical for constantly changing applications, hindering the development of broadly applicable performance prediction tools.

While **accuracy** remains important in optimization, **efficiency** (implying rapid adaptation in decision-making) and **generalization** (facilitating experience reuse by reducing repetitive and similar work) are equally crucial. In this diverse hardware ecosystem, current research lacks comprehensive methodologies to qualify the fidelity of proxy and parent applications, faces limitations in efficient simulation, and encounters difficulties in developing broadly applicable performance prediction models. This dissertation addresses these gaps by developing robust and adaptable optimization strategies for efficient, accurate, and generalizable optimization of HPC applications in heterogeneous environments.

1.2 Main Research Question

This dissertation strives to investigate these challenges and answer the following central research question: **How can we efficiently optimize HPC application design across a heterogeneous hardware environment using proxy applications, simulation, and ML modeling?** The term "efficiently optimizing" refers to accelerating the development cycle through rapid iteration. This approach aims to reduce the time and resources required for HPC application optimization, enabling developers to quickly improve application performance across diverse hardware platforms.

As illustrated in Fig. 1.1, this optimization strategy involves three key elements: proxy application representation, simulation of application behavior, and ML-based performance modeling. The figure exhibits how these elements are interconnected and imply their unifying concept: **Similarity**. This concept integrates the entire optimization process by comparing computational patterns, performance metrics, and behavior across applications and hardware.

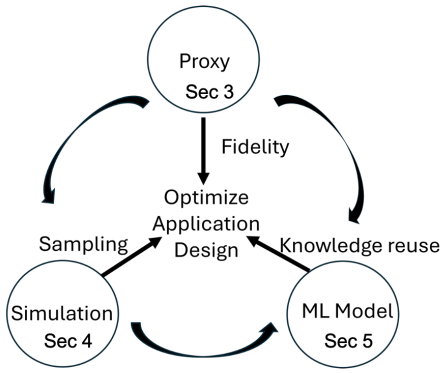


Figure 1.1: Three Approaches for Optimizing Application Design

- Proxy Representation. This component highlights the importance of representing the complex behavior of parent applications with proxy applications. The subquestion is: **How do we measure how closely proxy behavior on a system represents parent behavior?** This is crucial for reliable performance estimation when using proxy applications to facilitate easier testing and optimization compared to parent

applications. We use similarity measurements to quantify the closeness of the proxy and parent applications.

- Simulation. This component emphasizes the role of simulation in reducing the complexity and time required to evaluate HPC applications. The subquestion is: **What machine learning techniques can effectively identify the most representative segments from the application to preserve accuracy while greatly decreasing simulation time?** This is necessary for the accelerated simulation of HPC applications across a heterogeneous hardware environment, where a group of similar regions can be represented by one region of an application. We balance accuracy and efficiency using advanced dimension reduction and clustering approaches to select key regions.

- ML Modeling. The final component addresses the need for generalizable performance models. The subquestion is: **How can we generalize the modeling approach to gain insights into the performance of new applications quickly?** This is important for reducing the time needed to develop performance models for new applications. We build a meta-learning model by exploiting similarities across tasks within the same domain and transferring knowledge from the behavior of various applications on several heterogeneous hardware environments. Figure 1.1 shows how ML modeling builds on the insights from both proxy representation and simulation to create a meta-learning model.

These three elements are deeply interconnected: The proxy fidelity analysis provides a foundation for accurately representing complex HPC workloads, which is crucial for reliable performance estimation. The enhanced simulation acceleration technique informs efficient design exploration. Together, they support the development of a generalized performance forecasting model, enabling robust optimization across diverse hardware environments.

By leveraging similarity in these three areas, we can use proxy applications to represent complex HPC applications with minimal overhead, employ targeted simulations that capture essential performance characteristics, and develop ML models to predict performance across different hardware environments. Our goal of "efficiently optimizing" is to make the HPC application development process faster, more cost-effective, and more adaptable to the ever-evolving landscape of heterogeneous computing environments.

1.3 Research Contributions

My dissertation research combines multiple studies to optimize HPC application design across heterogeneous hardware environments efficiently. Each contribution addresses a specific subquestion and advances the state-of-the-art in HPC application optimization: Firstly, we quantify the fidelity of proxy applications using **Calder**. This contribution addresses the challenge of accurately representing complex parent applications with simpler proxies. Next, while proxy fidelity lays the groundwork for accurate simulation, we present **SimPoint++**, an advanced sampled HPC application simulation tool that significantly reduces simulation time while maintaining high accuracy. This tool enables rapid exploration of design spaces in heterogeneous environments. Finally, we develop **MetaCast**, a framework that combines proxy application simulations with meta-learning technology for quick and accurate runtime predictions of new applications on targeted architectures. This contribution enhances the generalizability of performance optimization across diverse HPC environments.

1.3.1 Quantify the Fidelity of Proxy Applications

This work focuses on characterizing similarities between proxy and parent HPC application behavior. Current methods for analyzing proxy application fidelity often

rely on manual comparisons and expert judgment, which can be subjective and time-consuming. We introduce a novel and versatile toolkit **Calder**. **Calder** provides a systematic and quantitative approach to assessing proxy fidelity, addressing issues such as lack of objective metrics and difficulty in identifying important features where proxy applications deviate from parent applications.

Contributions. Our primary contributions include:

1. The first comprehensive, quantitative characterization of proxy application fidelity at the hardware level using statistical similarity algorithms across a broad range of proxy and parent application pairs.

2. Advanced statistical techniques and machine learning algorithms that reduce data size by up to 95%.

3. The public availability of the **Calder** toolkit for providing actionable insights to improve proxy fidelity.

1.3.2 Accelerate Application Simulation

This work focuses on enhancing the simulation acceleration technology to provide reliable simulation regions. SimPoint is widely used in computer architecture research to automatically find a small set of simulation points that represent the complete execution of a program for efficient and accurate simulations. While many studies have used SimPoint as part of their methodology, there has been little consideration of whether the set of Simulation Points that SimPoint provides is as small as possible. We propose **SimPoint++**, an improved version of SimPoint that addresses its limitations. By incorporating advanced dimension reduction techniques and improved clustering algorithms, **SimPoint++** provides adaptive sampling strategies that dynamically adjust to the characteristics of each application.

Contributions. We make the following contributions to this work:

1. A novel approach to accelerate HPC application simulation, enhancing the original SimPoint tool, resulting in a 5x speed-up in simulation time compared to state-of-the-art solutions.

2. Provide comparable accuracy through advanced clustering and dimension reduction methods.

3. Overcome the limitations of existing methods with large-scale applications.

1.3.3 Generalize HPC Application Runtime Prediction

This work focuses on developing broadly applicable performance prediction models. Current runtime prediction methods are often constrained to specific applications and architectures. We introduce **MetaCast**, a novel approach that combines simulations based on proxy applications with meta-learning technology to enable quick runtime predictions of new applications on targeted architectures. **MetaCast** leverages similarities across tasks within the same domain and transfers knowledge from the behavior of various applications on several heterogeneous hardware environments.

Contributions. We present the following contributions:

1. A novel meta-learning approach that enables rapid adaptation to new applications and hardware configurations without extensive retraining.

2. Integration of proxy application simulations with advanced machine learning techniques to enhance prediction accuracy and generalizability.

3. We empirically validate **MetaCast** using real applications, demonstrating that with just ten training samples, **MetaCast** attains an average Mean Absolute

Percentage Error (MAPE) of 18% on the SPEC CPU 2017 benchmark and 25% on real applications.

4. This approach facilitates rapid performance assessments, achieving a 127× speedup in training time for additional tasks compared to traditional machine learning methods.

Chapter 2

Background

This chapter provides the literature review of the key areas pertinent to optimizing HPC application design, including the characterization of HPC applications, accelerated simulation methods, and runtime prediction techniques.

2.1 HPC Application Characterization

2.1.1 Building Proxy Application

As HPC systems grew more sophisticated, directly testing and optimizing full-scale (parent) applications has become increasingly challenging. Currently, the best approach to optimize application involves creating proxy applications that capture the essential characteristics of their parent while remaining tractable. An ideal proxy application mimics its parent's system-level overheads and behaviors, allowing efficient testing and optimization.

However, designing custom proxy applications from scratch for parent applications carries the risk of overfitting. Overfitting in this context means when the proxy application becomes too specialized to the specific conditions under which the parent application was observed. This can result in a fragile proxy application that is

susceptible to minor variations in parent usage or system design. Benchmark suites of proxy applications, such as those cataloged by the ExaScale Computing Project (ECP) [33], build proxies from shared properties classes of parent applications. These suites serve as powerful tools for system performance optimization and effectively reflect the behaviors of their corresponding parent applications. For example, a version of the SNAP potential in LAMMPS was initially implemented in the ExaMiniMD proxy app, but it performed poorly on GPUs. To address this, a new proxy app called TestSNAP was developed to experiment with various optimizations for different GPU programming models. These improvements were later integrated into the main LAMMPS code, leading to significant performance gains on GPUs [39].

In proxy application generation, Messer *et al.* [82] introduced a modular proxy-application framework that explores the performance impacts of communication interfaces and threading libraries. However, their work lacks quantitative performance comparisons. Yan *et al.* [124] generated synthetic code to replicate application behavior by tracing runtime events and evaluated performance similarity using a limited set of hardware counters. Additionally, Lehr *et al.* [69] developed tools for identifying and extracting computational kernels to create representative mini-apps, qualitatively assessing performance similarity through a limited set of relevant hardware performance counters. While these efforts aim to facilitate the generation of proxy applications, their evaluation processes are often not thorough or convincing enough to replace traditional proxy applications.

2.1.2 Proxy Application Characterization

Considerable research has been dedicated to characterizing proxy applications by comparing them to their parent counterparts.

Different studies have employed diverse comparison metrics. For instance, Aaziz *et al.* [12] utilized hierarchical clustering with a select subset of hardware performance

event counters to analyze behavioral similarities at the node level between proxy and parent applications. Similarly, Owenson *et al.* [88] focused on scalability within a single proxy/parent pair, while Kim *et al.* [61] used their KGen Fortran Kernel Generator tool to derive descriptive statistics from both parent applications and their resulting kernels, concentrating on kernel extraction. This process isolates the core computational components of an application, which is particularly useful for understanding and optimizing performance-critical sections. Further, Kwack *et al.* [64] applied the roofline performance model to assess application portability across GPUs, focusing on efficiency metrics rather than direct proxy-to-parent comparisons. The roofline model provides insights into the performance limits of a given computer architecture and how well an application utilizes the available resources. Innovatively, Islam *et al.* [56] developed the Veritas framework, which employs belief estimation in Dempster-Shafer theory to measure proxy/parent relationships through low-level resource measurements. This framework incorporates Principal Component Analysis (PCA) for dimensionality reduction and Grassmannian analysis for deeper similarity assessment. These characterization studies provide methods to assess similarity at different levels of granularity, thus helping to validate and improve proxy applications.

While these studies provide valuable insights into proxy characterization, they often focus on specific application pairs or methodologies, limiting their broader applicability. For example, Lin *et al.* [73] conducted a detailed analysis of one application pair, while Barrett *et al.* [20] developed a comparison methodology for several pairs, specifying performance domains such as computation time and inter-process communication. These studies, although robust, are largely specific to the application pairs studied, and lack a generalized framework applicable across various contexts.

Despite these contributions, a comprehensive evaluation of the metrics used to assess similarity among various proxy and parent applications remains lacking. This gap highlights the need for a systematic and generalizable framework that can facilitate

a better understanding of proxy application effectiveness across a broader range of contexts.

2.2 Accelerated Simulation

2.2.1 Simulator and Sampling Method

Simulation plays a crucial role in computer architecture research, enabling researchers to obtain diverse system performance data without resorting to numerous physical systems. Simulators can be classified based on three important factors [15]: simulation detail, target scope, and input type.

The level of detail in simulation is determined by its design, which can be functional, timing, or a combination of both. Functional simulators focus on the correctness of execution without considering timing, while timing simulators model the time taken for operations. Timing simulators can be further categorized into cycle-level, event-driven, and interval simulators. Cycle-level simulators model system behavior at each clock cycle, providing high accuracy at the expense of simulation speed. Event-driven simulators progress simulation time based on discrete events, offering a balance between accuracy and speed. Interval simulators, on the other hand, use analytical models to estimate the timing between important events, providing faster simulation with some loss of accuracy.

Regarding the scope of the target system, simulators can be classified as full-system or user mode. Full-system simulators model the entire computer system including the operating system, while user-mode simulators focus only on user-level code execution. Based on the input type, simulators can be classified as trace-driven and execution-driven. Trace-driven simulators use pre-recorded traces of program execution as input, whereas execution-driven simulators directly execute the program being simulated.

Notable examples of simulators include Gem5 [75], a cycle-level, modular, event-

driven simulator widely used in academia, and Sniper [21], which employs instrumentation tools (Pin tool) for parallel simulations on x86 system.

Sampling-based simulation techniques measure only selected sections (sampling units) of a benchmark's full execution stream. These techniques can be broadly categorized into statistical sampling (*e.g.*, SMARTS [123]) and targeted sampling (*e.g.*, SimPoint [95]). Statistical sampling selects sample units randomly or periodically, while targeted sampling aims to identify representative sections of the program execution. Researchers often use a targeted sampling method to simulate the region of interest in their applications.

2.2.2 SimPoint and Its Extensions

SimPoint [109] is founded on the observation that programs often exhibit repetitive behavioral patterns. It identifies representative pattern clusters and selects sample points from each cluster, enabling rapid sampling that accurately reconstructs the program's entire execution process. The SimPoint process consists of several key steps:

1. Program Slicing and Basic Block Vector (BBV) Generation: The program is first divided into intervals, with each interval representing a segment of execution. For each interval, Basic Block Vectors (BBVs) are generated to capture the program's behavior. A basic block, which serves as the foundation for these vectors, is a unit of code that has one entry point and one exit point.

2. Random Projection for Dimensionality Reduction: This step reduces the high-dimensional BBVs to lower dimensions for efficient processing.

3. K-means Clustering of Reduced BBVs: This groups similar intervals together.

4. Selection of Representative Sim_Points: One interval is chosen to represent each cluster.

5. Weight Assignment to Each Sim_Point: Weights are assigned based on the size of the cluster each point represents.

Extensions of SimPoint

Several works have built upon SimPoint's foundation. For example, BarryPoint [22] extends SimPoint that focuses on multi-threaded applications by identifying representative regions (simulation points) for each thread independently. PinPoint [92] implements SimPoint using the Pin dynamic instrumentation framework to gather program behavior data instead of simulation. LoopPoint [104] adapts SimPoint for loop-based multi-threaded program phase analysis.

SimPoint and its extensions may face challenges when applied to large-scale HPC applications. As the size and complexity of HPC workloads increase, the processes of dimensionality reduction and clustering become more computationally expensive. Moreover, the dynamic and irregular behavior of HPC workloads in heterogeneous systems makes it difficult to identify representative phases. These factors can potentially affect the accuracy of the selected simulation points.

2.2.3 Recent Advancements in SimPoint

Recent research has sought to improve SimPoint's accuracy and applicability. These advancements can be categorized into alternative profiling methods and alternative clustering methods.

Alternative Profiling Methods

Vengalam *et al.* [118] explored dynamic trace-based loop profiling, requiring fewer instructions per region than SimPoint. Ortiz*et al.* [87] proposed MEGsim for GPU workload characterization along the different stages of the graphics pipeline. Pati *et al.* [91] developed SeqPoint, an approach that accurately characterizes the behavior

of sequence-based neural networks by identifying some representative iterations. Flolid *et al.* [38] introduced SimTrace to represent a program's large-scale phase behavior over time phase. Additionally, Baddouh *et al.* [19] proposed a methodology targeted at GPGPU to reduce the simulation budget in scaled GPU workloads.

Alternative Clustering Methods

Hamerly*et al.* [45] and Sanghai *et al.* [105] compared the efficacy of multinomial clustering with K-means, concluding that a combination of both could reduce the number of simulation points needed without compromising accuracy. Multinomial clustering is a probabilistic method that assumes data points are generated from a mixture of multinomial distributions, which can be particularly beneficial for categorical data. Johnston *et al.* [58] employed a clustering model based on exponential Dirichlet compound multinomial (EDCM), a hierarchical Bayesian model capable of capturing more complex data structures than simple multinomial models. However, these approaches still utilize the Bayesian Information Criterion (BIC) method, like SimPoint, to determine the optimal number of clusters. BIC balances a model's likelihood with its complexity, helping to prevent overfitting. However, BIC's assumption of a Gaussian distribution, which is often not applicable to program behavior data.

Wudenhe *et al.* [122] propose TPUPoint, a performance analysis tool for TPU-based cloud platforms. TPUPoint comprises a profiler that automatically classifies recurrent patterns in TPU applications into distinct phases and an analyzer that offers three summarization methods: the conventional k-means algorithm, the Density-Based Spatial Clustering of Applications with Noise (DBSCAN), and a lower-overhead online linear-scan (OLS) algorithm. DBSCAN is a density-based clustering algorithm that can find arbitrarily shaped clusters and is robust to outliers. To determine the optimal number of clusters, TPUPoint employs the elbow method as a heuristic, terminating the clustering process when improvement ceases to increase significantly. The elbow

method involves plotting the explained variance as a function of the number of clusters and choosing the elbow of the curve as the number of clusters to use. Although TPUPoint replaced the clustering method in SimPoint, it lacks a comparison and emphasizes its advantages, resulting in it being overlooked within the larger framework.

These efforts have enhanced the simulation point selection techniques across various computing environments and application domains. While each approach offers unique benefits, there remains a pressing need for comprehensive comparisons of clustering methods and the identification of the most effective strategies in computer architecture simulation.

2.3 HPC Application Runtime Prediction

2.3.1 Application Specific Performance Evaluation

HPC runtime prediction methods have historically varied widely, encompassing rule-based categorizations [42], time series methodologies [110], statistics models [107], and Hidden Markov Models [99]. Hou *et al.* [51] argued that such approaches, are often tailored to jobs submitted from the same user, and thus lack general accuracy.

Simulation methods address these limitations by providing a controlled environment for predicting system behavior, helping workload characterization and performance tuning. However, there exists a trade-off between simulation time and accuracy, making it essential to balance these factors in runtime prediction efforts.

Conventional simulation methods for runtime prediction are often time-consuming [15] and may lack general applicability [78], which restricts their utility for large-scale challenges. Traditional analytical modeling techniques can improve the accuracy of classical simulations, but they are often expensive to both develop and run. For instance, interval analysis [36] and microarchitecture-independent characteristics [29] are employed to determine processor performance, yet both suffer from computational

inefficiencies. Interval analysis is a technique that breaks down processor execution time into intervals separated by miss events (e.g., cache misses and branch mispredictions). This breakdown allows for a more detailed understanding of performance bottlenecks by analyzing the impact of different types of misses on overall execution time. Conversely, microarchitecture-independent characteristics focus on program properties that are independent of the specific hardware implementations, such as instruction mix, data locality, and control flow complexity. These characteristics offer insights into program behavior without the need for detailed hardware-specific simulations. However, this approach necessitates the creation of new models for individual components, thereby constraining the ability of these analytical models to support novel designs.

2.3.2 The Role of Machine Learning

ML models are the current best performers in the HPC application modeling space([70, 89, 24, 77, 126]). ML is particularly powerful for performance analysis in complex systems because these models autonomously uncover latent patterns within training data. For instance, studies have shown that ML can be effectively utilized to predict job resource utilization in hPC environments. Tanash *et al.* [113] implemented ML techniques using Slurm data to enhance system efficiency and reduce power consumption. Additionally, Cengiz *et al.* [23] demonstrated that Deep Learning models could predict benchmark results on unseen hardware by learning from openly available SPEC 2017 benchmark results. Deep Learning models, particularly neural networks with multiple layers, can capture complex non-linear relationships in data, making them suitable for predicting performance across different hardware configurations.

Despite the advancement in ML for runtime prediction, a significant concern is the generalizability of ML models across different hardware architectures and application types [121, 23]. Models trained on specific datasets with high accuracy

may not perform adequately on new, unseen workloads or varying computational environments. Furthermore, the reliance on historical data can introduce biases, leading to prediction inaccuracy when workloads change significantly. Moreover, most existing models require substantial preprocessing and training time, which can hinder the decision-making process in dynamic HPC environments. There is a growing need for methodologies that facilitate rapid adaptation of ML models to new data and environments, ensuring that predictions remain accurate and relevant.

2.3.3 Cross-platform Performance Prediction

Attaining high accuracy in cross-application and cross-platform prediction poses a considerable challenge. Various approaches have used a combination of models and data to address this data limitation. Mankodi *et al.* [78] predicted the performance of target physical systems using a transfer learning model trained on a combined dataset from simulation-based systems and a source physical system. They utilized decision trees as the base model and applied a scaling factor to adjust predictions from simulated data to physical systems. However, dealing with small datasets typically yielded only moderate accuracy. Zheng *et al.* [127] used performance counters to predict ARM-based system performance from an x86-based system, while Qi [98] learned a matrix between platforms with the help of hardware performance counters. In Qi's approach, intermediate features (hardware performance counters) are divided into applications and platforms using Pearson Correlation Coefficients (PCC). This method involves building a power/performance model on the source platform, refining it with reduced training data on the target platform, and ultimately achieving a cross-platform model. Mariani *et al.* [79] employed transfer learning for cloud service performance prediction by coupling a cloud-side model (CP) with an application profile model (PP). In this case, the cloud provider generates a prediction model of the system, while the cloud user generates a prediction model of the target application.

The PP collects a hardware-independent profile for each training application and dataset, while the CP uses cloud configuration and the hardware-independent PISA profile to build runtime predictions. Additionally, Sun *et al.* [112] applied transfer learning, incorporating instruction counts such as loops, assignments, conditionals, and message-passing instructions as features to predict runtime on a target server from source servers. However, these methods require execution on the target hardware, which is incompatible with our goal of predicting the performance of potentially non-existent target systems. Moreover, the instrumentation involved in Sun *et al.*'s approach incurs significant overhead even for a single application.

While these approaches have made significant progress in cross-platform performance prediction, most methods lack generalizability across diverse computing environments due to data scarcity and difficulty in scaling efficiently with increasing system complexity. Some methods that rely on target hardware execution limit the usability for unseen systems. There is a need to develop a more robust and generalized method to address these limitations.

2.3.4 Meta-learning

Traditional ML models, particularly supervised models, are typically constructed for a singular, specific task. This process involves extensive training on numerous samples, where the model iteratively updates its parameters across multiple epochs, to minimize a loss function that measures the discrepancy between the predictions and the ground truth. Such models, while effective, often require vast amounts of data and suffer from decreased performance when applied to new tasks.

Meta-learning algorithms develop a generalized meta-model that can quickly adapt to new tasks with few training iterations using dual-level parameter updates: the meta-level (outer loop) and the task-specific level (inner loop). The meta-level focuses on adjusting high-level parameters, which could include initial settings, choice of

optimization algorithms, and network architecture, to set the stage for rapid learning on new tasks. The task-specific level then fine-tunes the model for optimal performance on individual tasks. Recent innovations have sought to enhance meta-learning through:

1. Improved task adaptation: This involves developing methods that can quickly adapt to new tasks with minimal fine-tuning. For example, Model-Agnostic Meta-Learning (MAML) [37] learns an initialization for the model parameters that allows for rapid adaptation to new tasks.

2. Computational efficiency: Techniques like Reptile simplify the meta-learning process by using first-order approximations [86], reducing computational complexity while maintaining performance.

3. Algorithmic optimizations: This includes developing more sophisticated optimization algorithms specifically designed for meta-learning, such as Meta-SGD [72], which learns not just the initial parameters but also the learning rates for each parameter.

4. Bolstering stability and generalization: Methods like MAML++ [18] introduce techniques like layer-wise learning rates and batch normalization to improve the stability and generalization of meta-learning algorithms. capabilities.

Meta-learning is often used in similar situations as transfer learning. However, while transfer learning repurposes a pre-trained model to tackle new, similar tasks, meta-learning crafts an adaptable meta-model, necessitating *minimal fine tuning* when encountering new tasks. Thus, meta-learning offers broader applicability without pre-supposing task similarity.

In the field of computer system architecture and HPC, the application of meta-learning has been limited. MetaTune [103] utilized a meta-learning-based cost model for optimizing parameters in Deep Learning compiler frameworks. This approach

involves learning a general strategy for tuning compiler parameters across different deep learning models and hardware configurations. Distinctively, Naghshnejad *et al.* [83] leveraged meta-learning to assess the reliability of system-generated job runtimes, aiming to enhance HPC scheduling. Their approach pivots on whether the prediction is highly confident: if so, a scheduled planning strategy is employed; otherwise, they resort to backfilling. Their model, based on gradient boosting with job descriptions as input, operated as an online scheduler, updated daily to manage new jobs. Gradient boosting is an ensemble machine learning technique that combines multiple weak learners (typically decision trees) to create a strong predictive model. The "online" nature of the scheduler means it can continuously learn and adapt to new data as it becomes available, rather than being trained once on a static dataset. This application of meta-learning to HPC scheduling demonstrates its potential for improving system performance and resource utilization in complex, dynamic environments. However, there remains significant room for further exploration and application of meta-learning techniques in HPC and computer architecture domains.

2.4 Conclusion

This chapter identifies existing methodologies and their limitations in optimize HPC application design across a heterogeneous hardware environment using proxy applications, simulation, and ML modeling. The evolution of proxy application characterization has underscored the importance of developing generalized frameworks that can be applied across various contexts, moving beyond methodologies that are specific to individual application pairs. Furthermore, advancements in simulation techniques, particularly through the use of methods like SimPoint and its extensions, highlight the ongoing need for improved accuracy and efficiency in sampling. Finally, the exploration of runtime prediction reflects the shifting landscape toward machine learning-driven

approaches, emphasizing the necessity for models that are adaptable, generalizable, and capable of real-time predictions.

Chapter 3

Beyond Guess and Check: Quantifying the Fidelity of Proxy Applications

3.1 Overview

High-performance computing (HPC) applications play a pivotal role in driving HPC system co-design, procurement, acceptance testing [65], and exploration of programming models [49] and communication bottlenecks [13]. Optimizing infrastructure for specific HPC applications can significantly enhance application performance predictability, security, and efficiency, leading to substantial power and cost savings. However, the complexity of these applications, along with their size and dependencies, can make this optimization difficult and time-consuming. Additionally, the sensitive nature of some HPC applications in terms of security and trade secrets often limits their availability to infrastructure designers. As targeted submodels, proxy applications offer a more manageable and privacy-preserving alternative for analyzing parent applications and therefore are valuable for system design and optimization.

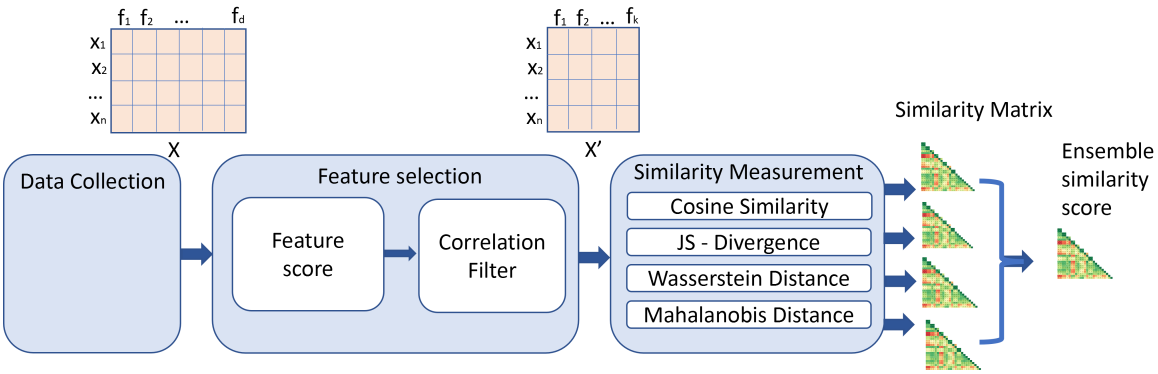


Figure 3.1: **Calder** Architecture

Proxy applications have successfully been used in HPC system design in the Exascale Computing Project (ECP) [27]. However, as parent applications continue developing and refining, their proxy counterparts undergo continuous transformation. The success of this approach relies on the **fidelity** with which proxy applications model the parent applications. Creating and maintaining an accurate proxy-parent relationship has been challenging without a consistent means of quantifying proxy fidelity.

Existing methods for quantifying proxy fidelity face several limitations. The lack of standardization in metrics and methodologies makes the comparisons across different studies difficult. The multi-dimensional nature of HPC application performance complicates the creation of a comprehensive fidelity metric. As HPC applications grow in size and complexity, traditional comparison methods become computationally expensive and time-consuming. Moreover, the dynamic behavior of HPC applications with different input data and system configurations further complicates the evaluation process.

Our work aims to bridge this gap by objectively quantifying the similarity between a parent and its proxy application. We introduce **Calder**, a novel and versatile toolkit that measures similarity across various axes, providing comprehensive, quantitative metrics for comparison. As depicted in Figure 3.1, **Calder** comprises three main

components: Data Collection, Feature Selection, and Similarity Measurement. The core functionality of **Calder** revolves around using unsupervised statistical techniques to evaluate the similarity in resource utilization between proxy and parent applications. By selecting diverse techniques that define similarity in uncorrelated ways, **Calder** interprets and synthesizes these perspectives to derive a unified measure of proxy fidelity.

We show how to select the most appropriate similarity algorithms for different datasets and correlate our results with external indicators, such as kernel timing. Furthermore, the similarity measurements and feature selection approach in **Calder** can also be applied to various HPC problems, including compiler optimization, code refactoring, and application input sensitivity. This practical and sustainable approach to evaluating how accurately proxy applications mirror their parent applications fulfills a critical need in the scientific community, providing a valuable tool for researchers and developers in HPC co-design.

3.2 Methods

3.2.1 Hardware Performance Counters

Advantages of Hardware Performance Counter

Common approaches for evaluating HPC application performance [76] include profiling, such as gprof, which provides detailed information about function-level performance and resource usage; tracing, like strace, which captures the sequence and timing of events during execution; and benchmarking, which allows for standardized performance comparisons. Performance modeling can predict behavior under various conditions, while static/dynamic code analysis and scalability testing offer additional insights. Each method has its strengths and is suited to different aspects of performance

evaluation.

Among these methods, hardware performance counters stand out as particularly effective for profiling applications. These special-purpose registers, built into modern processors, count hardware-related events such as cache misses, branch mispredictions, and CPU cycles. Hardware performance counters offer several advantages in comparing proxy and parent applications [74]. They provide low-level metrics that directly reflect the interaction between the application and the hardware. Additionally, they introduce minimal overhead to the application's execution, ensuring that the performance data collected is representative of the application's behavior. Moreover, hardware performance counters can capture a wide range of performance aspects simultaneously, allowing for a multi-faceted comparison between proxy and parent applications. This unique combination of depth, precision, and efficiency makes hardware performance counters our chosen method for comparing proxy and parent applications.

Usage of Hardware Performance Counters

Modern processors typically have limited hardware performance counter registers. Many Intel processors have 3 to 4 fixed-function counters and 4 to 8 programmable counters [53], while some AMD processors might have up to 6 programmable counters. Fixed-function counters are dedicated to specific, predefined events (usually critical ones like CPU cycles, instructions retired, etc.). Programmable counters are more flexible and can be configured to count a wide variety of events.

Due to hardware constraints, only a limited number of counters can be monitored simultaneously in a single run. When the number of desired performance events exceeds the available hardware counters, two common solutions are employed: time-multiplexing and sampling techniques. The time-multiplexing method switches between different sets of events during the program's execution, meaning not all events are monitored continuously. The sampling method, on the other hand, periodically

reads the counters and estimates the total counts. Both methods have drawbacks: time-multiplexing may miss short-lived events, while sampling can lead to statistical inaccuracies.

To overcome these limitations and collect comprehensive data, multiple program executions are necessary, each monitoring a different set of events. While this approach provides more accurate and complete performance analysis, it can be time-consuming and may introduce variability between runs. To address this, we monitor events in small groups and run each group 5 times to decrease variance. Additionally, architectural dependencies of hardware counters can complicate comparisons across different systems, so our compare the counter performance of applications within the same system. To fully understand the root causes of low-level hardware performance counter results, we also incorporate codebase analysis as a supplementary technique.

3.2.2 Feature Selection

Given the vast array of hardware performance counters available on most HPC architectures, identifying a comprehensive yet minimal set of features is challenging. There are two main concerns:

1. Data Collection is Time-Consuming. Collecting and processing vast amounts of data can be time-intensive and expensive, especially in HPC environments.

2. The Curse of Dimensionality. As the number of features increases, the volume of the feature space grows exponentially, which can lead to sparse data representations.

Feature selection and feature extraction are two categories of feature dimension reduction techniques. Feature selection chooses a subset of relevant features based on certain criteria, which is more suitable for our task. There are three kinds of feature selection methods. Filter Methods evaluate the intrinsic properties of the data independently of any learning algorithm. Examples include assessing data variance

and using Fisher scores to rank features by their importance. Wrapper Methods use a specific learning algorithm to evaluate the performance of different feature subsets, such as through recursive feature elimination. Embedding Methods select features during the construction of the learner itself, as seen in algorithms like random forests, which inherently evaluate feature importance.

Filter methods offer several advantages for our research context. They are computationally less expensive and not tied to a specific learning algorithm. Additionally, they provide clear interpretability through feature ranking. Given these benefits, we implement an efficient and effective filter-based feature selection process in this research. Our feature selection layer consists of the *Feature Score* that ranks the important features and the *Correlation Filter* that removes the correlated features.

Feature Score Methodology

Our feature selection process begins with the construction of a data tensor X that comprises n rows of application samples $x_1, ..., x_n$ and d features $f_1, ...f_d$. To isolate the most impactful features while maintaining the structural integrity of the data, we employ an unsupervised, graph-based ranking technique known as the Laplacian score [46].

The Laplacian score leverages graph-based representation and spectral properties to capture the local geometric structure of the data. It is a perfect choice for our task where maintaining local neighbor relationships (similarity between proxy and parent applications) is important. The Laplacian score can handle non-linear relationships and is particularly useful when dealing with complex, high-dimensional data. Furthermore, it can be used with various similarity measures to construct the graph.

The Laplacian score for the r^{th} feature is calculated as follows:

$$L_r = \frac{\Sigma_{ij} \left(f_{ri} - f_{rj}\right)^2 S_{ij}}{\text{Var}\left(\mathbf{f}_r\right)}, \tag{3.1}$$

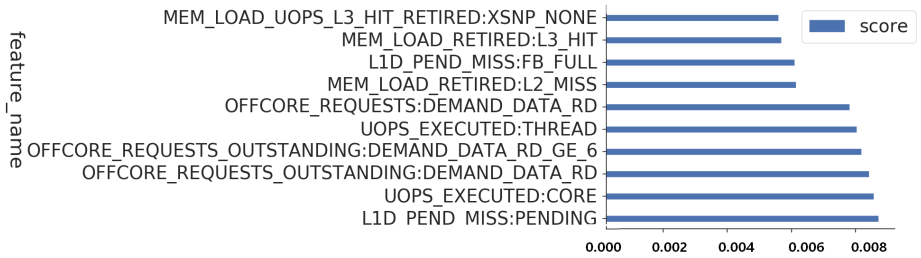


Figure 3.2: Laplacian Score for the Top 10 More Important Features Pre-correlation Filter

where the weight matrix $S_{ij} = e^{-\frac{\left\|x_i - x_j\right\|^2}{2}}$ is non-zero only for neighboring points. f_{ri} is the r^{th} feature value for point i. The score prioritizes features with lower values, as they are more critical for preserving neighbor similarity. Figure 3.2 shows the Laplacian scores for the hardware events (features) sorted in ascending order to identify the most important contributors to similarity. Feature names are on the y-axis, and corresponding scores are on the x-axis. For example, the first feature *'MEM_LOAD_UOPS_L3_HIT_RETIRED:XSNP_NONE'*, with a Laplacian score of 0.005378, is more essential for determining similarity than the consecutive feature *'MEM_LOAD_RETIRED_L3_HIT'*, with a score of 0.005463.

Correlation Filter

Despite the efficacy of the Laplacian score in identifying important features, it does not address feature redundancy. To refine our feature set, we apply a correlation filter using the Pearson correlation coefficient (PCC), which measures the linear relationship between two variables. The coefficient ranges from +1 (perfect positive correlation) to -1 (perfect negative correlation), with 0 indicating no linear correlation. We set a threshold of 0.9 for PCC to identify and remove highly correlated features, thus reducing redundancy without losing critical information:

$$\rho_{f_i,f_j} = \frac{\text{cov}(f_i, f_j)}{\sigma_{f_i}\sigma_{f_j}}, \tag{3.2}$$

After computing PCC values for all ranked features, we selectively include the most informative features, discarding any that exceed the correlation threshold (> 0.9 or < -0.9). This process continues until all significant features are selected.

While PCC is effective for evaluating linear relationships, it may not capture non-linear dependencies typical in more complex hardware event metrics. In such cases, we recommend using the Kendall rank correlation coefficient [59] for a more nuanced analysis.

3.2.3 Similarity and Distance

To evaluate the similarity between applications, we calculate the pairwise distance for each application pair using four representative similarity measurement methods. These methods were chosen for their complementary strengths in capturing different aspects of similarity. We then compare the outcomes of these similarity metrics and aggregate the results by averaging the similarity scores across all four methods to capture a comprehensive view of application similarity.

Cosine Similarity

Cosine similarity measures the angle between vectors in an inner product space. The inner product can be conceptualized as the projection of one vector x_i in the direction of another vector x_j. We choose this metric for its ability to capture directional similarity, regardless of magnitude differences. This is particularly useful when comparing application behaviors that may differ in scale but share similar patterns. The cosine similarity is defined as:

$$\cos(\theta) = \frac{\sum_{k=1}^{d} x_{ik} x_{jk}}{\|x_i\| \|x_j\|}.$$

The cosine similarity ranges from 1.0 (identical vector direction) to 0.0 (orthogonal vectors), with the angle θ varying from 0° (equivalent) to 90° (dissimilar). If two applications exhibit similar behaviors, their cosine similarity angle is expected to be closer to 0°.

Jensen-Shannon (JS) Divergence

JS divergence [34] measures the distance between two probability distributions P and Q. We select this metric for its ability to handle probability distributions and its symmetry, which is advantageous when comparing application pairs. First, we normalize each vector by dividing its elements by the sum of all elements, converting the vector into a probability distribution. JS divergence is a symmetric measure and a generalization of Kullback–Leibler (KL) divergence [62], which is defined as:

$$\begin{aligned} KL(P|Q) &= \sum_x P(x) \log \frac{P(x)}{Q(x)} \\ &= -\sum_x P(x) \log Q(x) + \sum_x P(x) \log P(x) \\ &= \text{cross entropy} - \text{entropy}. \end{aligned}$$

Unlike KL divergence, which is asymmetric and unbounded, JS divergence is symmetric and returns a value between 0 and 1, where values near 0 indicate similarity and values near 1 indicate divergence. JS divergence is defined as:

$$\text{JS}(P\|Q) = \frac{1}{2} KL(P\|M) + \frac{1}{2} KL(Q\|M), \quad M = \frac{1}{2}(P+Q).$$

Wasserstein Distance(WD)

WD[102], also known as Earth Mover's Distance, measures the minimum "cost" of transforming one probability distribution P into another Q. This "cost" is quantified as the amount of distribution weight moved, multiplied by the distance it is moved.

We include WD because it considers the order of events and compare distributions of different lengths. This is suitable for comparing application performance across platforms with varying hardware event counts. The p^{th} WD is defined as:

$$W_p(P,Q) = \left(\inf_{J \in \mathcal{J}(P,Q)} \int \|x-y\|^p dJ(x,y) \right)^{1/p},$$

where $\mathcal{J}(P,Q)$ denotes all joint distributions J for (X,Y) that have marginals P and Q. We use the first WD ($p=1$) between two 1-dimensional distributions. WD has no upper bound; 0 indicates equivalence, and values near 0 indicate similarity, while increasing values indicate growing divergence.

Mahalanobis Distance (MaD)

MaD is a multivariate distance metric that measures the distance between a point (vector) and a distribution, or between two points from the same distribution. We choose this metric for its ability to account for the covariance structure of the data, making it valuable for multivariate analysis and detecting subtle differences in application behavior. The MaD between two vectors x_i and x_j from the same distribution is defined as:

$$d_M(x_i, x_j) = \sqrt{(x_i - x_j)^T S^{-1} (x_i - x_j)},$$

where S is the covariance matrix of the dataset. Geometrically, MaD transforms the data by whitening and normalizing the covariance, then computes the Euclidean distance for the transformed data. It accounts for the variance of each variable and the covariance between variables, making it useful for multivariate anomaly detection and imbalanced classification. Note that MaD requires more samples than features to calculate the covariance matrix, so we reduce dimensionality with PCA beforehand. MaD has no upper bound; 0 indicates equivalence, and values near 0 suggest similarity,

while larger values indicate growing divergence.

3.3 Experiment

3.3.1 Application Suite

Our study uses a comprehensive suite of proxy and parent applications across various scientific domains, detailed in Table 3.1, alongside standard HPC benchmarks to establish a performance baseline. Many of these applications are widely used in their respective fields and represent real-world scientific workloads. We selected proxy applications that are officially recognized and maintained by the parent application developers or reputable HPC organizations. Due to export controls, not all applications have corresponding proxy/parent pairs. In these cases, we included additional HPC applications and benchmarks to provide a comprehensive view of HPC workloads and to establish performance baselines.

Each proxy application is tailored to replicate the computational, communication, and memory behaviors of its respective parent application, as documented by the developers. We ensure consistency by using matching or similar input problems and parameters for each proxy/parent pair, with all applications consuming approximately 50% of available memory. A single representative input reflecting typical workloads was chosen for each application to standardize resource utilization.

- LAMMPS [97] is a classical molecular dynamics code, with particles ranging from a single atom to a large composition of material. It implements mostly short-range solvers but does include some methods for long-range particle interactions.

- ExaMiniMD [115], which is a proxy for LAMMPS, implements limited types of interactions, and only short-range ones.

Table 3.1: Proxy/Parent and control apps

Proxy	**Version**	**Parent**	**Version**	**Other apps**	**Version**
ExaMiniMD	1.0	LAMMPS	8/17/2017	AMG2013	2013_0
miniQMC	0.4	QMCPACK	3.8	Castro	20.07
miniVite	1.0	Vite	9/30/2020	Laghos	3.0
Nekbone	3.1	Nek5000	19.0	PENNANT	0.9
PICSARlite	7/16/2020	PICSAR	7/16/2020	SNAP	1.09
SW4lite	2.0	SW4	2.0	HPCG benchmark	3.1
SWFFT	1.0	HACC	1.0	HPCC benchmark	1.5.0
XSBench	19.0	OpenMC	0.11.0		

- QMCPACK [60] is a quantum Monte Carlo package for computing the electronic structure of atoms.
- MiniQMC [100] covers QMCPACK's essential computational kernels. The computational themes of miniQMC and QMCPACK are particle methods, dense and sparse linear algebra, and Monte Carlo methods.
- Vite [41] is an implementation of Louvain method for (undirected) graph clustering or community detection.
- MiniVite [40] is a proxy application for Vite that implements a single phase of the Louvain method in distributed memory for community detection.
- Nek5000 [85] is a spectral element computational fluid dynamics solver.
- Nekbone [8], which is a proxy application for Nek5000, solves the Poisson equation with a spectral element multigrid preconditioned conjugate gradient solver.
- PICSAR [119] is Particle-In-Cell solver.
- PICSARlite [11] which is a proxy application for PICSAR, is a subset of the actual codebase.

- SW4 [96] is a geodynamics code that solves 3D seismic wave equations with local mesh refinement.

- SW4lite [33] is a scaled-down version of SW4 that has limited seismic modeling capabilities, but does solve the elastic wave equation and uses some of the same numerical kernels as those implemented in SW4.

- The Hardware Accelerated Cosmology Code (HACC) [43] is an N-body framework that simulates the evolution of mass in the universe, with both short and long-range interactions. The long-range solvers implement an underlying 3D FFT.

- SWFFT [33] is the 3D FFT that is implemented in HACC. Since this FFT accounts for a large portion of the HACC execution time, SWFFT serves as a proxy for HACC.

- OpenMC is a Monte Carlo particle transport code [101].

- XSBench [117] is a proxy application for OpenMC and represents the continuous energy macroscopic neutron cross-section lookup kernel, which is a key computational kernel of Monte Carlo particle transport.

- AMG2013 [47] is a proxy application for BoomerAMG and is a parallel algebraic multigrid solver for linear systems arising from unstructured grid problems. We ran the default Laplace problem with a custom resizing.

- Castro [17] is an adaptive mesh, astrophysical radiation hydrodynamics simulation code.

- Laghos [30] is a proxy application that is a high-order Lagrangian hydrocode meant to represent several compressible shock hydrocodes, including BLAST.

- PENNANT [7] serves as a proxy application for rad-hydro physics-based algorithms on an unstructured mesh, modeling the computation and memory access

Table 3.2: Hardware Characteristics

Component	**Skylake**	**Power9**
L1 data cache (private)	32 KB, 8-way	same
L1 instr. cache (private)	32 KB, 8-way	same
L2 cache (per core)	1 MB, 16-way	512KB, 8-way
L3 cache (shared)	24.75MB, 11 way	120MB, 20-way
Memory (per node)	192 GB, DDR4-2666	256GB, DDR4-2667
Cores/threads	18/36	24/48
Sockets/node	2	same
Total nodes	1488	54
Interconnect	Omnipath	Mellanox EDR Infiniband
Max Memory BW (per processor)	20GB/sec	170 GB/sec
Memory channels (per socket)	6	8

patterns typical to rad-hydro applications. It is modeled on, and thus serves as a proxy for, the LANL code FLAG.

- SNAP [10] serves as a proxy application for discrete ordinates neutral particle transport, modeling the computation and memory access patterns typical to neutral particle transport applications. It is modeled on, and thus is a proxy for, the LANL code PARTISN.

- HPCG [9] implements a suite of computational and data access patterns that closely match a broad set of important scientific applications.

- HPCC [5] is a benchmark suite designed to exercise standard memory access patterns that are common to many scientific applications.

3.3.2 System Platform

For data collection, we utilized two hardware systems: an Intel Skylake and an IBM Power9, both characterized in Table 3.2. These systems run RHEL7.8 and RHEL7.6 respectively. All applications were executed in MPI-only mode, using 128 ranks across

four nodes to balance the feasibility of conducting numerous experiments with the need to capture significant communication behaviors.

3.3.3 Data Collection and Preprocessing

Our data collection infrastructure employs the Lightweight Distributed Metric Service (LDMS [14]) and the Performance Application Programming Interface (PAPI [114]) to gather performance counter data. To ensure data quality, we examined all available performance events (over 500) on our hardware platform and conducted functional tests. The limited number of performance counter registers necessitates multiple runs for complete data collection.

The data was categorized into 15 subgroups (*e.g.*, Dispatch_Pipeline, Instruction_Cache) based on vendor recommendations [125] and node architecture insights. These subgroups were further organized by architectural concepts (*e.g.*, *cache*, *branch prediction*, *virtual memory*) to clarify the relationships between proxy and parent applications in terms of node components.

To ensure robustness and reliability, each event subgroup was run five times for each application, totaling over 3000 data collection runs. We processed this data by averaging the results across all ranks and runs, and then we normalized these averages by the number of CPU cycles executed. By normalizing based on CPU cycles, we establish a consistent method for comparing event rates across various applications and execution scenarios. However, this approach has limitations: it may introduce biases in I/O-bound or memory-bound scenarios (CPU waiting), might not fully capture infrequent but significant events, and assumes equal value for all CPU cycles. Despite these constraints, we believe this normalization offers a reasonable comparison basis.

This normalization produced a 500+ element vector for each application, representing a robust metric vector for similarity analysis. While a single metric sum might not fully capture the intricacies of a time series, using a vector of multiple metrics

offers a more comprehensive representation. Our analysis in §3.4.2 demonstrates that similar codebases consistently generate similar metric vector pairs. Thus, we consider our assumption of similarity based on sums of scalar metrics to be well-founded. To generate vectors for each application, we sample accumulated hardware counters throughout execution and calculate the average of these counters for the last 5 seconds of execution across all ranks. This yields an application vector x_i, which contains a series of averaged hardware event counters (denoted as x_{ik}).

We remove irrelevant (noisy) features before they are ranked. Hardware events in our collection platform have a prefix (Table III), which allows us to filter some events using domain knowledge. We observe, for instance, that a large number of hardware events with the prefix 'OFFCORE RESPONSE' always show extremely small values with little variance. Additionally, incomplete data features were removed to maintain consistency across results.

Despite the option to apply preprocessing steps like centralization (zero mean), normalization (norm 2 equals 1), or standardization (variance equals 1), we retained the original scales of our data to preserve the inherent physical meanings (hardware event counts per CPU cycle) and the relationships between features.

3.4 Results

3.4.1 Similarity Matrix Comparison

Intel Skylake platform

We use four similarity algorithms to evaluate the accuracy of the resultant proxy/parent pairs. Since the similarity matrix is symmetric on the diagonal, we use lower triangular heatmaps (Figure 3.3, 3.4a, 3.4b, and 3.4c) to visualize similarity. To facilitate direct comparison between methods, we scale the JS divergence, Wasserstein distance, and

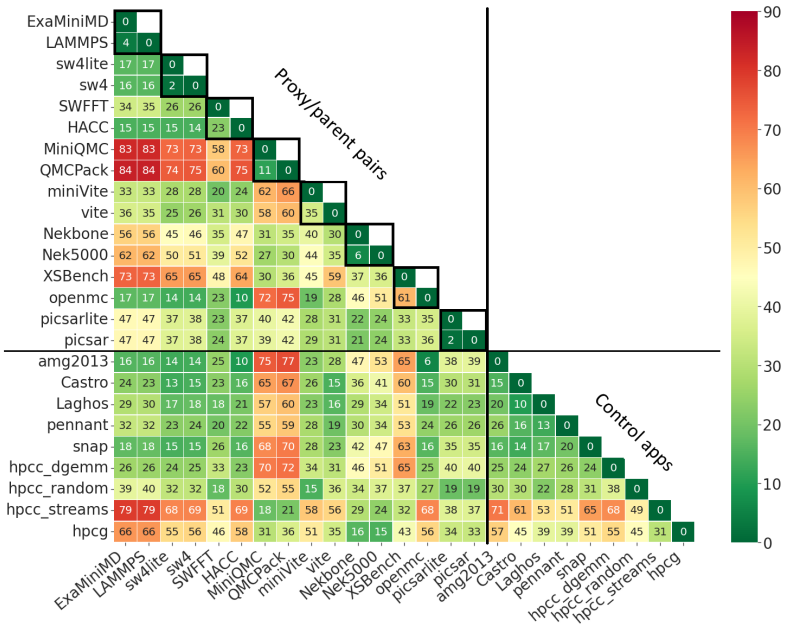
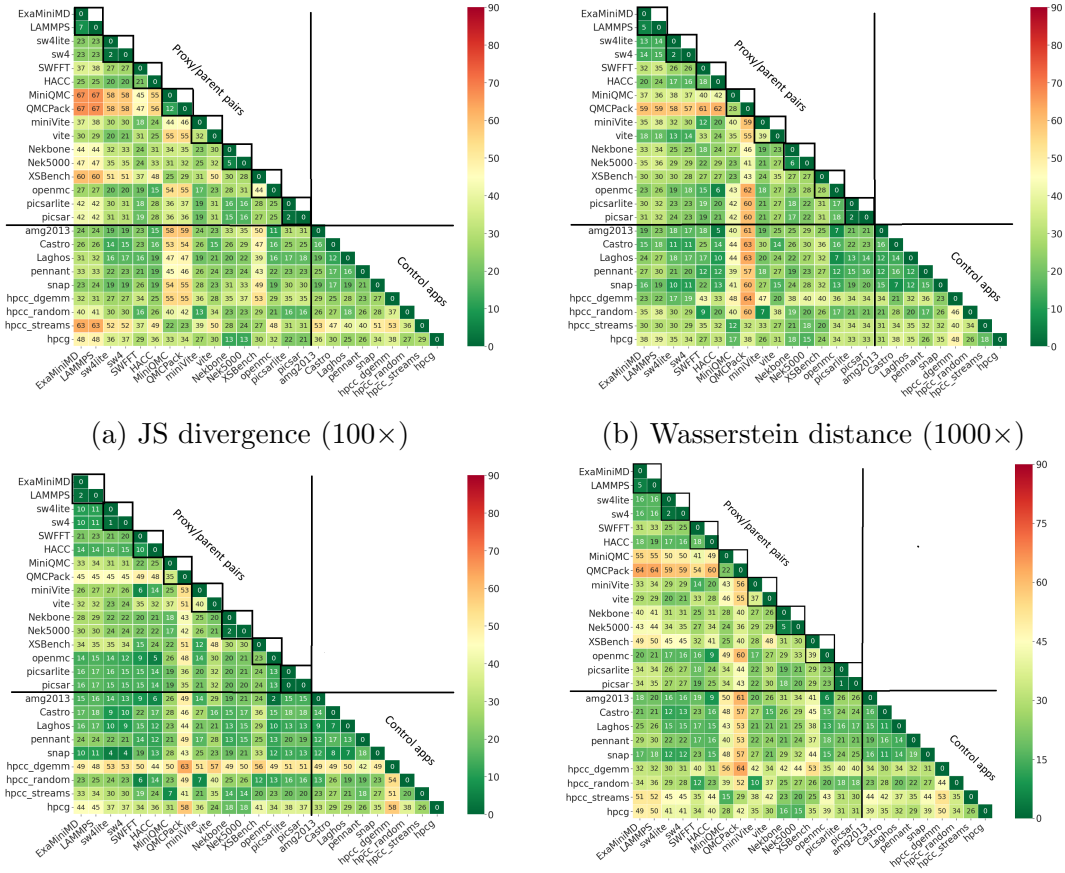


Figure 3.3: Cosine Similarity, Skylake

Mahalanobis distance values (multiplying by 100, 1000, and 10 respectively) to match the 0-90 range of cosine similarity scores. Figure 3.4d shows the aggregate result of averaging similarity scores across all four methods.

The diagonal entries are zero, indicating the distances between the applications and themselves. We normalize the scales in each figure for comparability. Dark green indicates high similarity, while dark red indicates high dissimilarity. Proxies with corresponding parents are listed first on the axes, with their parent immediately following. Eight 2×2 black-bordered blocks highlight the relationships between proxy/parent pairs. We expect the lower left of these blocks to be dark green, indicating high similarity. The nine miscellaneous applications (either proxies with no parents or vice versa) are listed at the bottom and right of the axes. Two lines divide each figure, placing proxy/parent pairs in the upper left quadrant and control applications in the lower right. Setting a threshold for similarity can be complex [71], as it depends on the design goals of the proxies. We set the threshold to 30 for our suite of algorithms, based on code-based kernel function analysis (§3.4.2) and the range of cosine similarity.



(a) JS divergence (100×)

(b) Wasserstein distance (1000×)

(c) Mahalanobis distance (10×)

(d) Similarity for ensemble methods

Figure 3.4: Our similarity methods show similar results for proxy-parent agreement.

Overall, the four distance metrics return similar correlations. For example, Figure 3.3 identifies pairs such as PICSARlite/PICSAR, SW4lite /SW4 as highly similar proxy/parent pairs using cosine similarity, These pairs are known to functionally represent similar applications (*e.g.*, PICSARlite is a subset of the PICSAR codebase). The cosine similarity results suggest that these proxy applications effectively capture the computational characteristics of their parent applications.

Two known proxy-parent pairs exhibit some behavioral gaps: miniVite/Vite are moderately similar, with a 35° angle between them, and XSBench/OpenMC are highly dissimilar, with a 60° angle between them. This discrepancy likely arises from differences in complexity between the parent and proxy; XSBench performs only the cross-section lookup portion of Monte Carlo neutron transport, while OpenMC

implements the full neutron transport code, which can better mask poor cache/memory behavior. For HPC system designers, this implies that miniVite may be a reasonable proxy for Vite in some scenarios, but caution should be exercised when using XSBench as a proxy for OpenMC, particularly in cache and memory-related studies.

The unpaired proxy applications (AmMG2013, Castro, Laghos, PENNANt, and SNAP) show relative similarity to each other but do not match the known pairs. Interestingly, AMG2013 is similar to OpenMC, suggesting it could serve as a potential proxy for OpenMC in certain computational contexts. For HPC system designers, this finding implies AMG2013 as a candidate for further validation (§ 3.4.5) and testing, especially in areas where memory-intensive computational patterns are critical. The other four HPC benchmark-related applications do not demonstrate mutual similarity, likely due to their design as synthetic programs measuring distinct memory or data patterns rather than simulating specific applications.

The pairs MiniQMC/QMCPack are relatively similar but distinctly different from other applications. Both are particularly sensitive to floating point, memory bandwidth, and memory latency performance, so it is not surprising that they are similar to HPCC_streams, which measures sustainable memory bandwidth. In our suite, only Nekbone/Nek5000 are similar to HPCG, which assesses the performance of basic operations (*e.g.*, matrix multiplication, vector updates); we attribute this to the gradient iterations in Nekbone/Nek5000 aligning closely with those cataloged by HPCG.

We also observe diversity in results across the four similarity algorithms. Dark red areas indicate that HPCC_streams and HPCG are highly dissimilar to other applications in terms of cosine similarity, JS divergence, and MaD, but similar in WD. This discrepancy might be attributed to WD placing a greater emphasis on the order of features within a vector, causing any change in the sequence to impact the WD calculation. For instance, if divergent events are positioned far apart in the vectors,

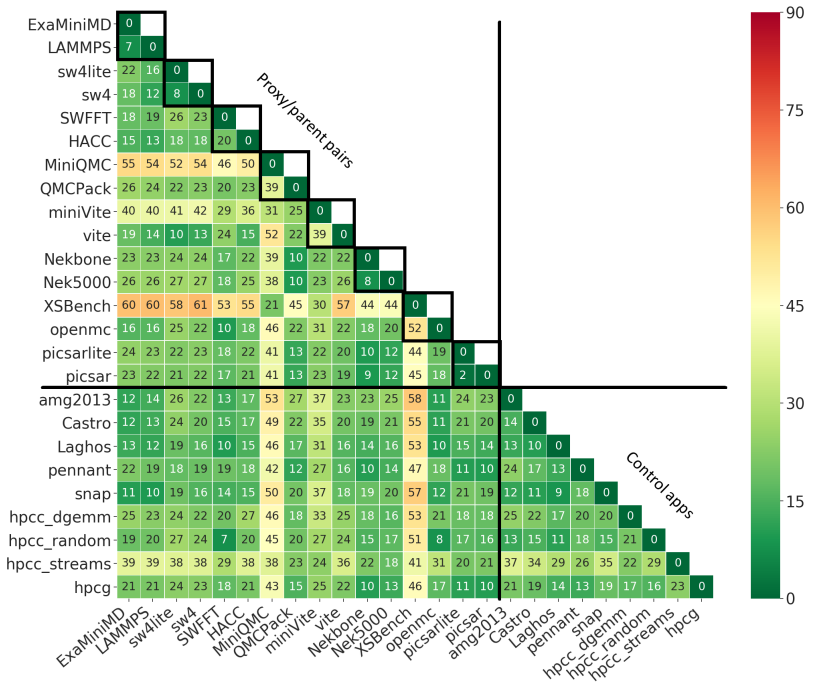


Figure 3.5: Cosine Similarity with All Features, Power9

the WD value increases. Notably, HPCC_degemm differs from other applications only in MaD, likely due to the whitening process utilized. MiniQMC/QMCPACK diverge more in WD and MaD compared to the other two algorithms. While most differences stem from how algorithms assess memory and cache subgroups (§3.4.5), the unexpected divergence between MiniQMC/QMCPACK in WD and MaD requires further investigation. For HPC system designers, this highlights the importance of selecting the appropriate similarity metric based on the specific performance characteristics being evaluated.

IBM Power9 Platform (Power9)

While the Power9 platform shares many characteristics with the Intel Skylake platform (Table 3.2), notable differences exist in the memory subsystem and SIMD (Single Instruction Multiple Data) widths, which may affect parallel performance. The Skylake processor supports up to 512-bit SIMD, whereas Power9 supports only 128-bit SIMD. For about half of the applications, we observed similar execution times on both

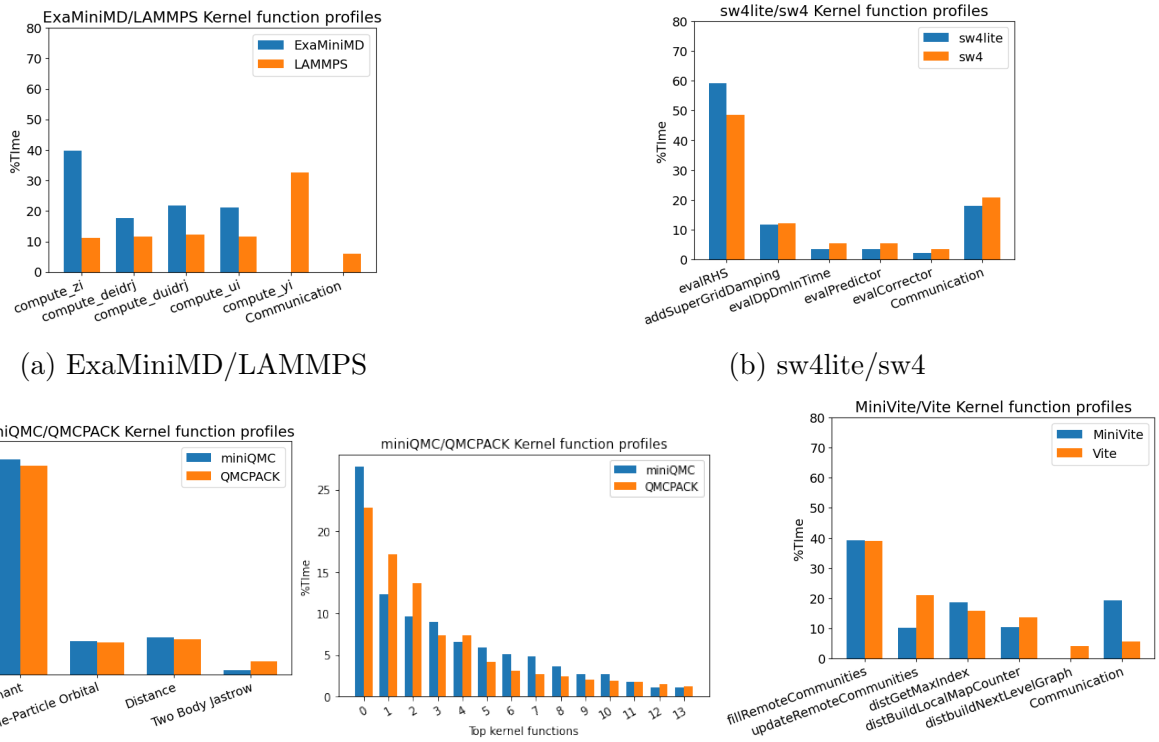
platforms. However, significant slowdowns were noted for others. Figure 3.5 visualizes cosine similarity for applications on Power9.

Overall, proxy/parent application pairs are more convergent on Power9 than on Skylake, except MiniQMC/QMCPack which show more divergence on Power9. We hypothesize that this divergence results from the improved memory subsystem in the Power9 processor. QMCPACK has undergone refactoring efforts to enhance memory efficiency, as evidenced by their changelog on GitHub. In contrast, MiniQMC shows no evidence of similar optimizations.

In summary, similar proxy/parent pairs maintain consistency across similarity algorithms, while dissimilarities are algorithm-dependent and may be exaggerated by different system environments. Since these distance methods yield consistent results for similar pairs with negligible runtime differences, we select cosine similarity for validating fidelity due to its simplicity, performance, and ease of interpretability via geometric angle. Unless otherwise specified, our subsequent analyses are based on the Skylake system.

3.4.2 Root Cause Analysis

To establish a process for determining ground truth and to further understand why our similarity algorithms find certain proxy/parent pairs similar or difficult to correlate, we investigate their code base implementations, particularly their kernel functions. Scientific applications consist of one or more kernel functions that collectively solve a certain scientific problem. We chose four proxy/parent pairs that cover a range of similarities: ExaMiniMD/LAMMPS, MiniVite/Vite, SW4lite/SW4, and miniQMC/QMCPack. Their key kernel profiles are illustrated in Figure 3.6a, 3.6b, 3.6c, and 3.6e, respectively. In each figure, the x-axis shows kernel function names, sorted by the normalized execution time (with some entries representing combinations of related kernel functions). The y-axis represents normalized execution time - the percentage of total



(a) ExaMiniMD/LAMMPS (b) sw4lite/sw4

(c) miniQMC/QMCPAC (d) miniQMC/QMCPAC details (e) MiniVite/Vite

Figure 3.6: Kernel Function Profiles of Proxy/Parent Pairs Sorted by Importance

application runtime spent in each kernel, excluding MPI communication time. We use the normalized kernel function execution time between the pairs rather than the clock time because the proxy is much smaller than the parent and thus executes more quickly. The normalized time, or time percentage, represents the ratio of a kernel's execution time compared to the total execution time of the application. Note that communication time is excluded, as it pertains to MPI communication rather than kernel execution. Kernels with zero value indicate their absence in a particular application.

LAMMPS, a classical molecular dynamics simulator, implements both short-range solvers and long-range particle interactions, while its proxy, ExaMiniMD, implements only short-range solvers. The dominant kernels are *compute_zi* and *computer_yi*. The kernel percentage time distributions are similar (Figure 3.6a), validating the 4° cosine similarity between the pair as indicated in (Figure 3.3). This high similarity suggests

that ExaMiniMD effectively models the short-range solver behavior of LAMMPS, making it a reliable proxy for optimizing these aspects of the parent application. However, its effectiveness may be limited when considering long-range particle interactions.

SW4lite is developed using the same code base as SW4, so it is unsurprising that most functions report close execution time percentages, especially the most significant functions, *evalRHS*. The kernel percentage time distribution is almost the same (Figure 3.6b), supporting the 2° similarity between SW4lite and SW4 **Calder** reports in Figure 3.3. This extremely high similarity indicates that SW4lite is a highly effective proxy for SW4, accurately representing its computational characteristics across all major kernels.

miniQMC and QMCPack are both quantum Monte Carlo packages for computing the structure of atoms. Although the four key kernel profiles are similar (Figure 3.6c), the function kernels within each kernel diverge (Figure 3.6d). This deeper divergence leads to the relative 11° similarity between miniQMC and QMCPAC (Figure 3.3). While miniQMC captures the overall behavior of QMCPack reasonably well, the differences in function kernels suggest that it may not accurately represent some of the finer-grained computational characteristics of the parent application. This could impact its effectiveness as a proxy for certain detailed performance optimizations.

MiniVite and Vite are both implementations of the Louvain method for graph clustering. Vite has slightly larger kernel times in the short-time functions compared to MiniVite. Additionally, Vite has an extra function for traversals between multiple graph levels that do not exist in MiniVite. The divergence in Figure 3.6e supports the similarity score of 35° between MiniVite and Vite (Figure 3.3). This significant difference in kernel-level behavior indicates that MiniVite may not be a highly effective proxy for Vite in all scenarios. It likely captures some of the core computational characteristics but misses important aspects related to multi-level graph traversals. This could lead to inaccurate performance predictions when using MiniVite to op-

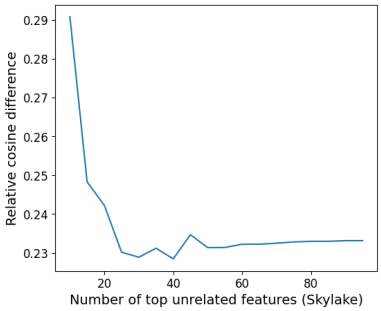


Figure 3.7: Relative difference of Cosine Similarity between top unrelated features and all features (Skylake)

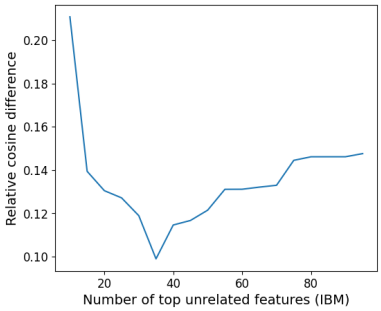


Figure 3.8: Relative difference of Cosine Similarity between top unrelated features and all features (IBM)

timize systems for Vite, particularly for workloads that heavily utilize the missing functionality.

To establish a robust ground truth for comparison, kernel-level analysis of proxy/parent pairs offers the most accurate foundation. Examining four such pairs, three demonstrate significant similarity in their function kernels, except MiniVite/Vite, which achieves only a 35% cosine similarity score. Given that the cosine similarity metric ranges from 0° (identical) to 90° (dissimilar), we set a similarity threshold of 30° for this analysis. This threshold suggests that MiniVite/Vite exhibits a notably lower degree of similarity than the other three pairs. This analysis underscores the importance of carefully evaluating proxy applications, as even those designed to represent specific parent applications may have limitations in accurately modeling all aspects of the parent's behavior.

3.4.3 Feature Selection and Feature sensitivity

To simplify the data collection for future performance similarity analysis, we sought to identify a concise subset of features that preserve the similarity of proxy/parent pairs compared to using all features (§ 3.2.2). Employing the Laplacian score algorithm, we ranked the features and subsequently fed them to the correlation filter to capture the top uncorrelated features. For the Laplacian score calculation, we set the neighbor size

parameter to 2, based on the assumption that each proxy/parent pair shows similar performance. After removing the correlated features through the correlation filter, we obtained a ranked set of 89 uncorrelated features.

To assess the robustness of our feature selection method, we conducted a sensitivity analysis by varying the number of top features selected. The similarity matrices remained largely stable across different feature subsets, with the most significant changes observed when using more than 20 features. The selection of 25 features for Skylake represents an optimal balance point where additional features provided diminishing returns in similarity preservation.

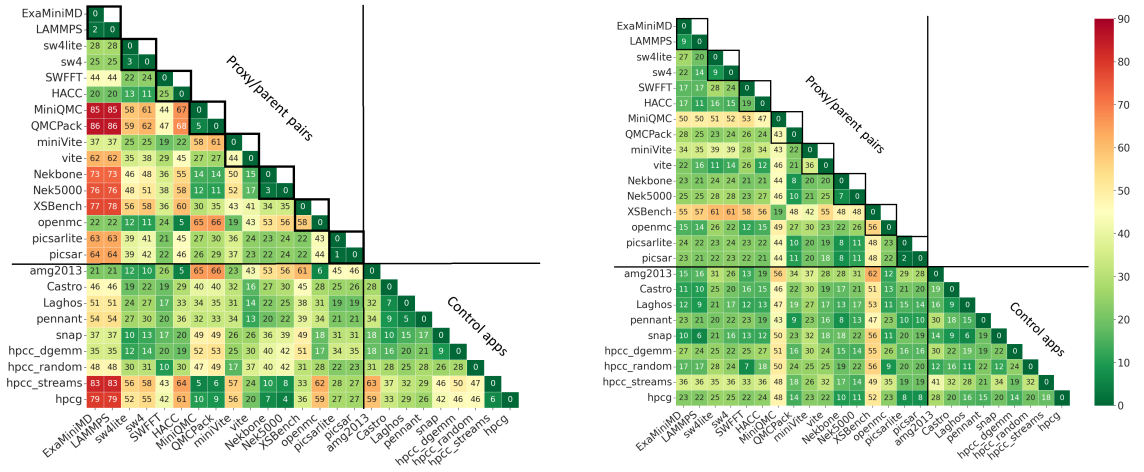
As illustrated in Figure 3.7, we ultimately selected the top 25 uncorrelated features to compute the similarity matrices for application pairs. Table 3.3 provides examples of top unrelated features for Skylake. Notably, certain features present in Figure3.2 are absent from Table3.3 due to their removal by the correlation filter.

Using only the top 25 features (Figure 3.9a) provided a more semantically interpretable explanation of the similarity scores for application pairs compared to using all 500 features (Figure 3.3). Figure 3.9a retains all 2×2 black-bordered blocks exhibiting high similarity between proxy and parent applications, while applications without a proxy/parent relationship remain dissimilar. With a similarity threshold of 30, 75% of the proxies in our suite demonstrate highly convergent behavior to their parents, except for MiniVite/Vite and XSBench/OpenMC.

Figure 3.9b illustrates the similarity scores of application pairs on the Power9 system using the top 35 features. Although the quantitative distances vary, the relative similarity between most proxy/parent pairs is preserved, similar to the results obtained on Skylake. However, Power9 requires selecting a larger number of top features (Figure 3.8) to capture the full spectrum of similarity information. Our sensitivity analysis revealed that Power9 results stabilized at around 35 features, showing minimal impact on the similarity matrices. This higher feature requirement

Table 3.3: Sample Top Features for Skylake

Rank	Hardware event count names
1	MEM_LOAD_UOPS_L3_HIT_RETIRED:XSNP_NONE
2	OFFCORE_REQUESTS:DEMAND_DATA_RD
3	UOPS_EXECUTED:THREAD
4	OFFCORE_REQUESTS_OUTSTANDING:ALL_DATA_RD
5	OFFCORE_REQUESTS_BUFFER:SQ_FULL



(a) Cos Similarity, Top 25 Features, Skylake (b) Cos Similarity, Top 35 Features, Power9

Figure 3.9: Important Features

is likely due to the richer set of hardware counter features available on Power9.

3.4.4 Feature Standard Deviation

In addition to selecting important features that preserve the similarity of proxy/parent pairs, we also investigated the factors contributing to dissimilarity. We analyzed the runtime time series by calculating the standard deviation (ω) of each feature for each parent application, using hardware event counts per second. We then checked whether the accumulated mean of the corresponding feature in the proxy application falls within two standard deviations of the parent's accumulated mean, assuming a normal distribution. While this is an approximation, it suffices for the granularity of our analysis.

Table 3.4: Dissimilarity Feature Source for Proxy/Parents Pairs

Proxy and Parent pairs	**>2std**	**>3std**	**>4std**	**>5std**
ExaMiniMD / LAMMPS	10	8	8	4
SW4lite / SW4	1	1	1	1
SWFFT / HACC	17	12	11	8
miniQMC / QMCPACK	13	10	8	6
miniVite / Vite	72	38	23	21
Nekbone / Nek5000	11	6	2	2
XSbench OpenMC	16	9	9	8
PICSARlite / PICSAR	1	1	1	0
Unique feature #s	99	64	49	38

Table 3.4 presents the number of features for each proxy/parent pair where the difference exceeds 2, 3, 4, or 5 standard deviations. The results align with our expectations, as shown in Figure 3.3. SW4lite/SW4 and PICSARlite/PICSAR are the most similar proxy/parent pairs, with only one feature in each pair deviating significantly from the parent (*'MOVE_ELIMINATION: SIMD_NOT_ELIMINATED'* and *'UOPS_EXECUTED: X87'*, respectively). Proxies with more features deviating beyond 2 standard deviations exhibit greater dissimilarity. For example, MiniVite and Vite show 72 features with large deviations, resulting in a moderate similarity of 35° in cosine similarity, as illustrated in Figure 3.3.

This standard deviation analysis provides actionable insights by identifying specific hardware events where their proxy applications deviate significantly from the parent applications. For instance, if a proxy shows large deviations in branch misprediction events, developers can focus on optimizing control flow patterns to better match the parent application. Researchers can establish concrete thresholds, such as maintaining critical hardware events within 2-3 standard deviations of the parent application's mean, and verify these thresholds across different architecture families.

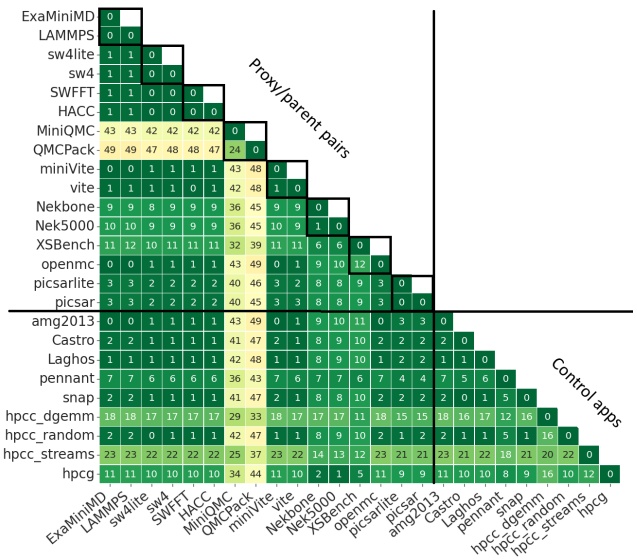


Figure 3.10: Cosine Similarity for L1 Cache

3.4.5 Subgroup Features

In addition to examining the overall features, we also investigate the similarity within subgroups and explore notable behaviors that influence proxy selection(§ 3.3.3). For example, when selecting proxy applications for a memory performance study, it is crucial to consider the diversity of memory behaviors among the candidate proxies. Conversely, if the objective is to refactor code for improved memory performance, minor discrepancies between the proxy and parent application in memory behavior may be negligible.

We analyze various subgroups of data, such as Branch, Instruction Mix, Instruction Cache, and L3 Cache. Some subgroups reveal a high degree of similarity, while others show applications that are outliers within those subgroups. Some matrices show extensive dissimilarities. For instance, Figure 3.10 illustrates similarity within L1 cache-related performance counters. MiniQMC/QMCPack exhibit a cosine similarity that approaches our upper threshold, making them relatively similar to each other and dissimilar to other applications in this subgroup. The observed low similarity in L1 cache behavior between MiniQMC and QMCPack is due to the code modifications

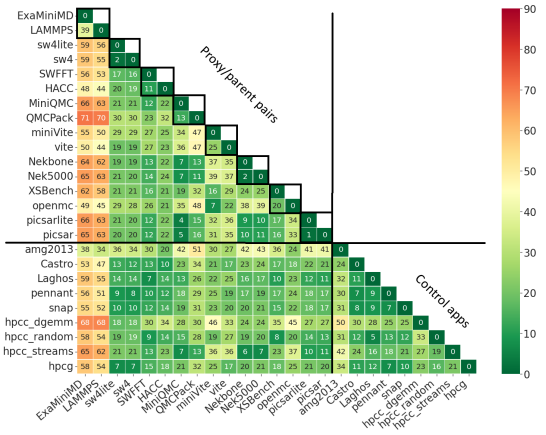


Figure 3.11: Cosine Similarity for Memory Pipeline

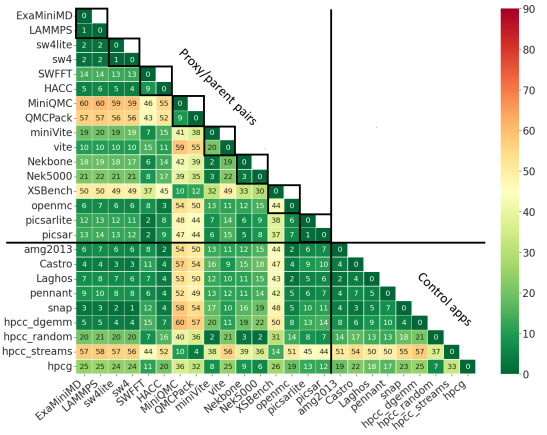


Figure 3.12: Cosine Similarity for Execution Pipeline

in QMCPack that are not reflected in MiniQMC, altering the memory and cache behaviors [60]. **Calder** can identify such changes, making it easier for proxy developers to observe the behavior differences, address code modifications, and create more accurate proxies.

Outlier applications display inconsistencies across various subgroups. For example, in the memory pipeline subgroup (Figure 3.11), ExaMiniMD/LAMMPS show dissimilarity from others, which may be because solving sparse matrix equations requires less back-end memory than other operations. In the Execution Pipeline subgroup (Figure 3.12), MiniQMC, QMCPack, XSBench, and HPCC_streams exhibit divergent behaviors from all other applications. In conclusion, investigating subgroups provides a valuable method for identifying similarities and differences in specific areas of application behavior.

3.4.6 Evaluation on Network Counters

To demonstrate the generalizability of **Calder**, we examine the similarity of proxy and parent applications using the performance of internal network counters. We chose the Cray Aries network for this evaluation, as it provides counters that reveal the network behavior of MPI applications. Using the LDMS Aries latency and bandwidth

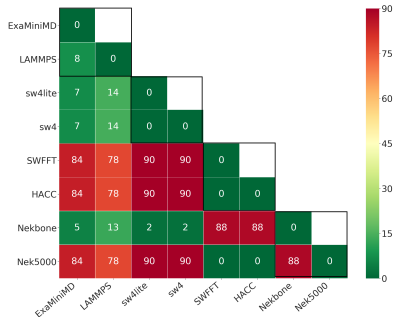
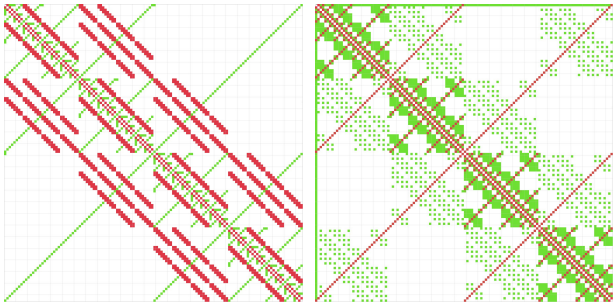


Figure 3.13: Aries lbw cosine similarity



(a) Nekbone (b) Nek5000

Figure 3.14: Network Point-to-Point Communicating

sampler, we collect the Aries performance counters and calculate the maximum and minimum response times for outstanding requests to the Aries network interface card (NIC). Additionally, we count the number of bytes sent and received by the NIC.

The results in Figure 3.13 show the similarity matrix for pairs. ExaMiniMD and LAMMPS, SW4lite and SW4, and SWFFT and HACC perform similarly, while Nekbone and Nek5000 show significant divergence. To further investigate this divergence, we collected pairwise communication patterns using CrayPat [6] for all application pairs. These patterns include peer-to-peer statistics that capture point-to-point communication between MPI processes and the total number of calls between them. The data records messages sent from a specific source process/rank to a specific destination process/rank.

Figure 3.14 indicates that the communication patterns of Nekbone and Nek5000 diverge, with approximately 68% of Nekbone's communicating pairs present in Nek5000, accounting for 58% of Nekbone's total communication. The MPI point-to-point data for the other proxy/parent pairs also exhibit similar patterns that agree with the Aries similarity matrix outcome in Figure 3.13.

3.5 Discussion

In this work, we show that the particularities of the similarity algorithms have less impact than predicted on the likelihood of correlating proxy/parent pairs. We posit that this is because HPC proxy/parent applications share core functionalities, even if they are implemented differently. This shared core behavior may lead to similar performance counter profiles, which are then captured by various algorithms. By comparing the similarity of subgroup features, we can identify and select proxies that represent the desired subgroup features of the parent.

Overall, our work provides a comprehensive, quantitative characterization of proxy application fidelity at the hardware level, using statistical similarity algorithms across a large suite of proxy/parent application pairs. The **Calder** toolkit and data collection infrastructure, which are publicly available for similarity measurements and collection, are actively used and deployed on HPC production systems. This research is crucial for advancing HPC and other domains by ensuring the reliability and efficiency of proxy applications.

In this work, we use kernel analysis in the codebase as ground truth to select the threshold for cosine similarity. While this approach has proven effective, establishing more robust ground truth metrics remains an open challenge. Our current methodology also has several limitations. First, our experiments are based on a single input for each application, which may not capture the full range of application behaviors. Second, the approach may face scalability challenges when dealing with larger application suites or more complex hardware configurations that generate more performance counters.

For future work, input sweep experiments using various inputs are the next step for a more comprehensive evaluation of our methodology. While we focused on proxies that represent node behavior, **Calder** is generalizable and can accept *any data* that it can preprocess and vectorize. We plan to apply **Calder** beyond HPC workloads to compare the I/O behavior of appropriate proxies to their parent applications, as well

as to applications running on GPUs.

Chapter 4

SimPoint++: Advanced Sampled HPC Application Simulation

4.1 Overview

SimPoint [109] is a widely adopted technique for simulation acceleration in computer architecture. It uses statistical sampling and clustering to capture a small set of simulation points to represent the complete execution of a program for efficient and accurate simulations. SimPoint has been integrated into numerous tools (*e.g.*, PinPoint[92], BarrierPoint [22], and LoopPoint [104]) and simulation frameworks (*e.g.*, Gem5 [75] and Sniper [21]), revolutionizing microprocessor simulation and performance analysis. However, the most recent version (SimPoint 3.0) from 2006 lacks advanced dimension reduction and clustering algorithms. Furthermore, SimPoint often identifies more representative points than necessary, increasing simulation time.

We introduce **SimPoint++**, an improved version of SimPoint to address these limitations. As shown in Fig. 4.1, **SimPoint++** inherits the architectural principles of SimPoint, featuring two updated components—dimensional reduction and clustering—which are highlighted with a gray background in the workflow. **SimPoint++** employs

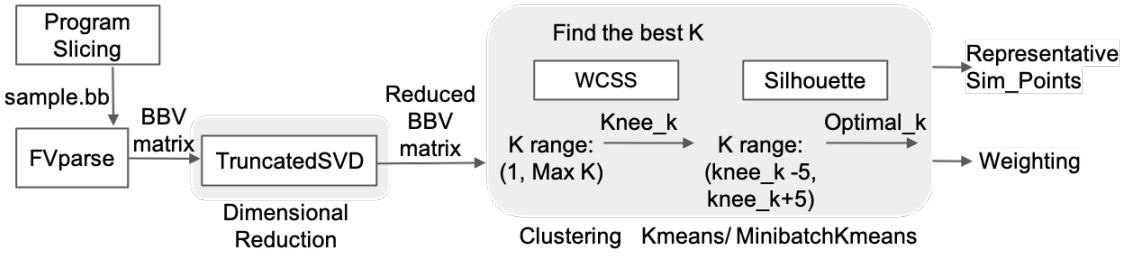


Figure 4.1: Workflow of **SimPoint++**

the within-cluster sum of squares (WCSS) and silhouette score to estimate and fine-tune the range of cluster number K values, replacing the less effective Bayesian Information Criterion (BIC) method. The new Python framework of **SimPoint++** also provides a dimension reduction pipeline for effective clustering and supports multi-thread application analysis. We evaluate **SimPoint++** with SPEC CPU 2017 benchmarks. **SimPoint++** achieves comparable or higher accuracy with significantly fewer simulation points, resulting in a 5x speed-up in simulation time compared to state-of-the-art solutions.

SimPoint++ offers a more robust and adaptive method for determining the optimal number of clusters, which is crucial for balancing simulation time and representativeness. It potentially enhances the efficiency and effectiveness of computer architecture simulation and analysis.

4.2 Background

4.2.1 Original SimPoint Workflow

SimPoint is based on the observation that programs often exhibit repetitive behavioral patterns over time, with program behavior directly linked to the code executed during specific intervals. When intervals show similar code patterns or "fingerprints", they typically have comparable performance characteristics and can be represented by a single sample. By identifying these representative patterns and selecting one

sample point from each, SimPoint enables rapid sampling that accurately reconstructs the program's entire execution process, significantly reducing simulation time while maintaining high accuracy in performance analysis.

The SimPoint process involves several steps: First, the program is sliced into chunks with the same time interval, and Basic Block Vectors (BBVs) are generated for each chunk to summarize its behavior. Random Projection is then applied to the BBVs to reduce dimensionality and limit computational complexity. Next, the K-means algorithm clusters the reduced BBVs, grouping similar program regions. Each cluster selects a region as a representative Sim_Point, effectively encapsulating the program's behavior. Finally, each Sim_Point is assigned a weight representing the relative frequency of its corresponding behavior in the overall program execution.

While this workflow has proven effective for many applications, it has limitations when dealing with increasingly complex modern HPC workloads. The following subsections delve deeper into two key components of SimPoint: Random Projection and K-means clustering. We examine their roles in the SimPoint process, their theoretical foundations, and their potential limitations. This analysis provides essential context for understanding how **SimPoint++**'s innovations overcome these limitations to better serve modern HPC simulation needs.

4.2.2 Random Projection

Random projection [28] is a dimensionality reduction technique that projects high-dimensional data onto a lower-dimensional space while approximately preserving the pairwise distances between points. The process is straightforward: multiply the dataset by a random matrix of size (original dimension $\times$ target dimension), with each matrix entry ranging between -1 and 1. The theoretical foundation for this method is provided by the Johnson-Lindenstrauss lemma[57]. Given the number of samples, this lemma establishes an upper bound on the number of target dimensions required to

preserve distances within a specified error margin. Specifically, it suggests choosing k $\approx \log(n)/\varepsilon^2$, where n is the number of data points and ε is the desired accuracy. For example, 1000 data points (n = 1000) and $\varepsilon = 0.2$: $k \approx \log(1000)/(0.2)^2 \approx 6.9/0.04 \approx$ 172. However, in practice, fewer dimensions may yield good results, especially when the data is sparse, which is the case for BBV. Notably, the lemma suggests that a single random projection is typically sufficient for dimension reduction.

Random Projection's computational efficiency and ease of implementation make it suitable for handling the large-scale data generated during program analysis. SimPoint uses 15 as the default reduced dimension number for SPEC CPU 2000 benchmarks. This choice represents a carefully considered balance between computational efficiency and accuracy.

However, using random projection in SimPoint has potential limitations. As applications and benchmark suites grow in size and complexity, the default dimension of 15 may be inadequate. This is because a significant amount of information might be lost during the dimensionality reduction process. Consequently, this loss of information could potentially result in less accurate phase detection. For example, in complex HPC applications with multiple interleaved computational phases, reducing dimensionality too aggressively might cause distinct phases to appear similar in the lower-dimensional space. This could lead to misclassification of program behaviors, resulting in less representative simulation points and reduced accuracy in performance estimation.

The limitations of random projection in SimPoint, particularly for modern, complex applications, highlight the need for more advanced dimensionality reduction techniques in SimPoint++, which can better preserve the intricate behavioral patterns of HPC workloads.

4.2.3 K-means

K-means is an unsupervised machine learning algorithm used for clustering data points into K groups based on their similarity. The algorithm operates iteratively, starting with randomly initialized cluster centroids. It first assigns each sample to the nearest cluster center and then updates these centers to the mean of all samples in each cluster. This process continues until convergence or until a maximum number of iterations is reached.

In SimPoint, K-means is employed to cluster program intervals with similar BBVs, identifying program phases. The algorithm uses the projected interval vectors obtained from random projection as input for efficient clustering in the reduced-dimensional space. However, determining the optimal number of clusters (K) is non-trivial and the initial centroid placement can lead to inconsistent results across different runs. SimPoint addresses these challenges by running K-means multiple times with varying values of K and using a scoring metric Bayesian Information Criterion(BIC) to select the best clustering.

One potential limitation of K-means is its tendency to create clusters with relatively similar spatial extent in each dimension, which may not always accurately represent the structure of program phases. This could lead to suboptimal clustering in program phases with significantly different scales or complex, non-complex shapes. Moreover, the sensitivity of K-means to initial centroid placement can result in inconsistent clustering across different runs, potentially affecting the reproducibility of simulation results.

To address these limitations, SimPoint++ considers alternative initialization methods such as K-means++. K-means++ improves centroid initialization by selecting initial centroids that are well-spread across the data space. In the context of program behavior clustering, this can lead to more stable and accurate clusters, especially for applications with diverse phase behaviors. By improving the initial placement of

centroids, K-means++ can help SimPoint++ more consistently identify representative program phases, enhancing the overall accuracy and reliability of the simulation process.

4.2.4 Why do we need to replace BIC in SimPoint?

The Bayesian Information Criterion (BIC) is a statistical measure for model selection, balancing model likelihood with complexity. SimPoint employs BIC to determine the optimal cluster number K. However, the SimPoint authors identified limitations in directly applying BIC to their clustering approach. They observed that BIC scores tend to increase with the number of clusters, which could lead to selecting the maximum possible K.

To address this, they developed an alternative method: instead of choosing the maximum BIC score, they use binary search to select a clustering that achieves 90% of the BIC score range, identifying a point beyond which the score increases only marginally. The problem of determining the best K is then transferred to choosing an appropriate MAX K, which is set by the researcher at the beginning of the process as a hyperparameter. The authors provide complex recommendations for scenarios requiring increased accuracy, such as reducing interval size and adjusting MAX K (300 or the square root of the total interval count). For users prioritizing accuracy, they suggest that if SimPoint selects a cluster count near the MAX K, it might indicate that the MAX K is insufficient to capture all unique behaviors, recommending doubling the MAX K and rerunning the analysis.

However, the 0.9 threshold, while shown to minimize IPC variance within clusters, may not be universally optimal across all programs or architectures. Furthermore, the method's dependency on MAX K may not always reflect the true underlying structure of the data, and it could potentially overlook more optimal solutions beyond the chosen MAX K. To select simulation points representing the top percent of

execution, SimPoint 3.0 offers a Coverage option, allowing users to choose only the largest clusters that constitute the majority of program's weight. While this strategy could reduce the number of clusters, it is ineffective when cluster sizes are similar.

Despite all these efforts, they do not fully resolve the underlying issues. The fundamental problem lies in BIC's assumption of a Gaussian distribution, which is often not applicable to program behavior data. Program behavior distributions are diverse, with varying characteristics across clusters and features. This deviation from the Gaussian assumption can lead to misinterpretation of cluster characteristics and makes BIC difficult to use as a reliable criterion for program behavior problems.

To illustrate this limitation, consider a program that exhibits three distinct phases: initialization, computation, and finalization, each with unique durations and characteristics. The BIC criterion, with its Gaussian assumption, might fail to distinguish these phases effectively, potentially creating unnecessary subdivisions within the longer computation phase. With a maximum cluster limit MAX K of 20, BIC might identify 15 clusters even though there are only 3 distinct computational phases. This over-clustering not only increases simulation time but also reduces the representativeness of each simulation point.

This example underscores the need for a more effective method for determining the optimal K that doesn't rely on potentially arbitrary thresholds or MAX K values, and that can adapt to the diverse and often non-Gaussian nature of program behavior distributions. This is why **SimPoint++** introduces the use of WCSS and silhouette scores, which provide a more robust and adaptive approach to identifying the optimal number of clusters across a wide range of program behaviors.

4.2.5 The Process of How SimPoint Finds the Optimal K

Let's work through the process of how SimPoint uses Binary search to find the optimal K value. We use the example program demo-matrix with 8 threads. First,

Table 4.1: Example of finding best K in Simpoint

		Trial 1	Trail2	Trial 3	Trial 4	Trial 5
Run 1	k=1	**25099**	25099	25099	25099	25099
Run 2	k=20	32103	29335	29931	32237	**32499**
Run 3	k=10	**30381**	30235	27924	30257	29870
Run 4	k=15	**32492**	32286	27584	31896	30434
Run 5	k=12	**31929**	30906	29950	31105	30260
Run 6	k=11	29609	30358	30357	**30579**	29796

we concatenate the 8 threads into a single BBV for each sample. Then, we apply SimPoint with a MAX K value of 20. Simpoint employs binary search to find the best K in at most log(MAX K) runs. Each run consists of 5 trials with different initializations for K-means clustering at a certain K value, The highest BIC score among the 5 trials is selected to be the final BIC score for this run. The process begins with two runs: $K = 1$ and $K = MaxK$. These runs establish the range of the BIC score. We then calculate a threshold using the formula:

$$\text{Threshold} = \text{BIC}_{K1} + (\text{BIC}_{Kmax} - \text{BIC}_{K1}) \times 0.90. \tag{4.1}$$

The binary search continues, evaluating K values between 1 and MAX K. The search concludes when we find the smallest K value that produces a BIC score exceeding the calculated threshold. This K value is considered the optimal K for the given program and thread configuration.

Table 4.1 illustrates the process of finding the optimal K value. In each run, the trial with the highest BIC score is highlighted in bold. Run 1 (K=1) and Run 2 (K=20) establish the minimum and maximum BIC values, respectively, defining the

BIC range. Using these values, we calculate the threshold as follows:

$$\begin{aligned} \text{Threshold} &= \text{BIC}_{K1} + (\text{BIC}_{K20} - \text{BIC}_{K1}) \times 0.90 \\ &= 25099 + (32499 - 25099) \times 0.90 \\ &= 31759 \end{aligned} \tag{4.2}$$

In this example, the binary search process involves six runs. When K=12, we obtain a BIC score of 31926 (in red), which exceeds the calculated threshold of 31759. Therefore, we determine that the optimal K value is 12. Based on this result, we proceed with clustering, generate simulation points (Sim_Points), and calculate corresponding weights using K=12 as the number of clusters.

However, this demo application addresses a relatively simple problem, and the maximum K value (MAX K) is set too high. This results in the algorithm selecting a best K value that is significantly larger than the truly optimal K. The 78 data points in this example can be adequately described using fewer than 12 cluster centers. This suggests that the chosen K value may be unnecessarily large for the given dataset, potentially leading to a bigger optimal K.

4.3 Method

4.3.1 Dimension Reduction

SimPoint++ employs Truncated SVD, aka Latent Semantic Analysis LSA [54] for dimension reduction. Truncated SVD approximates a high-dimensional matrix using a lower-rank representation. It works by performing a standard SVD decomposition on the input matrix, then only retaining the top k singular values and their corresponding singular vectors. This process effectively creates a compressed version of the original data that captures its most important features or patterns. By discarding the smaller

singular values, Truncated SVD reduces noise and focuses on the most significant components of the data. Given a matrix $A \in \mathbb{R}^{m \times n}$, the Truncated SVD of rank k (where $k < \min(m, n)$) can be expressed as:

$$A \approx A_k = U_k \Sigma_k V_k^T$$

Where A_k is the rank-k approximation of A, $U_k \in \mathbb{R}^{m \times k}$ contains the first k left singular vectors, $\Sigma_k \in \mathbb{R}^{k \times k}$ is a diagonal matrix containing the k largest singular values, $V_k^T \in \mathbb{R}^{k \times n}$ contains the first k right singular vectors (transposed). The singular values in Σ_k are arranged in descending order.

Comparison with other methods

1. PCA is a linear dimensionality reduction technique that identifies the directions (principal components) along which the data varies the most. PCA can be expressed as:

$$X\_reduced = XW,$$

where X is the original data matrix, W is the matrix of principal component loadings, and $X_{reduced}$ is the reduced-dimension data. PCA and Truncated SVD are closely related. When the data is centered (mean-subtracted), PCA is equivalent to Truncated SVD. However, Truncated SVD is more general and can be applied to non-centered data.

2. t-SNE is a non-linear technique that aims to preserve local structure by minimizing the KL divergence between the probability distributions of pairwise similarities in high-dimensional and low-dimensional spaces. It uses a momentum-based gradient

descent to minimize its cost function. The cost function is:

$$C = KL(P||Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}},$$

where p_{ij} and q_{ij} are the pairwise similarities in high and low dimensions, respectively. While t-SNE excels at preserving local structure and creating visually appealing embeddings, it has limitation in scalability and global representation.

3. Uniform Manifold Approximation and Projection (UMAP) [81] is a non-linear dimensionality reduction technique based on manifold learning and topological data analysis. Using stochastic gradient descent, UMAP optimizes the layout of the low-dimensional embedding by minimizing the cross-entropy between the high-dimensional and low-dimensional sets. UMAP balances local and global structure preservation. Its objective function can be expressed as:

$$\text{UMAP Loss} = \sum_{i \neq j} \left(v_{ij} \log \left(\frac{v_{ij}}{w_{ij}} \right) + (1 - v_{ij}) \log \left(\frac{1 - v_{ij}}{1 - w_{ij}} \right) \right),$$

Where v_{ij} represents the edge weights in the high-dimensional space and w_{ij} represents the edge weights in the low-dimensional space. While UMAP is faster than t-SNE, it's generally slower than Truncated SVD, especially for very large datasets.

In conclusion, Truncated SVD is specifically designed to handle large, sparse matrices efficiently. It works directly with sparse matrix formats and avoids computing the full covariance matrix, making it particularly suitable for high-dimensional, sparse data often encountered in program behavior analysis. Unlike Random Projection or t-SNE, Truncated SVD produces deterministic results, ensuring reproducibility of the analysis. The singular vectors in Truncated SVD can be interpreted as directions of maximum variance in the data, providing insights into the underlying structure. Truncated SVD scales well to large datasets, which is crucial for analyzing extensive program behavior data. By focusing on the most significant singular values, Truncated

SVD naturally reduces noise in the data. Compared to t-SNE and UMAP, Truncated SVD is computationally more efficient, especially for large datasets. Therefore, Truncated SVD an ideal choice for reducing the dimensionality of our data while maintaining important structural information.

4.3.2 Optimized K-means Clustering

Clustering is a widely used technique for grouping similar data points. The five most widely used types are centroid models, distribution models, connectivity models, density models, and spectral methods. Centroid models (*e.g.*, K-means) use the distance between a data point and the cluster's centroid to group data. Distribution models (*e.g.*, Gaussian Mixture Model (GMM)) segment data based on their probability of belonging to the same distribution. Connectivity models (*e.g.*, Hierarchical clustering) use the closeness of data points to decide the clusters. Density models (*e.g.*, HDBSCAN) scan the data space and assign clusters based on the density of data points. Spectral clustering methods (*e.g.*, Normalized cuts) transform the data into a lower-dimensional space using eigenvectors of a similarity matrix before clustering.

Each type of clustering method has its own advantages and disadvantages. For our program behavior dataset, we do not want to set specific parameters or have an uncontrollable number of clusters. This rules out density models like DBSCAN and HDBSCAN, which require careful parameter tuning. Additionally, our data may not follow a specific distribution, making distribution models like Gaussian Mixture Models less suitable. Connectivity models, such as hierarchical clustering, are also less desirable as they can be computationally expensive, especially for large datasets. Spectral clustering, while powerful for capturing complex cluster shapes, can be computationally intensive for large datasets and requires careful selection of the similarity metric and the number of neighbors. Therefore, K-means clustering remains a suitable choice for our high-dimensional HPC data.

However, traditional K-means has limitations when dealing with large-scale, high-dimensional datasets typical in HPC environments. To address these scalability issues, **SimPoint++** adopts two key improvements: K-means++ initialization and MiniBatchKMeans.

Instead of the vanilla K-means used in SimPoint, **SimPoint++** adopts the K-means++ initialization scheme [94]. This method selects initial cluster centroids using sampling based on an empirical probability distribution of the points' contributions to the overall inertia. K-means++ significantly improves the speed of convergence and the quality of the final clustering solution, especially in high-dimensional spaces. By selecting initial centroids that are well-spread, K-means++ reduces the likelihood of poor local optima, which is particularly beneficial for complex HPC trace data.

When dealing with extremely large datasets, **SimPoint++** employs MiniBatchK-Means [108], a variant of K-means that processes data in mini-batches. This technique addresses scalability issues in high-dimensional HPC data analysis. MiniBatchKMeans significantly reduces computation time while still optimizing the same objective function as standard K-means. It achieves this by using small, random subsets of the data in each iteration, making it much more efficient for extensive trace analysis. This is crucial when dealing with the vast amounts of data generated in HPC environments, as it allows for timely analysis without compromising the clustering objective.

The combination of K-means++ initialization and MiniBatchKMeans provides a balance between clustering quality and computational efficiency, making it well-suited for modern HPC program behavior analysis.

Find the Best Cluster Number K Determining the optimal number of clusters is a critical step in clustering analysis. While various methods exist, combining multiple techniques can provide a more robust and comprehensive approach. In this case, we employ the combination of Within-Cluster Sum of Squares (WCSS) and Silhouette

methods.

Within-Cluster Sum of Squares (WCSS)

WCSS is a metric used to quantify the compactness of the clusters, with lower values indicating more compact clusters. It measures the sum of the squared distances between each data point and its assigned cluster center. it is defined as:

$$\text{WCSS} = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2$$

where k is the number of clusters, C_i is the i-th cluster, x is a data point in cluster C_i, and μ_i is the centroid of cluster C_i.

As the number of clusters (K) increases, the WCSS typically decreases, as each new cluster can better capture the variance in the data. The "elbow" point in the WCSS curve represents the point where the marginal benefit of adding more clusters starts to diminish, suggesting an optimal number of clusters.

Silhouette

The Silhouette score is a metric that evaluates both the cohesion and separation of clusters. For each data point i, the Silhouette score $s(i)$ is calculated using two main components: cohesion and separation. **Cohesion** $a(i)$ is the average distance from the data point i to all other points within the same cluster, reflecting the compactness of the cluster. **Separation** $b(i)$ is the minimum average distance from the data point i to all points in the nearest cluster that it is not a part of, indicating how distinct the clusters are. The Silhouette score for a single data point is then computed as:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}.$$

It has a value range between -1 and 1. A score close to 1 suggests that the data

point is well-clustered and far from other clusters, a score close to 0 indicates that the data point is on the boundary between clusters, and a negative score implies potential misclassification.

The average Silhouette score across all data points can be used to determine the optimal number of clusters K, with the highest average Silhouette score typically corresponding to the best clustering configuration.

WCSS + Silhouette

While WCSS effectively identifies the general region where the optimal number of clusters might lie, it can sometimes be ambiguous, especially when the elbow point is not clearly defined. On the other hand, Silhouette offers a more nuanced evaluation of cluster quality but can be sensitive to cluster shape and density, and computationally expensive when applied to a wide range of K values.

To combine WCSS and Silhouette with the KneeLocator method [106], we first use the WCSS curve to find the "elbow" or "knee" point using the KneeLocator. The KneeLocator tries to identify the point where the rate of change in the WCSS curve's slope is greatest, as this represents the point where the benefit of adding more clusters starts to diminish. We then narrow the search range for the optimal number of clusters K to the region around the knee point, specifically (`knee_k` - 5, `knee_k` + 5). Next, we compute the Silhouette score for each value of K in the narrowed search range and select the value of K that corresponds to the highest average Silhouette score within this range, which we denote as `optimal_k`.

By combining the WCSS and Silhouette metrics, along with the KneeLocator method, we can effectively determine the optimal number of clusters for our program behavior dataset. The WCSS curve helps identify the general region of the optimal K, and the Silhouette score provides a more refined evaluation to select the best clustering configuration. This approach can be beneficial when dealing with complex or noisy

data, where the elbow point in the WCSS curve may not be clearly defined, and the Silhouette score can help validate the optimal number of clusters and provide a more robust way to determine the best clustering solution.

4.3.3 Spectral Clustering

Spectral clustering is a graph theory-based clustering technique. It transforms the original dataset into a lower-dimensional space, the eigenvectors of the Laplacian matrix, revealing inherent patterns within datasets. This method is particularly beneficial when dealing with non-linearly separable clusters, where traditional methods like K-means might fail.

In **SimPoint++**, we explored spectral clustering as a potential alternative to K-means due to its ability to handle complex data structures. This exploration was motivated by the possibility that program behavior data might exhibit non-linear relationships that K-means could miss. Spectral clustering would be especially advantageous in scenarios where program phases form intricate, intertwined patterns in the feature space, rather than well-separated, convex clusters.

To implement the spectral clustering algorithm, we first need to construct the graph of the dataset. We use the function kneighbors_graph from scikit-learn to construct the k-nearest neighbors (KNN) graph. The number of neighbors for each point $n\_neighbors$ is the most important parameter we need to tune for spectral clustering. We set the closest $n\_neighbors$ points for each point to be its neighbors. We then symmetrize the graph by adding it to its transpose and dividing by 2, creating an undirected graph where an edge exists only if both points consider each other neighbors. This mutual step enhances robustness to variations in local density.

Next, we check whether the graph is fully connected and stop further calculation if the graph is disconnected, otherwise, the Laplacian matrix will have multiple zero eigenvalues, and the corresponding eigenvectors will not provide useful information

for clustering.

Then comes the eigen decomposition. To improve numerical stability and make the algorithm less sensitive to variations in graph structure, we calculate the normalized Laplacian [120] instead of the standard Laplacian. Thus, all eigenvalues are in the range $[0, 2]$. The normalized graph Laplacian L is defined as:

$$L = I - D^{-1/2} W D^{-1/2}, \tag{4.3}$$

where I is the identity matrix, W is the weighted adjacency matrix, and D is the diagonal degree matrix with $D_{ii} = \sum_j W_{ij}$.

We performs partial eigendecomposition for the k smallest eigenvalues using scipy.sparse.linalg.eigsh which speeds up the computation. To estimate the number of clusters, we employ the eigengap heuristic [26]. This method analyzes the eigenvalues of the Laplacian matrix, where the largest gap in sorted eigenvalues indicates a natural division in the data. The eigenvalues represent the "importance", or the amount of variation, in the data along different directions, and a large gap suggests a significant difference in the structure of the data before and after that dimension.

$$\text{n\_clusters} = \arg\max_i (\lambda_{i+1} - \lambda_i), \tag{4.4}$$

where λ_i are the eigenvalues of L in ascending order.

Once we get the cluster number $n\_clusters$, we construct a matrix $U \in \mathbb{R}^{n \times k}$ by using the $n\_clusters$ eigenvectors corresponding to the $n\_clusters$ smallest eigenvalues of the Laplacian matrix L as columns, where n is the number of data points. We then apply the k-means algorithm to cluster the rows of U, which represent the data points in this lower-dimensional eigenspace. This transformation often results in more easily separable clusters, as it leverages the spectral properties of the graph Laplacian to capture the underlying structure of the data.

Finding the appropriate number of neighbors $n\_neighbors$ for the k-nearest neighbors graph is crucial for the performance of spectral clustering as it affects the construction of the affinity matrix. Too few neighbors may result in a disconnected graph, while too many can obscure the local structure of the data. A common starting point is to set n_neighbors to the square root of the number of samples. We employ an adaptive approach according to the data size to determine the optimal $n\_neighbors$, with the searching range of $n\_neighbors$ like this:

$$n\_neighbors \in [\max(10, \sqrt{n}), \min(10\sqrt{n}, 160, n)]. \tag{4.5}$$

For each $n\_neighbors$, we construct the mutual k-nearest neighbors graph and perform spectral clustering. We evaluate the quality of clustering with the silhouette score. The silhouette score considers both the cohesion within clusters and the separation between clusters. The optimal $n\_neighbors$ is selected as:

$$n\_neighbors^* = \underset{n\_neighbors}{\arg\max} \left(\frac{1}{n} \sum_i s(i) \right) \tag{4.6}$$

While spectral clustering demonstrated its ability to handle complex data structures, it was ultimately not selected as the primary clustering method for SimPoint++ for several reasons:

1. Computational Complexity: The eigen decomposition step in spectral clustering can be computationally expensive for large datasets, which are common in HPC program behavior analysis. This could potentially slow down the overall simulation process, contradicting one of the main goals of **SimPoint++**.

2. Scalability: As the size of the dataset grows, the memory requirements for spectral clustering increase significantly, which could be problematic for analyzing very large program traces.

3. Parameter Sensitivity: The performance of spectral clustering is highly dependent on the choice of the similarity graph and its parameters (*e.g.*, n_neighbors). This sensitivity could lead to inconsistent results across different programs or execution environments.

4. Empirical Performance: In our tests, while spectral clustering performed well on certain datasets, it did not consistently outperform the optimized K-means approach (using k-means++ initialization and MiniBatchKMeans) across a wide range of HPC program behavior data.

In conclusion, while spectral clustering offers theoretical advantages for complex, non-linearly separable data, the practical considerations of computational efficiency, scalability, and consistent performance across diverse HPC workloads led us to favor the optimized K-means approach for SimPoint++. However, the insights gained from spectral clustering remain valuable and could inform future improvements or specialized applications of **SimPoint++** for particularly complex program behaviors.

4.4 Experiment

We conduct experiments on Cloudlab [32] using an x86 architecture node (c220g2). We use a diverse set of applications to demonstrate **SimPoint++**'s effectiveness and versatility across different computing scenarios. The selection of these applications is based on their relevance to current HPC and general-purpose computing environments.

We choose Graph500 as our serial application due to its importance in benchmarking graph-processing capabilities, which are increasingly relevant in big data and analytics workloads. Graph500 was simulated using the Gem5 simulator.

For multi-threaded applications, we select a demo application, Matrix-OMP, and four applications from SPEC CPU 2017 [4] with train input. SPEC CPU 2017 represents a standardized, industry-recognized benchmark suite that covers a wide

range of real-world application scenarios. The specific SPEC CPU 2017 applications are selected to provide a mix of compute-intensive and memory-intensive workloads, allowing us to test SimPoint++ across varied program behaviors. These multi-threaded applications were simulated using the LoopPoint [104] framework on the Sniper simulator, with passive wait policies and 8 threads.

The choice of these applications aligns with the broader goals of this dissertation, as discussed in Chapter 5. They represent a mix of traditional HPC workloads and emerging application areas, providing a comprehensive test bed for evaluating the effectiveness and efficiency of **SimPoint++** across diverse computational patterns.

LoopPoint is a sampling simulation framework for multi-threaded applications. Instead of using instruction counting as done in single-threaded applications, LoopPoint uses loop entries as slice boundaries. Each simulation region is then specified using a (PC, count) pair, which defines the starting and ending loop entries. LoopPoint applies standard SimPoint to cluster these regions and identify the representative Loop_Points.

While SimPoint only supports single-threaded data point clustering, LoopPoint concatenates vectors from each thread to a combined long vector and then provides the vector to SimPoint for further analysis. In **SimPoint++**, we first apply dimension reduction for each thread in parallel, then concatenate the reduced vectors together. This approach decreases the computation time and uses less memory, making it more scalable for systems with numerous threads or extended execution traces. Our method potentially preserves thread-specific characteristics better while providing a more computationally efficient solution for multi-threaded program analysis.

4.5 Results

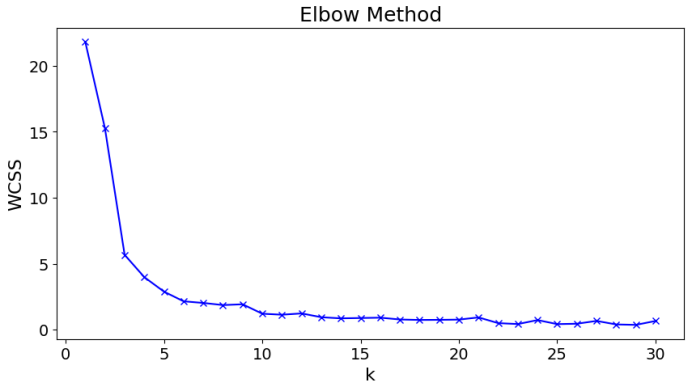
4.5.1 Finding the Best K

To illustrate how **SimPoint++** determines the optimal number of clusters, we use the Graph500 serial application as an example, with input parameters "-s 14 -e 14".

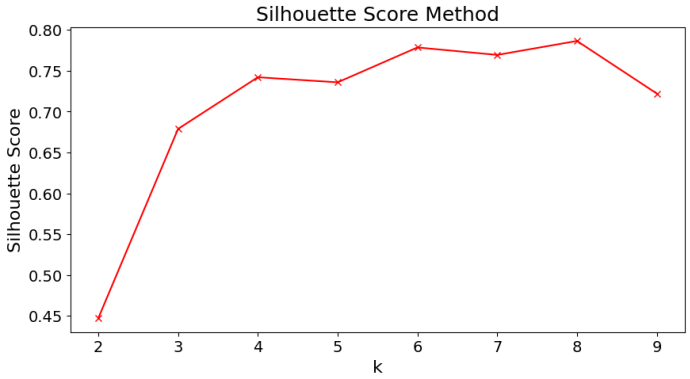
We employ the WCSS method to identify the knee point. The knee point represents the point of maximum curvature on the WCSS curve, where adding more clusters does not reduce much the within-cluster variance. For example, in Fig. 4.2a, as we increase the number of clusters from 1 to 20, the WCSS value decreases rapidly initially but then starts to level off. Using the kneedle algorithm [106], we identify this point of diminishing returns at `Knee_k` = 5. This suggests that the optimal number of clusters likely lies near this region, as additional clusters beyond this point provide minimal improvement in cluster cohesion relative to the computational cost.

The knee point helps improve efficiency by providing a focused range for our subsequent detailed analysis. Instead of calculating Silhouette scores for all possible cluster numbers (which could be computationally expensive), we concentrate on a narrow range around the knee point (Fig. 4.2b). Within this range, we select the number of clusters that yields the highest Silhouette score as the optimal K. In our example, this results in `optimal_k` = 8, which is only half of the `optimal_k`=16 that Simpoint provides based on the BIC criterion. With fewer simulation points, we can reduce the simulation time.

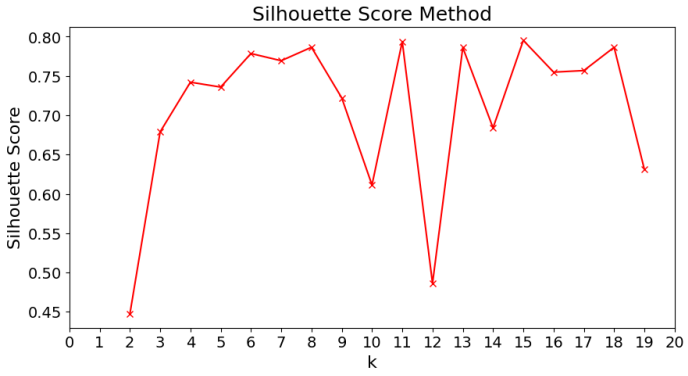
To validate the efficiency of our combined approach, we also compute Silhouette scores across a broader range of cluster numbers (2 to 20) without the WCSS-based restriction (Fig. 4.2c). This broader analysis reveals several cluster numbers with high Silhouette values. Notably, 8 remains the smallest among these high-scoring options, confirming that our WCSS-guided approach successfully identified an efficient solution while examining only a fraction of the possible cluster numbers. This demonstrates how



(a) WCSS curve showing diminishing returns in cluster cohesion beyond the knee point (`Knee_k`=5), indicating an efficient initial estimate for cluster count



(b) Silhouette analysis in the focused range around WCSS knee point, revealing `optimal_k` = 8 as the best balance between cluster separation and cohesion



(c) Extended Silhouette analysis (2-20 clusters) validating that `optimal_k` = 8 remains efficient when compared against a broader range of cluster numbers

Figure 4.2: Analysis of optimal cluster count using WCSS and Silhouette methods. The combination of these methods efficiently identifies the optimal number of clusters (k=8) while examining only a focused range of possibilities, demonstrating the effectiveness of our two-step approach in reducing computational overhead while maintaining clustering quality.

the knee point method effectively narrows down the search space without compromising the quality of the final clustering solution.

4.5.2 Speedup and Accuracy

SimPoint++ demonstrates superior performance compared to SimPoint, achieving higher speed-ups compared to full simulation (Fig. 4.4) with fewer simulation points. The average speed-up for **SimPoint++** is 109, which is more than five times the 20.7 speed-up achieved with SimPoint.

Furthermore, **SimPoint++** generally has similar or slightly lower error rates compared to SimPoint (Fig. 4.3). The overall absolute prediction error for **SimPoint++** is 1.93%, which is lower than the 2.65% error observed with SimPoint. This result is noteworthy given that **SimPoint++** uses fewer simulation points to encapsulate the full execution.

However, the application xz_s presents an interesting exception, showing a 6.83% absolute prediction error rate with **SimPoint++**, higher than the 4% error rate with SimPoint. This higher error rate may be due to the phase transitions throughout xz_s execution, making it more challenging for **SimPoint++**'s clustering algorithm to capture all behavioral variations with fewer simulation points. While SimPoint uses 19 simulation points for xz_s, **SimPoint++** selects only 2 points based on our optimization criteria, with WCSS finding `Knee_k` = 4 and Silhouette analysis yielding `optimal_k` = 2. This unusually low optimal cluster count, despite the application's complex behavior, suggests that the feature vectors of xz_s do not form well-separated clusters in the feature space. While the Silhouette score measures both cluster cohesion and separation, a high score doesn't always guarantee the best representation of an application's diverse behavioral patterns, especially when the underlying program phases exhibit gradual transitions rather than distinct boundaries. This limitation of the Silhouette metric in capturing the temporal aspects of program behavior may

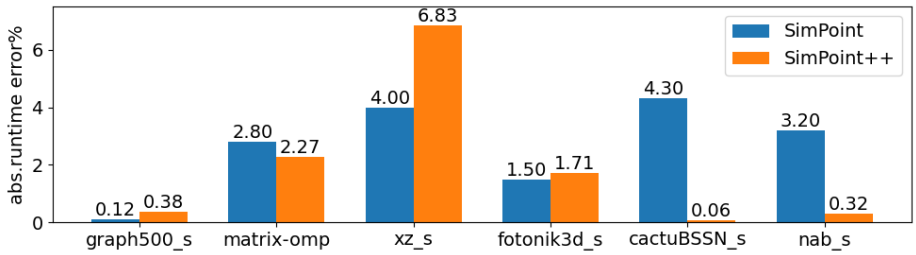


Figure 4.3: Absolute prediction error: SimPoint VS **SimPoint++**

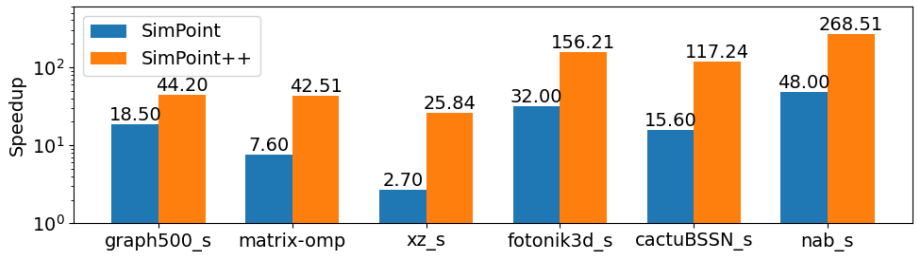


Figure 4.4: Speed up: SimPoint VS **SimPoint++**

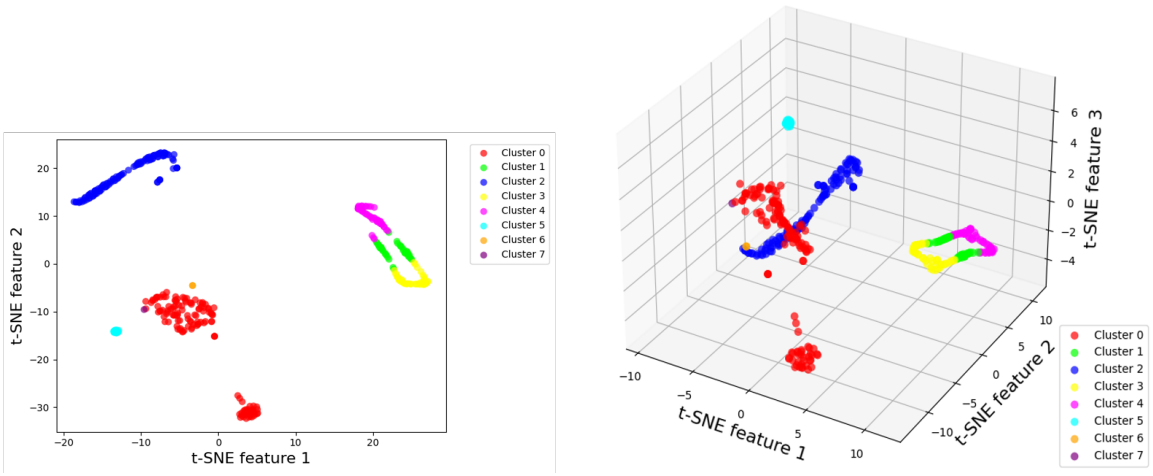
contribute to the higher error rate we observe.

We conducted additional experiments increasing the number of simulation points for xz_s to 5, which reduced the error rate to 4.5%. This suggests that for applications with complex phase behavior like xz_s, allowing for more simulation points could help mitigate the accuracy trade-off. However, this would come at the cost of reduced simulation speed-up. This trade-off between accuracy and simulation speed highlights the importance of considering application-specific characteristics when applying sampling techniques.

4.5.3 Comparison with Spectral Clustering

Since Spectral Clustering is more computationally expensive compared to K-means, we only have some preliminary results. We still use the Graph500 serial application as an example. We visualize the clustering results in both t-SNE 2D and 3D to provide a direct insight into the clustering performance.

Fig 4.5a and Fig 4.5b show the optimized K-means clustering, while Fig 4.6a and



(a) t-SNE 2d (b) t-SNE 3d

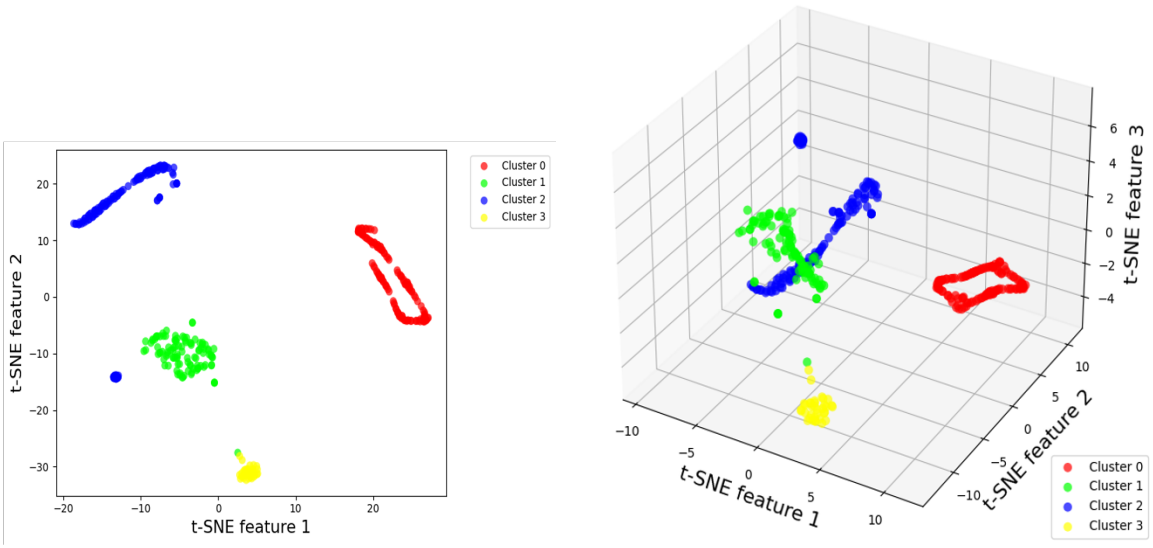
Figure 4.5: Visualization with t-SNE in 2D and 3D when using optimized K-means clustering

Fig 4.6b show the spectral clustering results. While Spectral clustering provides more reasonable clustering when the data point groups are in complex shapes, it achieves similar accuracy to K-means, 0.3871% and 0.3874% respectively. Although K-means does not perfectly group neighboring points, it still provides relatively good results. In future work, further experiments with more applications are needed to tune the adaptive approach and find the optimal n_neighbors value for spectral clustering in different scenarios.

4.6 Discussion

In this chapter, we conducted a thorough investigation of the standard SimPoint methodology, uncovering limitations in its approach to determining the optimal number of clusters. Our analysis revealed opportunities for improvement, particularly in the areas of dimension reduction and cluster number optimization.

Building upon these insights, we introduced **SimPoint++**, a novel approach that addresses the limitations of SimPoint. **SimPoint++** employs enhanced techniques for dimension reduction and cluster number optimization, resulting in significantly fewer



(a) t-SNE 2d (b) t-SNE 3d

Figure 4.6: Visualization with t-SNE in 2D and 3D when using spectral clustering

clusters compared to SimPoint while maintaining comparable or improved accuracy across most applications.

While **SimPoint++** shows promising results, we acknowledge that performance can vary across different types of applications. This variability underscores the complexity of workload sampling and the need for continued research in this area. Several potential directions for future improvement and expansion of **SimPoint++** include:

- **Alternative Clustering Algorithms:** As program behaviors become more complex and feature dimensions increase, exploring density-based clustering algorithms like DBSCAN could better handle non-spherical cluster shapes and varying cluster densities.

- **Extended Application Domains:** While our current focus has been on HPC workloads, **SimPoint++** could be adapted for emerging workloads such as machine learning training and cloud computing applications. This would require investigating additional program features and potentially modifying the

clustering criteria.

- **Dynamic Adaptation:** Implementing an adaptive mechanism that automatically adjusts the number of simulation points based on program complexity and desired accuracy-speed trade-offs could enhance **SimPoint++**'s versatility.

The potential reduction in simulation time offered by **SimPoint++**, combined with its comparable or improved accuracy, is important for computer architecture research and development. By enabling faster iterations in processor and application design and optimization, **SimPoint++** could greatly accelerate the pace of innovation in the field.

Chapter 5

METACAST: Generalizing HPC Application Runtime Prediction

5.1 Overview

Precise High Performance Computing (HPC) application performance evaluation and prediction helps identify performance bottlenecks within an application, enables customized configurations of hardware and software components co-designed to match the specific requirements of each application [31], and guides code design and optimization [52, 116]. Existing approaches such as simulation, analytical modeling, and learning-based modeling [98] can predict HPC performance at scale but often fall short in generalizability. This limitation arises because HPC application performance is almost always input dependent, requiring classic analysis or machine learning (ML) performance extrapolation approaches to characterize the input space, making them highly **application-specific** [23].

In this work, we build accurate and **application-generalizable** models for HPC performance prediction using *meta-learning*. Meta-learning [50], also known as "few-shot" learning, trains models to quickly assimilate new tasks using limited data. A

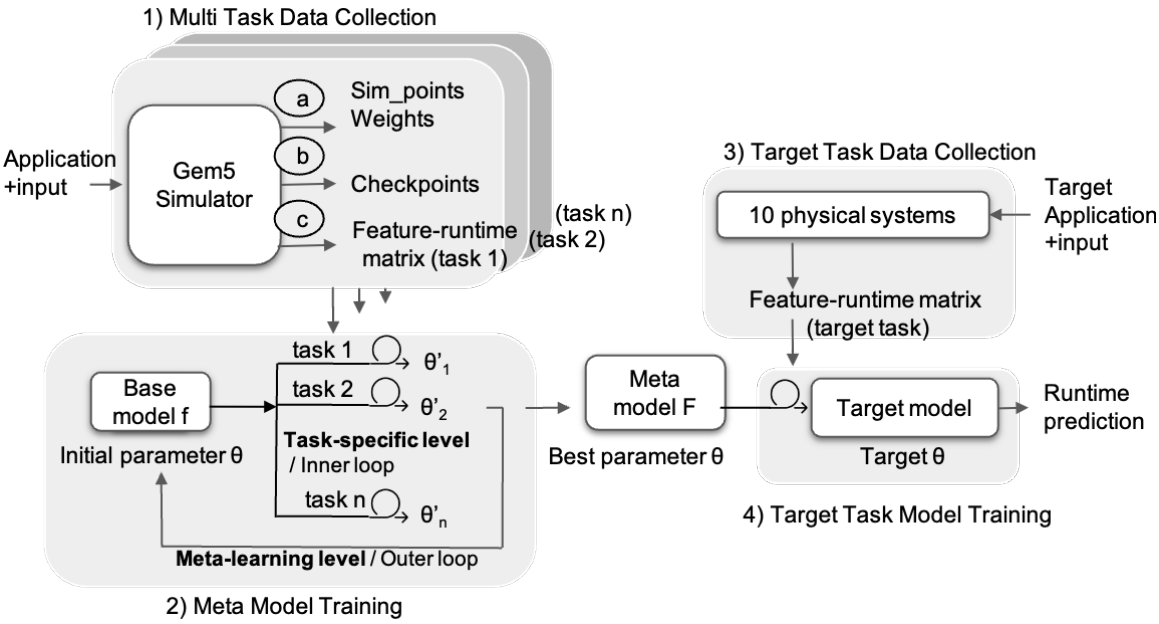


Figure 5.1: **MetaCast** Workflow

task here refers to a specific application runtime prediction problem. Meta-learning models circumvent the need to construct bespoke models for each application by leveraging a wealth of *related* tasks. This approach initializes the model to a highly informed state, expediting adaptation processes for new HPC applications—ideal for rapid deployment in dynamic environments.

To efficiently generate realistic multi-task datasets, we employ Gem5 [75], a sophisticated simulator that provides *cycle-level accuracy* and *comprehensive flexibility* for emulating application performance across diverse computer architectures and microarchitectures. Building on the foundational aspects of meta-learning and dataset collection via simulation, we develop **MetaCast**, a robust performance engine framework designed for precise runtime predictions for new applications on targeted architectures. As depicted in Fig.5.1, the workflow of **MetaCast** includes four steps: Multi-Task Data Collection, Meta-Model Training, Target Task Data Collection, and Target Task Model Training. By incorporating the first-order meta-learning algorithm[37], **MetaCast** constructs a general model that can be swiftly fine-tuned for novel applications. This significantly reduces model development time and catalyzes

the co-design process.

5.2 Methods

5.2.1 Multi Task Data Collection

While several performance results are available (*e.g.*, on the SPEC website [4]), the features and their diversity do not meet our requirements. Therefore, we have turned to simulation for data collection. Simulating a workload's full execution performance across various architectural configurations in Gem5 is a notably time-intensive task. In **MetaCast**, we reduce simulation time with the simulation acceleration tool SimPoint [44] described in the last chapter.

In our workflow, we leverage Gem5's diverse CPU models and memory access types to balance simulation accuracy and speed. Gem5 supports three primary memory access types: timing accesses (realistic timing and contention), atomic accesses (quicker approximations for cache warming), and functional accesses (instantaneous, primarily for debugging). These access types are utilized by various CPU models, including AtomicSimpleCPU, TimingSimpleCPU, and the more granular O3CPU, which simulates an out-of-order pipeline. Specifically, in sub-step a, we employ Gem5 with atomic memory accesses without caching to identify sim_points and weights for each task. We choose atomic accesses here because SimPoint analysis requires only basic block vectors, not detailed timing information, and atomic accesses provide sufficient accuracy while being significantly faster than timing accesses. In sub-step b, we generate checkpoints using TimingSimpleCPU rather than O3CPU because checkpoint creation primarily involves capturing architectural state, where the additional complexity of out-of-order execution simulation would increase computation time without improving checkpoint quality. We use atomic memory accesses with caching in this step to establish a warmup cache. Finally, in sub-step c, we employ

timing memory accesses to meticulously record the accurate performance (runtime) for each sim_point checkpoint across various system configurations, simulating cache accesses and memory system responses in detail. By accumulating the runtime of each sim_point checkpoint according to weights, we achieve the runtime for each system configuration.

5.2.2 Meta-Model Training

Meta-model training includes dual-level parameter updates in the base model and meta-learning model.

Base model

Meta-learning iteratively updates a base model across a range of tasks to form an optimized, generalized model, subsequently fine-tuning this model for specific target applications. For our meta-learning framework, the base model f could be any gradient-descent-based architecture, such as neural networks for regression, such as a Recurrent Neural Networks (RNN) (variants of which include Long Short-Term Memory (LSTM) [48] networks, which excel at learning long-term dependencies, and Gated Recurrent Units (GRUs) [25], which address the vanishing gradient problem), or a Convolutional Neural Networks (CNN). In developing **MetaCast**, we opt against more complex models and stick with a classical neural network with 2 hidden layers. In testing, we found that more complex models lowered accuracy, likely due to our limited feature size. For systems with more features, a more complex base model would be easy to substitute into **MetaCast**.

Meta-learning model

Consider a base model f designed for the runtime prediction of an HPC application. This model functions by mapping an input vector x, representing various system

Algorithm 1: meta-learning Training step

Data: $p(\mathcal{T})$:distribution over tasks, inner loop learning rate α, outer loop learning rate β, Loss function $\mathcal{L}$ is MSE
Result: meta-model initialization θ

Randomly initialize parameters θ
while *iteration not done* **do**
  Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
  for *all* $\mathcal{T}_i$ **do**
    Sample K training points $\mathcal{D} = \{x^{(j)}, y^{(j)}\}$ from $\mathcal{T}_i$
    Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ for this K examples
    Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
    Sample testing points $\mathcal{D}' = \{x^{(j)}, y^{(j)}\}$ from $\mathcal{T}_i$ for meta-update
  end
  $\beta' = \beta * (1 - iteration/iterations)$
  Update $\theta \leftarrow \theta - \beta' \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}\left(f_{\theta_i'}\right)$
end

configurations (*e.g.*, memory architecture or CPU microarchitecture), to an output y, which is the application's runtime on that system. We conceptualize the runtime prediction for a single application as one task, denoted as $\mathcal{T}_i$. Each task $\mathcal{T}_i$ encompasses multiple sample data points $x^{(j)}, y^{(j)}$ within it. These tasks are characterized by a certain degree of diversity. In **MetaCast**, we randomly select one task from a distribution of such tasks, represented as $\mathcal{T}_i \sim p(\mathcal{T})$. This approach allows us to train the model across a variety of scenarios, enhancing its adaptability and generalization capabilities for runtime prediction in diverse system architectures.

In Algorithm 1, we initiate the process by randomly initializing the meta-parameter θ. Subsequently, a batch of tasks is sampled from the task distribution $p(\mathcal{T})$. For each sampled task $\mathcal{T}_i$, the base model parameters θ are adjusted to θ_i' through an inner loop learning process. This adjustment is based on gradient updates applied to the training dataset $\mathcal{D}$ specific to each task. The updated parameters θ_i' are then evaluated against the test dataset $\mathcal{D}'$ for each task $\mathcal{T}_i$. The gradients computed during this evaluation, with respect to the original meta-parameter θ, are aggregated across

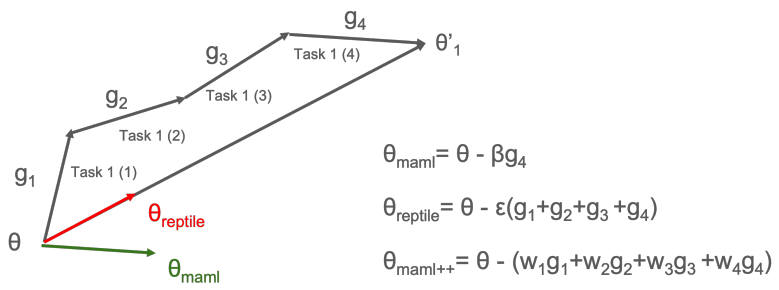


Figure 5.2: Meta-learning algorithms

all tasks. These accumulated gradients are then used to update θ in what we term the outer loop learning process. This iterative process is repeated multiple times until the model achieves the desired level of accuracy. Two critical hyperparameters in this process are the learning rates for the inner (α) and outer (β) loops. The careful tuning of these rates is essential for the effective training and convergence of the model.

We focus on the backpropagation of the meta-gradient through the gradient operator in various meta-learning algorithms. First-Order MAML (FOMAML) [37] simplifies calculations by omitting second derivatives, effectively utilizing only the final update from the inner loop. For instance, if the inner loop comprises four steps, the outer loop update in FOMAML is expressed as $\theta \leftarrow \theta - \beta g_4$, where g_4 is the gradient at the fourth step. Reptile [86] adopts a more gradual approach for meta-parameter updates: $\theta \leftarrow \theta - \varepsilon(\theta'_i - \theta)$. This translates to $\theta \leftarrow \theta - \varepsilon(g_1 + g_2 + g_3 + g_4)$, effectively incorporating the sum of gradients from all steps in the inner loop. MAML-SGD [72] introduces a matrix of learning rates, represented as α, instead of a single scalar value. Thus, the meta-parameters in MAML-SGD include both θ and α. However, the complexity of MAML-SGD leads to convergence challenges due to the increased number of meta-parameters. MAML++ [18] introduces a nuanced approach by applying annealed weighting to the inner loop steps. This method assigns progressively increasing weights to later steps: $\theta \leftarrow \theta - (w_1 g_1 + w_2 g_2 + w_3 g_3 + w_4 g_4)$, allowing for a balanced incorporation of gradients from all steps. Given these differences, which

are visually summarized in Fig. 5.2, we chose to implement MAML, Reptile, and MAML++ in our framework, excluding MAML-SGD due to its difficulty in achieving convergence. Our selection aims to maximize performance by leveraging the unique strengths of each algorithm.

While selecting the appropriate meta-learning algorithms, we considered the specific constraints of our HPC performance prediction task that limit the applicability of many advanced techniques. First, since our training dataset is derived from simulations, we needed to balance the number of features with the variety within each feature. Second, the time-intensive nature of data collection in HPC environments restricts the feasibility of methods that require extensive training data. Moreover, our primary goal was to effectively illustrate the potential of meta-learning for HPC performance prediction. Thus, we opted for classic meta-learning approaches that demonstrate this concept without the added complexity and overhead of more advanced methods.

5.2.3 Target Task Data Collection

Our objective is to accurately predict the performance of a new application on potential architectural configurations (the target task). In contrast to the broader training process, this phase requires only a minimal set of data samples. Since we need so few samples, we collect these samples from physical systems rather than simulations.

While physical system data collection offers advantages such as real-world performance characteristics and less data collection time, it faces limitations including restricted hardware availability, environmental variability, and measurement inconsistencies across different runs. Despite these challenges, we suggest including target task data from a wide variety of physical hardware to improve performance. When possible, we perform multiple runs per system configuration on dedicated systems to mitigate measurement variability and ensure reliable data collection.

5.2.4 Target Task Model Training

In **MetaCast**, fine-tuning for a target task commences with the pre-optimized meta-model parameters, significantly reducing the overall training duration. This process involves only a few update iterations using standard gradient descent with the target task data, which swiftly optimizes the model specifically for the target task. Once this stage is complete, the model is fully prepared to make accurate performance predictions across various potential architectural setups.

5.3 Experiment

We conduct our experimental evaluation on Cloudlab [32] with an x86 architecture node (c220g2) equipped with a Haswell processor and 4GB of memory. The software versions are Ubuntu 18.04, GCC/G++/FORTRAN 7.5.0, python 3.6.9, and gem5 22.1.0.0.

5.3.1 Simulation Platform

Initially, we use Gem5 with NonCachingSimpleCPU to acquire sim_points and weights for each task, setting the BBV chunk interval at 100 million instructions. Subsequently, we switch to AtomicSimpleCPU for checkpoint creation, followed by runtime collection using O3CPU with tailored parameters in Gem5.

We construct 140 x86-based systems, incorporating 18 representative features that significantly impact processor performance. These features fall into three main categories: memory hierarchy, instruction processing, and resource management.

The memory hierarchy parameters include three levels of cache (L1-L3) with varying sizes and associativity. For example, L1 cache sizes range from 32KB to 128KB, matching common configurations in modern processors. The instruction processing parameters, such as fetchWidth and decodeWidth, determine how many

instructions can be processed simultaneously in different pipeline stages. For instance, fetchWidth (4-12 instructions) represents the processor's ability to fetch multiple instructions per cycle, directly affecting instruction-level parallelism. Resource management parameters include numROBEntries (Reorder Buffer entries, ranging from 100 to 512), which determines how many instructions can be executed out of order, and LQ/SQEntries (Load/Store Queue entries), which control the number of outstanding memory operations. These parameters are important for modern out-of-order processors as they influence the processor's ability to decrease memory latency and exploit instruction-level parallelism.

We draw a combination of system configurations from Table 5.1 according to system setting rules and physical system settings [53]. For example, cache line size must be a power of 2 and a multiple of the fetch buffer to maintain cache coherency. While Gem5 offers a broader parameter space, we focus on representative features to validate **MetaCast**, as they capture the key architectural aspects that influence application performance.

5.3.2 Application Workload

We use the SPEC CPU 2017 benchmark suite [4] because its collection of real-world applications spans diverse domains such as compilers, chess engines, video compression, and weather forecasting. This suite provides a broad spectrum of workloads with dynamic instruction counts and large data footprints, ensuring our experiments are comprehensive. From the SPEC CPU 2017 suite, we select 25 benchmarks with their initial reference input, allocating 20 for training and 5 for testing **MetaCast**.

To mitigate overfitting, we augment our training dataset with synthetic tasks. In our original benchmarks, each sim_point is associated with a weight that represents its importance in the overall program behavior, and these weights are determined based on the frequency and similarity of program phases. To create synthetic tasks,

Table 5.1: System configuration in Gem5 simulator

	Parameter	Measure	Values	Explanation
1	l1i_size	kb	32,64,128	Size of Level 1 Instruction cache
2	l1d_size	kb	32,48,64,128	Size of Level 1 Data cache
3	l1i_assoc		2,4,8	Associativity of L1 Instruction cache
4	l1d_assoc		2,4,8,12	Associativity of L1 Data cache
5	l2_size	kb	256,512,1024,1280	Size of Level 2 cache
6	l2_assoc		4,8,10,16,20	Associativity of L2 cache
7	l3_size	mb	1,2,3,4,6,8,16,32	Size of Level 3 cache
8	l3_assoc		4,8,12,16	Associativity of L3 cache
9	cacheline_size	byte	32,64,128	Size of each cache line
10	mem-type		DDR3_2133_8x8, DDR4_2400_16x4, LPDDR2_S4_1066_1x32	Type of main memory
11	cpu_clock	Ghz	2,3,4	CPU clock frequency
12	fetchWidth		4,8,10,12	Number of instructions fetched per cycle
13	fetchBufferSize		16,32,64	Size of buffer holding fetched instructions
14	fetchQueueSize		12,16,24,32	Size of queue for fetched instructions
15	decodeWidth		4,6,8,12	Number of instructions decoded per cycle
16	LQEntries		16,32,64,128,192	Number of Load Queue entries
17	SQEntries		16,32,36,64,128	Number of Store Queue entries
18	numROBEntries		100,168,256,352,512	Number of Reorder Buffer entries

we manipulate these SimPoint weights. While maintaining the original sim_points, we randomly reassign weights to these points, creating synthetic programs. This process alters the relative importance of different program phases without changing the underlying code segments. For each original training task, we generate 20 additional synthetic tasks using this weight reassignment method. These synthetic tasks differ from real-world tasks in that they are artificial combinations of existing benchmark characteristics.

The synthetic tasks are designed to introduce variability into the training data, potentially helping the model generalize better. However, our experimental findings indicate that these additional synthetic tasks do not enhance accuracy. We attribute this lack of improvement to the inherently constrained size and diversity of the original task set. The synthetic tasks, while introducing some variation, may not significantly expand the range of behaviors and patterns that the model needs to learn. Given these results, it may be worth considering excluding synthetic tasks in future experiments or exploring alternative methods of data augmentation. Alternatively, expanding the range of real-world benchmarks or using different input sets for existing benchmarks might prove more effective in improving model accuracy.

Our experimental dataset encompasses a total of 362 tasks, with each task containing 140 distinct samples. For the training phase, we dedicate 352 tasks, reserving 5 tasks exclusively for validation, which aids in the hyperparameter tuning process. The remaining 5 tasks are allocated for the testing phase to evaluate the model's performance. Notably, we ensure the integrity of the test tasks by refraining from creating synthetic tasks from them.

5.3.3 Model Training

We construct the model using PyTorch [90] and explore three meta-learning algorithms: MAML [37], Reptile [86], and MAML++ [18], to identify the most effective parameter

Table 5.2: Hyperparameters in Meta-model

Feature	Range	Best Value
n_shot	5–20	9
tasks	10–30	18
outer_step_size	0.001– 0.2	0.001
inner_step_size	0.001–0.03	0.00926
inner_grad_steps	1–8	6
eval_grad_steps	5–20	18
first_level	32–64	40
second_level	10–32	31

updating strategy for our specific problem domain. For the base model, we use three architectures: a neural network for regression, an RNN (LSTM and GRU), and a CNN. The neural network employs a two-layer structure with tanh activations and is optimized using stochastic gradient descent (SGD). For the RNNs and CNN, after data preprocessing to fit the required input format, we set a sequence length of 3 and an input dimension of 6, utilizing ReLU activations and the SGD optimizer for training.

5.3.4 Hyperparameter Optimization

While meta-learning effectively initializes model parameters, optimizing certain hyperparameters remains crucial. Improper selection of mini-batches, for instance, can drastically affect model performance. To address this, we employ Bayesian Optimization (BO), which outperforms traditional grid or random search methods by leveraging Gaussian Processes (GPs) for efficient hyperparameter selection. This process starts with constructing a surrogate model from existing data, which is then used to predict outcomes at new data points and iteratively refined.

For our experiments, we predefined a range for each hyperparameter and used the AX [1] platform for the Bayesian Optimization process. This approach systematically determines the most effective settings, particularly for inner and outer loop learning

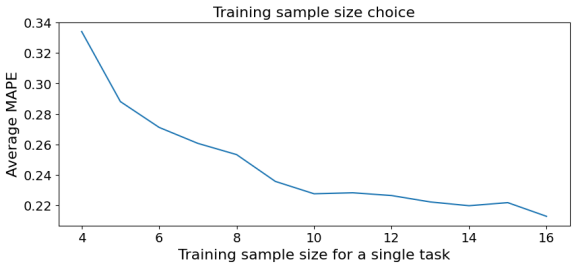


Figure 5.3: More training samples lead to improved prediction.

rates and step sizes. **MetaCast** has the following hyperparameters (Table 5.2): N_short is the number of examples per task that the model is provided to learn from when training the meta-model. The number of tasks included in one batch of meta-learning. Each task in the batch can be used to update the model during training. outer_step_size and inner_step_size are the learning rates. The outer loop updates the meta-parameters, while the inner loop adapts to specific tasks. inner_grad_steps and eval_grad_steps are the numbers of gradient update steps in the inner loop (for task adaptation) and during evaluation, respectively. first_level and second_level are the number of neurons in the first and second hidden layers, respectively.

We find that learning rates are the most influential hyperparameter for **MetaCast** performance.

5.4 Results

5.4.1 Meta-model Accuracy for Benchmarks

To evaluate **MetaCast**, we test the efficacy of the trained models. We adjust the training sample size for each new task to balance accuracy and application execution complexity. While a very small sample size (*e.g.*, 3 or 5) does not yield satisfactory accuracy, a slightly larger number (*e.g.*, 15) only improves accuracy by 1%. Consequently, as shown in Fig. 5.3, we settle on 10 as the optimal size, which also aligns with the elbow point identification method.

We randomly select 10 samples for each new task, train a new model starting from the pre-trained meta-model, and then evaluate it across 30 architecture configurations, calculating the average mean absolute percentage error (MAPE). To ensure robustness against variability in training task samples, we conduct 20 repetitions of training-testing splits and meta-model training. Each test task is then assessed 10 times. Thus, we have built 20 meta-models, and we test on the remaining, unused, SPEC applications. Among the three tested meta-learning algorithms, MAML with outer learning rate annealing shows a marginally superior performance. Using neural network regression as the base model, we achieve an average MAPE of 18% with a standard deviation of 0.013. In contrast, base models using RNNs, their variants, and CNNs did not yield improvements in accuracy.

The standard deviation of 0.013 in our MAPE results indicates consistency in performance across various types of applications and workloads. In practical terms, users can expect relatively uniform performance from **MetaCast**. The small standard deviation also implies that there are no significant outliers where the model's performance drops substantially, further reinforcing the robustness of our approach.

Additionally, we compare the model's performance with the previous SPEC CPU 2006 suite [111]. The results are analogous; we achieve an average MAPE of 19.8% with a standard deviation of 0.019. This slightly higher standard deviation for the older suite suggests a bit more variability in performance, but still indicates consistent accuracy across different benchmark versions. These results confirm **MetaCast**'s effectiveness and stability across suite versions.

Benchmark: The performance of **MetaCast** on the SPEC CPU2017 benchmarks is illustrated in Fig. 5.4. We observed a consistent average MAPE of approximately 18% across most benchmarks, indicating stable performance across different test cases. However, we identified outliers in each benchmark with higher MAPE values. This discrepancy is attributed to the limited variability in the 10 training samples, which

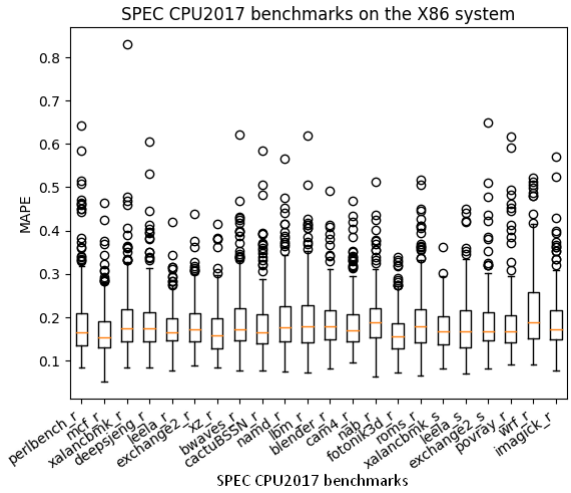


Figure 5.4: SPEC CPU2017 benchmarks on x86. The average MAPE is 18%.

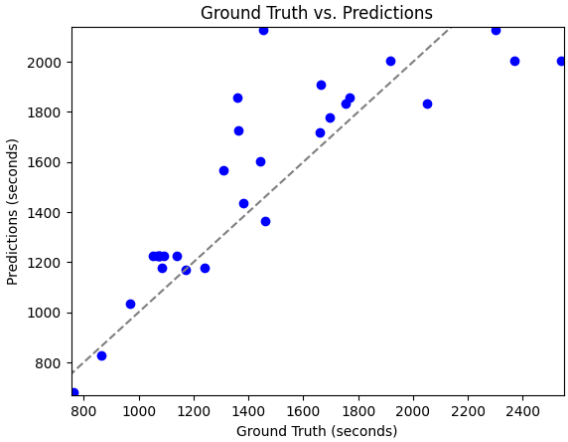


Figure 5.5: **MetaCast** predictions for 30 tests on the perlbench_r benchmark with just 10 training samples. Points closer to the dashed line are more accurate.

proves insufficient for the target model to fully capture the benchmark's characteristics, complicating predictions for new architectures. These findings underscore the importance of collecting diverse data samples for the target application to ensure the target model's effectiveness.

Single Application: Fig. 5.5 shows the results for a specific application benchmark perlbench_r in SPEC CPU 2017. Points near the dashed line show a strong correlation between the model's prediction runtimes and their actual results, demonstrating the model's proficiency in learning and adapting to system differences. However, some

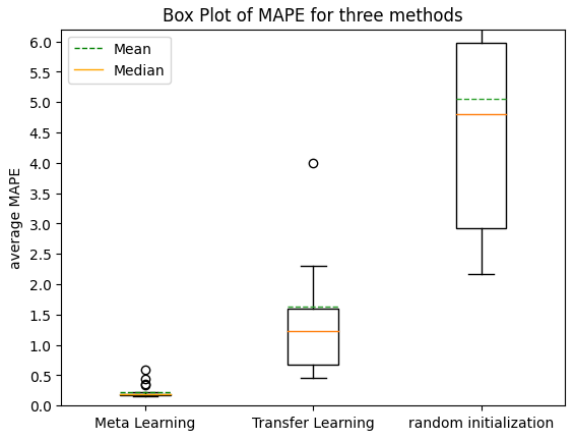


Figure 5.6: **MetaCast** achieves superior results compared to transfer learning (average MAPE=160%) and random initialization (average MAPE=510%).

points show larger discrepancies , indicating lower prediction accuracy. This issue likely arises when the distribution of test samples significantly diverges from the training samples, reinforcing the need for training samples from a diverse range of systems to enhance model performance for new tasks. We observed that some outliers perform better as training sample sizes increase, but a few remain unpredictable. This indicates that the prediction problem is challenging given the features we have.

Transfer Learning: We benchmark **MetaCast** against transfer learning and randomly initialized models as baselines. The transfer learning model uses all data from different tasks to build a single pre-trained model. Our results (Fig. 5.6) reveal that the meta-model surpasses these baselines in accuracy, with a notably smaller deviation in its predictions. Although transfer learning demonstrates an advantage over random initialization, it still has a relatively big MAPE compared to meta-learning because of the wide distribution of runtimes in the training tasks, which undermines the premise of task similarity essential for effective transfer learning. **MetaCast** achieves comparable or superior results compared to other approaches utilizing transfer learning. For instance, Mariani*et al.* [79] reported a 30% prediction error for their profile-prediction model, Sun*et al.* [112] observed an average prediction error of 20%, and Mankodi*et al.* [78] reported an average error ranging from 10% to 25%.

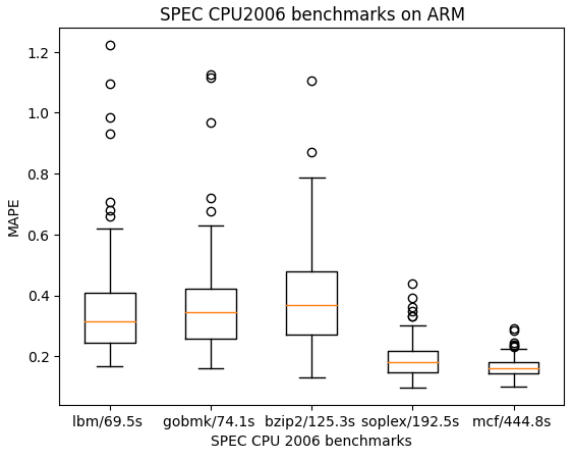


Figure 5.7: SPEC CPU2006 benchmarks on ARM. Runtime is denoted after each benchmark with a slash. Note that longer benchmarks show lower MAPE.

5.4.2 Cross-Architecture Generalizability

HPC community increasingly favors power-efficient systems that offer reduced power consumption, occupy less space, and generate less heat. ARM architectures, distinct from x86 with their Reduced Instruction Set Computer (RISC) architecture, are increasingly prevalent in the server market, noted for their superior performance-per-watt in enterprise environments. Testing on ARM systems helps evaluate **MetaCast**'s adaptability.

We do not utilize the Instruction Set Architecture (ISA) type (x86 or ARM) as a feature in our model. Due to architectural differences, Gem5 checkpoints created on an x86 system cannot be used for ARM system simulations, and vice versa. Thus, we need separate data collection processes for each system. Our meta-model is initially trained using data from x86-based SPEC CPU2017 benchmarks, which would traditionally limit the model's applicability to x86 systems only. The key benefit of **MetaCast** is the ability to use this x86-trained meta-model as a starting point for predictions on ARM systems.

To demonstrate this, we extended the **MetaCast** meta-model—originally trained on x86-based SPEC CPU2017 benchmarks—to ARM systems with the SPEC CPU2006

Table 5.3: Evaluation applicaitons

Application	Description	Language
miniQMC	real space quantum MC	C++
PICSARlite	testing Particle-In-Cell kernels	Fortran
Quicksilver	dynamic MC particle transport	C++
Goulash	test compiler-linker interoperability	C++
snap	discrete particle transport	Fortran/C++
PENNANT	mesh data structures	C++
lammps	molecular dynamics simulation	C++
vite	Louvain undirected graph clustering	C++
minivite	Single-phase Louvain	C++
sw4	3-D seismic modeling	C++
sw4lite	barebone version of SW4	C++

benchmarks, without altering the meta-model's structure or re-training (Fig. 5.7). By fine-tuning the target model with a small amount of ARM system data, we show that **MetaCast** can accurately predict performance on previously unseen ARM configurations.

Remarkably, longer runtime benchmarks (*e.g.*, mcf, soplex) from SPEC CPU2006 on ARM (m400 on Cloudlab) matched the performance (MAPE $< 20\%$) seen in the x86 experiments. This consistency highlights the untouched meta-model's capacity to serve as the foundation for new models to accurately predict performance across different architectures and benchmark suites. Conversely, higher MAPE values were observed in benchmarks with inherently shorter runtimes (*e.g.*, lbm, gobmk, bzip2), which is an anticipated outcome given the complexity and longer runtimes of the SPEC CPU2017 benchmarks used for initial meta-model training.

This successful application of a single, unchanged meta-model across architectures and different sets of applications, achieving commendable MAPE scores, underscores the flexibility and utility of **MetaCast** to create powerful future co-designed architectures.

5.4.3 Meta-model Accuracy for Real Applications

To assess **MetaCast**'s real world performance, we conduct tests on various HPC applications from the ECP [35] project (Table 5.3), running them on 13 distinct physical nodes with varying architectures in Cloudlab (c220g2, c220g5, c8220, r320, m510, xl170, c6220, rs630, rs620, d170, c6525-25g, r650, c6420). See more information on applications in Appendix A.1 and nodes in Appendix A.2 These nodes span a range of Intel microarchitectures, including Nehalem, Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake, and Ice Lake running on Ubuntu 22, and compiled with GCC/G++/FORTRAN 11.4.0. These applications are executed in serial mode instead of parallel mode, since the limitations of Simpoint when simulating with Gem5 restrict **MetaCast** to serial application training.

To minimize variability, we average runtimes from five executions of each application per system. For each application, we use 10 samples for training and 3 for testing. The data are randomly divided, and we train the target model for each application 100 times to ensure robust evaluation. The prediction accuracy of the meta-learning model is detailed in Fig. 5.8.

Our analysis of higher MAPE errors in the results reveals a trend: tasks with shorter runtimes (*e.g.*, minivite, vite) tend to have higher MAPEs, while those with longer runtimes (*e.g.*, quicksilver, sw4lite, lammps) typically show lower MAPEs, similar to the SPEC CPU 2017 benchmarks. This pattern reflects the training data characteristics of the SPEC CPU benchmarks, which predominantly feature longer runtimes, suggesting **MetaCast**'s proficiency in predicting applications with extended durations. For more accurate predictions of applications with shorter runtimes, incorporating tasks with similar characteristics into the training dataset is crucial. An outlier, picsarlite, exhibits high MAPE and significant standard deviation. The high variability in picsarlite's prediction accuracy can be attributed to several factors. Firstly, picsarlite demonstrates considerable runtime variation even within the same

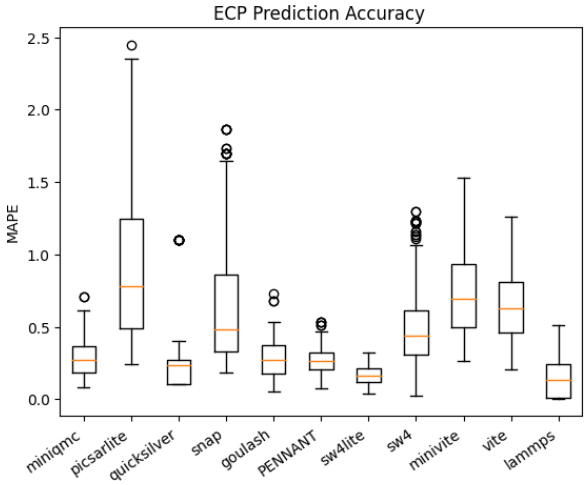


Figure 5.8: Real HPC applications from ECP.

system. This inherent variability in the application's behavior leads to inconsistent training data, making it challenging for the model to capture a stable performance pattern. Secondly, picsarlite may have unique computational patterns or memory access behaviors that differ significantly from the SPEC CPU benchmarks used in training the meta-model. This mismatch could contribute to the prediction difficulties.

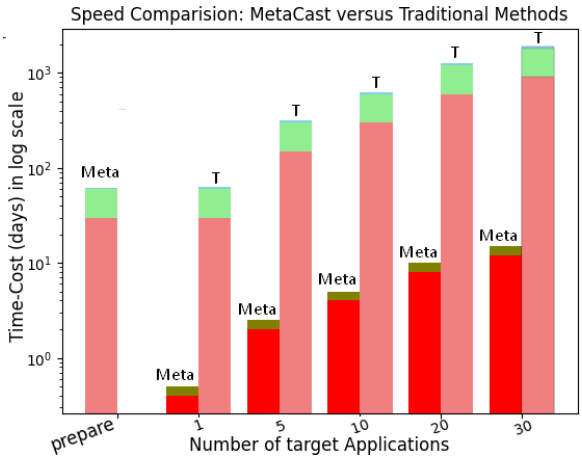
To mitigate such issues in future work, for applications showing high variability like picsarlite, increasing the number of repeated runs on each physical system could help in capturing a more representative performance profile. Additionally, implementing more rigorous system isolation techniques during execution could minimize interference from background processes or system events, potentially reducing run-to-run variability.

Our suite includes pairs of parent/proxy applications (*e.g.*, sw4/sw4lite and vite/minivite). Proxy applications are simplified models of larger parent applications, representing similar characteristics. Despite handling the same input problem, the runtime differences between these pairs result in disparate prediction accuracies. This observation highlights the need for careful consideration when using proxy applications for performance prediction in HPC environments.

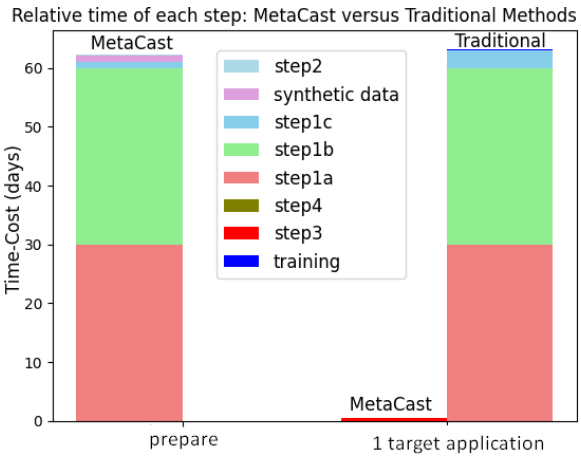
5.4.4 Time Efficiency in MetaCast versus Traditional Methods

The optimal training dataset size can vary, depending on the problem complexity, to ensure model generalizability and reduce overfitting. Previous studies required 200 samples for models with 12 features [67], and about 300 samples for 15-feature models [68]. Based on these studies, we set 500 samples as adequate for our 18-feature model. Using Figure 5.1 as a reference, for the SPEC CPU 2017 benchmarks, the average simulation time per application in steps $1a$ and $1b$ is about a month each. Step $1c$, involving checkpoint reloads for one architecture configuration, takes roughly 30 minutes per sample. While simulations are serial, checkpoint reloads can be parallelized, making step $1c$ about a day for 140 data points or three days for 500 data points. Synthetic task generation, calculating runtime for varied system configurations, averages an hour per application, totaling around a day for all 25 benchmarks.

Hyperparameter training for the meta-model in step 2 takes about 4 hours, with formal training requiring approximately 0.8 hours. In contrast, traditional methods for new applications require complete data collection and training. Unlike parallel data collection for meta-model training, new tasks necessitate sequential data gathering on real systems, averaging 1 hour per system, resulting in 10 hours for 10 samples in step 3. Fine-tuning the target model in step 4 takes about 0.1 hours, compared to 0.5 hours for traditional training on 500 data points. **MetaCast** approach significantly reduces both data collection and model fine-tuning times. As shown in Figure 5.9a, **MetaCast** becomes more cost-effective with an increasing number of applications. Figure 5.9b shows a detailed relative time of steps. The preparation stage for **MetaCast** takes 62.2 days, while traditional methods start at zero cost. However, for each new application, **MetaCast** requires only 0.5 days, compared to 63.5 days for traditional methods, achieving a 127× increase in speed.



(a) **MetaCast** is consistently more efficient than the traditional application-specific method (T). * y-axis is on a logarithmic scale.



(b) **MetaCast** takes extra time to prepare but becomes more efficient than the traditional application-specific method for every new target application. The relative time of each step is shown in different colors.

Figure 5.9: Comparison of efficiency and time distribution between **MetaCast** and traditional methods.

5.5 Discussion

5.5.1 Accuracy Considerations

SimPoint Representation: Sim_points cluster code segments using k-means, but in some applications, the Basic Block Vectors (BBVs) may not be Gaussian, potentially reducing the representativeness of the sim_points. This limitation also affects the effectiveness of our data augmentation method, which relies on weight reassignment to generate synthetic applications. If the sim_points are not sufficiently representative, the data augmentation may not perform as expected.

Gem5 Simulation Incompleteness: We observe segmentation faults in Gem5 when reloading checkpoints for certain applications, resulting in checkpoint building failures. This issue affected approximately 15% of our simulation attempts across all benchmarks. While uniform failures across all configurations for a single application have limited impact, partial failures introduce noise, as Gem5 fails to reload the same checkpoints for different configurations, effectively altering the application under different settings.

Within-Task Similarity and Diversity: The range of system configurations is somewhat limited, which might affect task diversity and similarity. When randomly selecting training samples for each task, if the training samples are too similar to each other and significantly different from the testing set, the predictions may become outliers.

Task Selection: Our tasks include original benchmarks and synthetic ones derived from them. While the synthetic tasks do not notably improve accuracy, incorporating more original applications might enhance the model's performance and generalizability.

5.5.2 Future Directions

For future work, we may also explore advanced simulation methods to enhance dataset accuracy or whether the initial conditions provided a sufficiently adaptable meta-model for a broad spectrum of new tasks.

In this work, we focus on serial applications. We are currently expanding this research to include multi-threaded applications, which leverage the parallelism of multi-core systems. Analyzing multi-threaded applications is inherently challenging due to issues like thread idleness, interference, and unbalanced workloads [16]. The concept of using loop iterations as slices for single-threaded programs was introduced in [66]. Our data collection process works as follows: LoopPoint [104] uses loop entries as slice boundaries, allowing us to specify simulation regions using a (PC, count) pair to mark the starting and ending loop entries for each region. We use Simpoint to select the regions of interest (ROI), build region pinballs, transfer them to Elfies [93] files, and replay them with the Gem5 simulator. Finally, we apply different configurations to obtain the runtimes.

5.5.3 Conclusion

Runtime prediction is pivotal for optimizing hardware/software co-design, resource allocation, and evaluating hardware modifications. Traditional approaches, typically application and architecture-specific, are limited in scope and require extensive training data. Addressing these challenges, **MetaCast** innovatively combines meta-learning with architecture simulation to predict runtime for a wide range of applications and systems. **MetaCast** facilitates quick, preliminary insights into application performance across various architectures without necessitating an in-depth analysis, streamlining the performance analysis process.

MetaCast not only covers a broad spectrum of applications and diverse architectural platforms but also significantly reduces the need for extensive training samples.

With **MetaCast**, we achieve an average Mean Absolute Percentage Error (MAPE) of 18% on the SPEC CPU 2017 benchmarks and 25% on real applications with as few as **ten** training samples per task. This efficiency translates into a notable 127× speedup in training time for additional tasks/applications compared to traditional ML methods. Remarkably, longer runtime benchmarks from SPEC CPU2006 on ARM matched the performance seen in the x86 experiments, emphasizing the adaptability and robustness of **MetaCast**, paving the way for **MetaCast** to improve the co-design of future HPC systems.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This dissertation presents an integrated approach to accelerating HPC application design in heterogeneous environments. Our work introduces innovative tools that collectively address critical challenges in proxy analysis, simulation, and ML-based performance prediction. The synergy between these tools represents a significant advancement in optimizing HPC application design, offering a unified solution that streamlines the entire process from initial concept to optimized implementation.

Our contributions form a cohesive and novel strategy for HPC application design optimization: **Calder** establishes a robust foundation for accurate workload representation through quantitative proxy fidelity assessment. **SimPoint++** builds upon this by enabling efficient simulation and analysis of these representations. **MetaCast** leverages the outputs from both **Calder** and **SimPoint++** to deliver generalized performance predictions across diverse architectures. This integrated approach significantly reduces the time and computational resources required for HPC application design, enabling more rapid iterations and informed decision-making throughout the development process.

We first introduce **Calder**, a novel framework for quantifying the fidelity of proxy applications. **Calder** employs a systematic and quantitative approach to assess proxy fidelity, which is crucial for accurately representing complex HPC workloads and providing reliable performance estimations. By identifying the most representative features, **Calder** significantly reduces the data collection overhead by up to 95%, while maintaining high fidelity in workload representation. Moreover, it highlights features exhibiting dissimilarity between proxy and parent pairs, informing future proxy co-design efforts and enabling more targeted optimizations.

Second, we build **SimPoint++**, an advanced sampled HPC application simulation tool. By incorporating advanced dimension reduction techniques and improved clustering algorithms, **SimPoint++** identifies the most representative simulation points, reducing simulation time by up to 80% while maintaining high accuracy. This enables HPC system designers to efficiently explore design spaces in heterogeneous environments, rapidly evaluate design alternatives, and optimize computer architecture simulation and analysis, significantly accelerating the prototyping process.

Third, we introduce **MetaCast**, a framework that combines proxy application simulations with meta-learning technology. **MetaCast** provides quick and accurate runtime predictions for new applications on targeted architectures, offering a generalized performance modeling method. By leveraging meta-learning, MetaCast adapts to diverse workloads and architectures more effectively than traditional methods, speeding up model retraining time up to 127× and making it particularly suitable for dynamic HPC environments.

This integrated approach represents a significant advancement over existing isolated solutions for HPC design. It addresses key challenges in HPC application design and optimization, enabling rapid prototyping, testing, and refinement of HPC applications. By significantly reducing the time from initial design to optimized implementation, our approach allows developers to make quick, informed decisions about application

design and hardware selection without the need for extensive, time-consuming full-scale simulations or physical testing on every potential hardware configuration. This comprehensive solution accelerates the entire HPC application design cycle, potentially leading to faster innovations and more efficient utilization of HPC resources.

6.2 Future work

This dissertation can be extended from the following aspects.

1. **Incorporation of Time-Series Analysis in Calder:** To address the current limitation of representing only static behaviors, future work could integrate time-series analysis into **Calder**. For example, applying techniques like Dynamic Time Warping (DTW) could reveal temporal patterns such as intermittent resource bottlenecks in complex HPC applications, which static analysis might miss. This enhancement would provide a more comprehensive representation of application behavior over time.

2. **Development of a Combined Clustering Method for SimPoint++:** While our research has demonstrated the efficacy of updated K-means, the optimal clustering method may vary depending on the specific application. Future work could focus on developing a hybrid or ensemble clustering approach to balance computational efficiency with accuracy in representing diverse HPC workloads. This combined method could adapt more flexibly to varying data distributions, potentially leading to more robust and generalizable results across diverse applications.

3. **Expansion of Multi-thread MetaCast:** To overcome the challenges of data collection time and consistency in multi-threaded environments, future work should leverage SimPoint++ to accelerate multi-thread task data collection. This

direction is particularly crucial given the increasing importance of parallelization in HPC. Research could focus on addressing scalability issues and evaluating the impact of different threading models on simulation accuracy and efficiency.

4. **Expansion to Heterogeneous Computing Environments:** The works in this dissertation focus primarily on CPU architectures and micro-architectures. As computing landscapes become increasingly diverse, extending our methods to GPUs and specialized accelerators is a valuable direction for future research. This would involve adapting our techniques to the unique characteristics of these architectures, such as different memory hierarchies and parallelism models. Such expansion would enhance the applicability of our approach in modern, heterogeneous HPC environments.

Appendix A

Appendix

A.1 Real application information

Below is the list of real applications used in this work, with the required information to build:

1. Application: miniQMC
   (a) Repo: https://github.com/QMCPACK/miniqmc.git
   (b) Input: -r 0.99 -g '2 2 2'
2. Application: PICSARlite
   (a) Repo: https://bitbucket.org/berkeleylab/picsar.git
   (b) Input: example/input_file.pxr # line 33, change t_max = 100
3. Application: Quicksilver
   (a) Repo: https://github.com/LLNL/Quicksilver.git
   (b) Input: –nSteps 500
4. Application: Goulash

   (a) Repo: https://github.com/LLNL/goulash.git

   (b) Input: make check

5. Application: snap

   (a) Repo: https://github.com/lanl/SNAP.git

   (b) Input: Sample Input in official Repo with nsteps=100

6. Application: PENNANT

   (a) Repo: https://github.com/lanl/PENNANT

   (b) Input: ../test/nohpoly/nohpoly.pnt

7. Application: Lammps

   (a) Repo: https://download.lammps.org/tars/lammps-29Oct2020.tar.gz

   (b) Compilation flags: CCFLAGS = -pg LINKFLAGS = -pg

   (c) Input: in.snap.Ta.mod.single_attaway(The file located in the repo)

8. Application: vite

   (a) Repo: https://github.com/Exa-Graph/vite

   (b) Compilation flags: -pg

   (c) Input: happy.bin (generated with this command: mpiexec -n 8 /vite/bin/graphClustering -n 1024000 -s happy.bin)

9. Application: minivite

   (a) Repo: https://github.com/Exa-Graph/miniVite

   (b) Compilation flags: -fopenmp -pg

   (c) Input: happy.bin

10. Application: sw4

    (a) Repo: https://github.com/geodynamics/sw4

    (b) Compilation flags: -pg

    (c) Input: gaussianHill.in(The file located in the repo)

11. Application: sw4lite

    (a) Repo: https://github.com/geodynamics/sw4lite

    (b) Compilation flags:OPT = -pg

    (c) Input: gaussianHill.in

A.2 Real system configuration per node

Table A.1: Real system configuration per node

node	l1i _size	l1d _size	l1i _assoc	l1d _assoc	l2 _size	l2 _assoc	l3 _size	l3 _assoc	cache line _size	mem -type	cpu _clock	fetch Width	fetch Buffer Size	fetch Queue Size	decode Width	LQ Entries
c220g2	32	32	8	8	250	8	50	16	64	DDR4 2114	2.6	16	32	56	4	64
c220g5	32	32	8	8	1000	4	27.5	16	64	DDR4 2666	2.2	6	16	32	6	72
c8220	32	32	8	8	250	8	50	16	64	DDR4 1600	2.2	16	32	32	4	48
r320	32	32	16	8	250	16	20	8	64	RDIMM	2.1	4	16	28	4	64
m150	32	32	8	8	250	16	12	16	64	DDR4 2133	2	16	64	64	4	64
x170	32	32	8	8	250	8	25	16	64	DDR4 2400	2.4	16	64	64	4	64
c6220	32	32	8	8	250	8	40	16	64	DDR3	2.6	16	32	32	4	48
rs630	32	32	8	8	250	8	50	16	64	DDR4	2.6	16	32	56	4	64
rs620	32	32	16	8	250	16	50	8	64	DDR3	2.2	4	16	28	4	64
d710	32	32	4	8	500	8	8	16	64	DDR3	2.4	4	16	28	4	64
c6525-25g	32	32	8	8	500	8	128	16	64	DDR4	3	16	32	56	4	64
r650	48	32	8	12	125	8	108	16	64	DDR4 3200	2.4	6	16	70	6	128
c6420	32	32	8	8	1000	4	44	16	64	DDR4 2666	2.6	6	16	32	6	72

Bibliography

[1] AX, Hyperparameter Optimization using Bayesian. https://ax.dev/docs/why-ax.html.

[2] TOP500 homepage. https://www.top500.org/lists/top500/2024/06/.

[3] Co-Design: Deploying Leading-Edge Computing Capabilities. https://computing.llnl.gov/collaborations/co-design.

[4] Standard Performance Evaluation Corporation (SPEC) website. https://www.spec.org/cpu2017/.

[5] Hpc challenge benchmark. https://icl.utk.edu/hpcc/, 2012.

[6] Aries hardware counters. https://docplayer.net/55709631-Aries-hardware-counters.html, 2015.

[7] The pennant mini-app. https://github.com/lanl/PENNANT, 2016.

[8] Nekbone. https://github.com/Nek5000/Nekbone, 2017.

[9] Hpcg benchmark. https://www.hpcg-benchmark.org/, 2020.

[10] Snap: Sn (discrete ordinates) application proxy. https://github.com/lanl/SNAP, 2021.

[11] Picsar: Particle-in-cell scalable application resource. https://picsar.net/code/, 2022.

[12] O. Aaziz, J. Cook, J. Cook, T. Juedeman, D. Richards, and C. Vaughan. A methodology for characterizing the correspondence between real and proxy applications. In *CLUSTER*, pages 190–200, 2018.

[13] Omar Aaziz, Courtenay Vaughan, Jonathan Cook, Jeanine Cook, Jeffery Kuehn, and David Richards. Fine-grained analysis of communication similarity between real and proxy applications. In *PMBS*, pages 93–102. IEEE, 2019.

[14] Anthony Agelastos et al. The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *SC'14*, pages 154–165. IEEE, 2014.

[15] Ayaz Akram and Lina Sawalha. A survey of computer architecture simulation techniques and tools. *Ieee Access*, 7:78120–78145, 2019.

[16] Alaa R Alameldeen and David A Wood. Variability in architectural simulations of multi-threaded workloads. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 7–18. IEEE, 2003.

[17] A. S. Almgren et al. CASTRO: A New Compressible Astrophysical Solver. I. Hydrodynamics and Self-gravity. *Astrophysical Journal*, 715:1221–1238, June 2010. doi: 10.1088/0004-637X/715/2/1221.

[18] Antreas Antoniou, Harrison Edwards, and Amos Storkey. How to train your maml. *arXiv preprint arXiv:1810.09502*, 2018.

[19] 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 *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 724–737, 2021.

[20] R.F. Barrett et al. Assessing the Role of Mini-Applications in Predicting Key Performance Characteristics of Scientific and Engineering Applications. *Journal of Parallel and Distributed Computing*, 75(Supplement C):107 – 122, 2015. ISSN 0743-7315.

[21] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[22] Trevor E Carlson, Wim Heirman, Kenzo Van Craeynest, and Lieven Eeckhout. Barrierpoint: Sampled simulation of multi-threaded applications. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12. IEEE, 2014.

[23] Mehmet Cengiz, Matthew Forshaw, Amir Atapour-Abarghouei, and Andrew Stephen McGough. Predicting the performance of a computing system with deep networks. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*, pages 91–98, 2023.

[24] Xiaomeng Chen, Hui Zhang, Hanli Bai, Chunming Yang, Xujian Zhao, and Bo Li. Runtime prediction of high-performance computing jobs based on ensemble learning. In *Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications*, pages 56–62, 2020.

[25] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[26] Madalina Ciortan. Spectral graph clustering and optimal num-

ber of clusters estimation. https://towardsdatascience.com/spectral-graph-clustering-and-optimal-number-of-clusters-estimation-32704189a Jan 2019.

[27] Jeanine Cook, Hal Finkel, Christoph Junghams, Peter McCorquodale, Robert Pavel, and David F. Richards. Proxy app prospectus for ecp application development projects. doi: 10.2172/1477829. URL https://www.osti.gov/biblio/1477829.

[28] Sanjoy Dasgupta. Experiments with random projection. *arXiv preprint arXiv:1301.3849*, 2013.

[29] Sander De Pestel, Sam Van den Steen, Shoaib Akram, and Lieven Eeckhout. Rppm: Rapid performance prediction of multithreaded workloads on multicore processors. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 257–267. IEEE, 2019.

[30] V. Dobrev, Tz. Kolev, and R. Rieben. High-order curvilinear finite element methods for lagrangian hydrodynamics. *SIAM Journal on Scientific Computing*, 34:B606–B641, 2012.

[31] Sudip S Dosanjh, Richard F Barrett, DW Doerfler, Simon D Hammond, Karl S Hemmert, Michael A Heroux, Paul T Lin, Kevin T Pedretti, Arun F Rodrigues, TG Trucano, et al. Exascale design space exploration and co-design. *Future Generation Computer Systems*, 30:46–58, 2014.

[32] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of cloudlab. In *USENIX Annual Technical Conference*, pages 1–14, 2019.

[33] ECP. Exascale Proxy Application Suite. `https://proxyapps.exascaleproject.org`, 2020.

[34] Dominik Maria Endres and Johannes E Schindelin. A new metric for probability distributions. *IEEE Transactions on Information theory*, 49(7):1858–1860, 2003.

[35] Exascale Computing Project. ECP Proxy Applications. `https://proxyapps.exascaleproject.org/app/`, 2023. Accessed: 2023-10-01.

[36] 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.

[37] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR, 2017.

[38] Steven Flolid, Emily Shriver, Zachary Susskind, Benjamin Thorell, and Lizy K John. Simtrace: Capturing over time program phase behavior. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 226–228. IEEE, 2020.

[39] Rahulkumar Gayatri, Stan Moore, Evan Weinberg, Nicholas Lubbers, Sarah Anderson, Jack Deslippe, Danny Perez, and Aidan P Thompson. Rapid exploration of optimization strategies on advanced architectures using testsnap and lammps. *arXiv preprint arXiv:2011.12875*, 2020.

[40] S. Ghosh et al. MiniVite: A Graph Analytics Benchmarking Tool for Massively Parallel Systems. In *IEEE/ACM Perf. Modeling, Benchmarking and Sim. of High Perf. Computer Systems (PMBS)*, November 2018.

[41] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarrià-Miranda, Arif Khan, and Assefaw Gebremedhin. Distributed louvain algorithm for graph community detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Vancouver, BC, Canada, May 2018.

[42] Richard Gibbons. A historical application profiler for use by parallel schedulers. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 58–77. Springer, 1997.

[43] S. Habib et al. Hacc: Extreme scaling and performance across diverse architectures. *Commun. ACM*, 60(1):97–104, December 2016. ISSN 0001-0782.

[44] 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.

[45] Greg Hamerly, Erez Perelman, and Brad Calder. Comparing multinomial and k-means clustering for simpoint. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 131–142. IEEE, 2006.

[46] Xiaofei He, Deng Cai, and Partha Niyogi. Laplacian score for feature selection. *Advances in neural information processing systems*, 18, 2005.

[47] Van Emden Henson and Ulrike Meier Yang. Boomeramg: A parallel algebraic multigrid solver and preconditioner. *Appl. Num. Math.*, 41:155–177, 2002.

[48] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[49] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, Victor Basili, Jeffrey K

Hollingsworth, and Marvin V Zelkowitz. Parallel programmer productivity: A case study of novice parallel programmers. In *SC'05*, pages 35–35. IEEE, 2005.

[50] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-learning in neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(9):5149–5169, 2021.

[51] Zhengxiong Hou, Hong Shen, Xingshe Zhou, Jianhua Gu, Yunlan Wang, and Tianhai Zhao. Prediction of job characteristics for intelligent resource allocation in hpc systems: a survey and future directions. *Frontiers of Computer Science*, 16(5):165107, 2022.

[52] Hameed Hussain, Saif Ur Rehman Malik, Abdul Hameed, Samee Ullah Khan, Gage Bickler, Nasro Min-Allah, Muhammad Bilal Qureshi, Limin Zhang, Wang Yongji, Nasir Ghani, et al. A survey on resource allocation in high performance distributed computing systems. *Parallel Computing*, 39(11):709–736, 2013.

[53] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2. `https://www.intel.com/content/www/us/en/content-details/812388/intel-64-and-ia-32-architectures-software-developer-s-manual-volume-3b-syste` `html`, December 2023.

[54] Ioana. Latent Semantic Analysis: intuition, math, implementation. `https://towardsdatascience.com/latent-semantic-analysis-intuition-math-implementation-a194aff870f8`, may 2020.

[55] Engin İpek, Sally A McKee, Rich Caruana, Bronis R de Supinski, and Martin Schulz. Efficiently exploring architectural design spaces via predictive modeling. *ACM SIGOPS Operating Systems Review*, 40(5):195–206, 2006.

[56] Tanzima Z. Islam, Jayaraman J. Thiagarajan, Abhinav Bhatele, Martin Schulz, and Todd Gamblin. A Machine Learning Framework for Performance Coverage Analysis of Proxy Applications. In *SC '16*, pages 46:1–46:12. IEEE Press, 2016. ISBN 978-1-4673-8815-3.

[57] William B Johnson, Joram Lindenstrauss, and Gideon Schechtman. Extensions of lipschitz maps into banach spaces. *Israel Journal of Mathematics*, 54(2): 129–138, 1986.

[58] Joshua Johnston and Greg Hamerly. Improving simpoint accuracy for small simulation budgets with edcm clustering. *Worksh. on Statistical and Machine learning approaches to ARchitectures and compilaTion (SMART08)*, 2008.

[59] Maurice George Kendall. Rank correlation methods. 1948.

[60] P. R. C. Kent et al. QMCPACK: Advances in the Development, Efficiency, and Application of Auxiliary Field and Real-Space Variational and Diffusion Quantum Monte Carlo. *J. Chemical Physics*, 152(174105), 2020.

[61] Y. Kim, J. M. Dennis, and C. Kerr. Assessing Representativeness of Kernels Using Descriptive Statistics. In *CLUSTER*, pages 818–825, Sept 2017.

[62] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[63] JaeHyuk Kwack, John Tramm, Colleen Bertoni, Yasaman Ghadar, Brian Homerding, Esteban Rangel, Christopher Knight, and Scott Parker. Evaluation of performance portability of applications and mini-apps across amd, intel and nvidia gpus. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 45–56, 2021. doi: 10.1109/P3HPC54578.2021.00008.

[64] JaeHyuk Kwack, John Tramm, Colleen Bertoni, Yasaman Ghadar, Brian Homerding, Esteban Rangel, Christopher Knight, and Scott Parker. Evaluation of performance portability of applications and mini-apps across amd, intel and nvidia gpus. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 45–56. IEEE, 2021.

[65] Verónica G Vergara Larrea, Michael J Brim, Arnold Tharrington, Reuben Budiardja, and Wayne Joubert. Towards acceptance testing at the exascale frontier. In *CUG*, 2020.

[66] Jeremy Lau, Erez Perelman, and Brad Calder. Selecting software phase markers with code structure analysis. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 12–pp. IEEE, 2006.

[67] Benjamin C Lee and David M Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ACM SIGOPS operating systems review*, 40(5):185–194, 2006.

[68] Benjamin C Lee, Jamison Collins, Hong Wang, and David Brooks. Cpr: Composable performance regression for scalable multiprocessor models. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 270–281. IEEE, 2008.

[69] Jan-Patrick Lehr, Christian Bischof, Florian Dewald, Heiko Mantel, Mohammad Norouzi, and Felix Wolf. Tool-supported mini-app extraction to facilitate program analysis and parallelization. In *Proceedings of the 50th International Conference on Parallel Processing*, pages 1–10, 2021.

[70] Jingbo Li, Xingjun Zhang, Li Han, Zeyu Ji, Xiaoshe Dong, and Chenglong Hu. Okcm: improving parallel task scheduling in high-performance computing

systems using online learning. *The Journal of Supercomputing*, 77:5960–5983, 2021.

[71] Yuliang Li, Jianguo Wang, Benjamin Pullman, Nuno Bandeira, and Yannis Papakonstantinou. Index-based, high-dimensional, cosine threshold querying with optimality guarantees. *Theory of Computing Systems*, 65(1):42–83, 2021.

[72] Zhenguo Li, Fengwei Zhou, Fei Chen, and Hang Li. Meta-sgd: Learning to learn quickly for few-shot learning. *arXiv preprint arXiv:1707.09835*, 2017.

[73] Paul T. Lin, Michael A. Heroux, Richard F. Barrett, and Alan B. Williams. Assessing a mini-application as a performance proxy for a finite element method engineering application. *Concurrency and Computation: Practice and Experience*, 27(17):5374–5389, 2015. ISSN 1532-0634. cpe.3587.

[74] Zhengchun Liu, Ryan Lewis, Rajkumar Kettimuthu, Kevin Harms, Philip Carns, Nageswara Rao, Ian Foster, and Michael E Papka. Characterization and identification of hpc applications at leadership computing facility. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–12, 2020.

[75] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

[76] Yirong Lv, Bin Sun, Qingyi Luo, Jing Wang, Zhibin Yu, and Xuehai Qian. Counterminer: Mining big performance data from hardware counters. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 613–626. IEEE, 2018.

[77] Amit Mankodi, Amit Bhatt, Bhaskar Chaudhury, Rajat Kumar, and Aditya Amrutiya. Evaluating machine learning models for disparate computer systems

performance prediction. In *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pages 1–6. IEEE, 2020.

[78] Amit Mankodi, Amit Bhatt, and Bhaskar Chaudhury. Performance prediction from simulation systems to physical systems using machine learning with transfer learning and scaling. *Concurrency and Computation: Practice and Experience*, page e6433, 2021.

[79] Giovanni Mariani, Andreea Anghel, Rik Jongerius, and Gero Dittmann. Predicting cloud performance for hpc applications before deployment. *Future Generation Computer Systems*, 87:618–628, 2018.

[80] Satoshi Matsuoka, Jens Domke, Mohamed Wahib, Aleksandr Drozd, Andrew A Chien, Raymond Bair, Jeffrey S Vetter, and John Shalf. Preparing for the future—rethinking proxy applications. *Computing in Science & Engineering*, 24 (2):85–90, 2022.

[81] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.

[82] OE Bronson Messer, Ed D'Azevedo, Judy Hill, Wayne Joubert, Mark Berrill, and Christopher Zimmer. Miniapps derived from production hpc applications using multiple programing models. *The International Journal of High Performance Computing Applications*, 32(4):582–593, 2018.

[83] Mina Naghshnejad and Mukesh Singhal. A hybrid scheduling platform: a runtime prediction reliability aware scheduling platform to improve hpc scheduling performance. *The Journal of Supercomputing*, 76:122–149, 2020.

[84] Philippe Olivier Alexandre Navaux, Arthur Francisco Lorenzon, and Matheus da Silva Serpa. Challenges in high-performance computing. *Journal of the Brazilian Computer Society*, 29(1):51–62, 2023.

[85] Nek. Nek5000 version 19.0. `https://nek5000.mcs.anl.gov`, December 2019.

[86] Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

[87] Jorge Ortiz, David Corbalán-Navarro, Juan L Aragón, and Antonio González. Megsim: A novel methodology for efficient simulation of graphics workloads in gpus. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 69–78. IEEE, 2022.

[88] AMB Owenson, Steven A Wright, Richard A Bunt, YK Ho, Matthew J Street, and Stephen A Jarvis. An unstructured cfd mini-application for the performance prediction of a production cfd code. *Concurrency and Computation: Practice and Experience*, 32(10):e5443, 2020.

[89] Gence Ozer, Sarthak Garg, Neda Davoudi, Gabrielle Poerwawinata, Matthias Maiterth, Alessio Netti, and Daniele Tafani. Towards a predictive energy model for hpc runtime systems using supervised learning. In *Euro-Par 2019: Parallel Processing Workshops: Euro-Par 2019 International Workshops, Göttingen, Germany, August 26–30, 2019, Revised Selected Papers 25*, pages 626–638. Springer, 2020.

[90] 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*, 32, 2019.

[91] Suchita Pati, Shaizeen Aga, Matthew D Sinclair, and Nuwan Jayasena. Seqpoint: Identifying representative iterations of sequence-based neural networks. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 69–80. IEEE, 2020.

[92] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 81–92. IEEE, 2004.

[93] Harish Patil, Alexander Isaev, Wim Heirman, Alen Sabu, Ali Hajiabadi, and Trevor E Carlson. Elfies: Executable region checkpoints for performance analysis and simulation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 126–136. IEEE, 2021.

[94] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[95] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):318–319, 2003.

[96] N.A. Petersson and B. Sjrogreen. Sw4 v2.0. computational infrastructure of geodynamics, 2017.

[97] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J. Comput. Phys.*, 117(1):1–19, March 1995. ISSN 0021-9991. doi: 10.1006/jcph.1995.1039.

[98] Xinxin Qi, Juan Chen, and Lin Deng. Cp 3: Hierarchical cross-platform power/performance prediction using a transfer learning approach. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 117–138. Springer, 2022.

[99] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[100] D. Richards et al. FY18 Proxy App Suite Release. Milestone Report for the ECP Proxy App Project. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA, 2018.

[101] Paul K. Romano, Nicholas E. Horelik, Bryan R. Herman, Adam G. Nelson, Benoit Forget, and Kord Smith. Openmc: A state-of-the-art monte carlo code for research and development. *Ann. Nucl. Energy*, 82:90–97, 2015. URL `https://doi.org/10.1016/j.anucene.2014.07.048`.

[102] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. The earth mover's distance as a metric for image retrieval. *International journal of computer vision*, 40(2): 99–121, 2000.

[103] Jaehun Ryu and Hyojin Sung. Metatune: Meta-learning based cost model for fast and efficient auto-tuning frameworks. *arXiv preprint arXiv:2102.04199*, 2021.

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

[105] Kaushal Sanghai, Ting Su, Jennifer Dy, and David Kaeli. A multinomial clustering model for fast simulation of computer architecture designs. In *Proceedings*

of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, pages 808–813, 2005.

[106] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a" kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st international conference on distributed computing systems workshops*, pages 166–171. IEEE, 2011.

[107] Jennifer M Schopf and Francine Berman. Using stochastic intervals to predict application behavior on contended resources. In *Proceedings Fourth International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'99)*, pages 344–349. IEEE, 1999.

[108] David Sculley. Web-scale k-means clustering. In *Proceedings of the 19th international conference on World wide web*, pages 1177–1178, 2010.

[109] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *ACM SIGPLAN Notices*, 37 (10):45–57, 2002.

[110] Warren Smith, Ian Foster, and Valerie Taylor. Predicting application run times with historical information. *Journal of Parallel and Distributed Computing*, 64 (9):1007–1016, 2004.

[111] Standard Performance Evaluation Corporation. SPEC CPU 2006. `https://www.spec.org/cpu2006/`, 2006. Accessed: 2023-10-01.

[112] Jingwei Sun, Guangzhong Sun, Shiyan Zhan, Jiepeng Zhang, and Yong Chen. Automated performance modeling of hpc applications using machine learning. *IEEE Transactions on Computers*, 69(5):749–763, 2020.

[113] Mohammed Tanash, Huichen Yang, Daniel Andresen, and William Hsu. Ensemble prediction of job resources to improve system performance for slurm-based hpc systems. In *Practice and experience in advanced research computing*, pages 1–8. 2021.

[114] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

[115] Aidan P. Thompson and Christian Robert Trott. A brief description of the kokkos implementation of the snap potential in examinimd. 11 2017. doi: 10.2172/1409290.

[116] Abdul Jabbar Saeed Tipu, Pádraig Ó Conbhuí, and Enda Howley. Artificial neural networks based predictions towards the auto-tuning and optimization of parallel io bandwidth in hpc system. *Cluster Computing*, pages 1–20, 2022.

[117] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis. In *PHYSOR*, 2014.

[118] Uday Kumar Reddy Vengalam, Anshujit Sharma, and Michael Huang. Loopin: a loop-based simulation sampling mechanism. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 224–226. IEEE, 2022.

[119] H. Vincenti and J.-L. Vay. Detailed analysis of the effects of stencil spatial variations with arbitrary high-order finite-difference maxwell solver. *Computer Physics Communications*, 200:147, 2016.

[120] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser,

Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

[121] Nan Wu and Yuan Xie. A survey of machine learning for computer architecture and systems. *ACM Computing Surveys (CSUR)*, 55(3):1–39, 2022.

[122] Abenezer Wudenhe and Hung-Wei Tseng. Tpupoint: Automatic characterization of hardware-accelerated machine-learning behavior for cloud computing. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 254–264. IEEE, 2021.

[123] Roland E Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 84–97, 2003.

[124] Tao Yan, Qingguo Xu, Jiyu Luo, Jingwei Sun, and Guangzhong Sun. Synthesizing proxy applications for mpi programs. *arXiv preprint arXiv:2301.06062*, 2023.

[125] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *ISPASS*, pages 35–44. IEEE, 2014.

[126] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. Rlscheduler: an automated hpc batch job scheduler using reinforcement learning. In *SC20:*

International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–15. IEEE, 2020.

[127] Xinnian Zheng, Lizy K John, and Andreas Gerstlauer. Lacross: Learning-based analytical cross-platform performance and power prediction. *International Journal of Parallel Programming*, 45:1488–1514, 2017.