The Pennsylvania State University
The J. Jeffrey and Ann Marie Fox Graduate School

ADAPTIVE OPTIMIZATION TECHNIQUES FOR HIGH-PERFORMANCE COMPUTING

A Dissertation in
Computer Science and Engineering
by
Gulsum Gudukbay Akbulut

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2024

The dissertation of Gulsum Gudukbay Akbulut was reviewed and approved by the following:

Mahmut Taylan Kandemir
Distinguished Professor of Computer Science and Engineering
Dissertation Advisor & Chair of Committee

Jack Sampson
Associate Professor of Computer Science and Engineering

Vijaykrishnan Narayanan
Robert Noll Chair Professor of Computer Science and Engineering

Anton Nekrutenko
Professor of Biochemistry and Molecular Biology

Antonio Blanca
Associate Professor of Computer Science and Engineering
Director of Graduate Studies

Abstract

The dataset sizes and computing needs of increasingly prevalent high-performance computing (HPC) applications have grown exponentially over the last decade. Moreover, modern computing architectures are evolving with different paradigms, and accelerators have become indispensable parts of computing. Consequently, the imperative for performance optimization for HPC applications and intelligent resource management for evolving architectures has become paramount, particularly within the ones that leverage Machine Learning/Deep Learning (ML/DL).

This dissertation aims to **optimize performance and resource management in modern computing architectures for high-performance computing applications** by targeting it from different angles, which are characterizing and analyzing applications, intelligent job scheduling, dynamic architecture reconfiguration, and compiler optimizations.

Toward this objective, this dissertation consists of four methods from different computation layers that target performance optimization and resource management and answer the following questions: As compute platforms are evolving to meet the needs of state-of-the-art HPC applications, understanding the characteristics of these applications running on emerging architectures is crucial: (i) *How could we efficiently comprehend the microarchitectural characteristics for applications from different areas and characterize them?* As multi-GPU hardware platforms are widespread, finding intelligent job scheduling for HPC applications hosted by these platforms is important: (ii) *How to efficiently and dynamically schedule jobs to GPUs in multi-GPU platforms to ensure fewer context switches and higher performance in time-critical applications?* Memory accesses dominate the end-to-end execution times of HPC applications due to increased computational power demand and dataset size. So, there is a need to optimize the number of data accesses: (iii) *How could we leverage a compiler-guided strategy to decrease costly data accesses and increase the performance of multithreaded applications?* As renewable energy is becoming popular among HPC hardware platforms, optimization for energy-harvesting (EH) processors is vital: (iv) *How to decrease instantaneous energy utilization of EH processors to allow more forward progress using dynamic reconfiguration of microarchitectural resources?*

All these methods can be utilized collectively as a boost for HPC applications running on intelligently managed resources of modern architectures.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

I embarked on this Ph.D. journey as a newly graduated undergraduate from Turkey with little formal education in computer architecture. As I look back, this journey has been transformative, and there are many people without whom this accomplishment would not have been possible.

First and foremost, I am deeply grateful to my advisor, Prof. Mahmut T. Kandemir, for his unwavering support throughout my Ph.D. He trusted me with critical responsibilities and allowed me to explore and grow. His constant motivation and guidance, especially when I felt lost, have been instrumental in shaping me into the researcher I am today.

I would also like to thank my committee members, Prof. Jack Sampson, Prof. Vijaykrishnan Narayanan, and Prof. Anton Nekrutenko. Their insightful feedback and valuable advice during our meetings have significantly improved the quality of my research. I sincerely appreciate their time, effort, and encouragement.

A special thanks goes to Dr. Jihyun Ryoo, Cyan, Dr. Jashwant Gunasekaran, and Sethu, who have been pivotal during my Ph.D. journey. In my early years, when I was just a novice, they provided immense support. The countless research discussions with Cyan greatly enhanced my research abilities. He was a continuous mental support during my journey. Jihyun, my mentor in performance analysis, taught me the nuances of this field and helped me immensely during my first project and internship. Jashwant taught me to be dedicated to excellence in paper writing, showing me that only the best would suffice. Sethu's calm and reassuring presence helped me navigate the nervous early stages of my Ph.D. He was also the one who taught me how to use an essential architectural simulator.

I am fortunate to have been surrounded by amazing friends and colleagues, including Shruti, Siddhartha, Soheil, Burak Gulhan, Burak Topcu, Yilin, Scott, Myungjun, Jun-Liang, Omer Faruk, and Yuanqing, who made our lab feel like a second home. Outside the lab, my friends Esra, Deniz, Oguzhan, Ceren, Ruya, and Ecem made State College feel like home. My heartfelt thanks go to my childhood friends Derya, Ekin, Julia, Cigdem, and Kaya, whose constant mental support has been invaluable.

I want to express my profound appreciation to my father, Dr. Ugur Gudukbay, who gave me the best educational opportunities and supported me academically and emotionally. His unwavering belief in me shaped my persistence and dedication to perfection. My mother, Leyla Gudukbay, has always been my source of strength. She raised me to be resilient and taught me never to give up. My brother Goktan has been a

constant source of encouragement throughout my life. I am also profoundly grateful to my maternal and paternal grandmothers, who have always been there for me.

Finally, I want to express my deepest gratitude to my husband, Dr. Suat Akbulut, without whose love and support I would not have made it through this Ph.D. journey. He has been my mentor, most excellent motivator, and anchor. His unwavering belief in me pushed me to explore opportunities I never thought possible, and his encouragement made me stronger. During this Ph.D., our son Aslan came into our lives, bringing me immense joy and renewing my purpose. Suat and Aslan have made this journey more meaningful, and I am forever indebted to them for their love and support.

I also enjoyed my time at Intel and AMD during my internships, where I had the opportunity to learn about many cutting-edge areas. I want to thank Dr. Meena Arunachalam for making this experience possible. Thank you to everyone who has been a part of this journey.

Chapter 4 contains material from "GYAN: Accelerating bioinformatics tools in galaxy with GPU-aware computation mapping." by Gulsum Gudukbay, Jashwant Raj Gunasekaran, Yilin Feng, Mahmut T. Kandemir, Anton Nekrutenko, Chita R. Das, Paul Medvedev, Björn Gruning, Nate Coraor, Nathan Roach, Enis Afgan, which appears in IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2021. The author of this dissertation is the primary investigator of this paper. The material in Chapter 4 is copyrighted by the Institute of Electrical and Electronics Engineers (IEEE) DOI 10.1109/IPDPSW52791.2021.00037. Permission can be requested and approved online through RightsLink, an automated permission-granting service from the Copyright Clearance Center. This research was supported by NSF award #1931531. We thank NSF Chameleon Cloud project CH-819640 for its generous compute grant.

Chapter 5 contains material from "Data Recomputation for Multithreaded Applications" by Gulsum Gudukbay Akbulut, Mahmut T. Kandemir, Mustafa Karakoy, and Wonil Choi, which appears in IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2023. The author of this dissertation is the primary investigator of this paper. The material in Chapter 5 is copyrighted by the Institute of Electrical and Electronics Engineers (IEEE) DOI 10.1109/ICCAD57390.2023.10323776. Permission can be requested and approved online through RightsLink, an automated permission-granting service from the Copyright Clearance Center. This work was supported by NSF grants 2211018, 2008398, 1822923, and 1908793. This work was also supported by the MOTIE (1415181081), KSRC (20019402), and IITP (2022-0-00117, RS-2023-00221040). Wonil Choi (author) is supported by a research fund from Hanyang University (HY-2021-2596).

This research was supported by NSF awards #1931531, 2211018, 2008398, 1822923, and 1908793. Any opinions, findings, conclusions, or recommendations expressed in this document are those of the author and do not necessarily reflect the views of the National Science Foundation.

Dedication

To my son Aslan for filling my life with immeasurable joy and purpose. As you grow, may you always chase your dreams with the same determination and passion that guided me through this journey. This is for you, with all my love.

Chapter 1
Introduction

In recent decades, high-performance computing (HPC) has experienced significant evolution, moving beyond traditional supercomputers to a broader array of processors [185]. Modern HPC applications frequently integrate machine learning (ML) and deep learning (DL) kernels [15, 17, 23, 25, 29, 41, 63, 86, 87, 119, 141, 170, 186, 192, 194], driving a shift in computational demands. Due to this shift, computer architectures evolve to keep pace [1, 9, 11, 21, 48, 65, 142], which increases the heterogeneity in hardware [32]. The increasing heterogeneity in hardware architectures – using CPUs, GPUs, and emerging architectures like chiplet designs, energy harvesting processors, and 3D memory – requires new optimization approaches. Existing performance optimization techniques must adapt to these evolving architectures. In parallel, hardware advancements continue to reshape computational paradigms. Understanding and aligning with these shifts is essential for resource management, job scheduling, and performance in HPC applications.

The Problem: Due to the shift in the computational demands of HPC applications, the heterogeneity in the hardware in which the evolutionary HPC applications are hosted increased significantly. However, as the use of CPUs, GPUs, and emerging architectures such as chiplet-based processors and 3D memory increased, the existing performance and resource allocation optimization techniques started to have shortcomings. The shift in computational demands necessitates a deeper understanding of microarchitectural characteristics across diverse hardware platforms. So, the first challenge is understanding the application characteristics robustly to determine the architectural shortcomings. Intelligent job scheduling is critical for diverse heterogeneous hardware platforms (i.e., multi-GPU platforms) that host HPC applications. The second challenge is ensuring optimal job scheduling is performed on these platforms. As HPC applications evolve, the memory accesses start to dominate the execution times. Data recomputation is of paramount interest in overcoming this problem. Moreover, as HPC centers leverage

renewable energy resources such as solar energy, energy harvesting processors emerge and require better (i.e., dynamic) resource allocation and configuration.

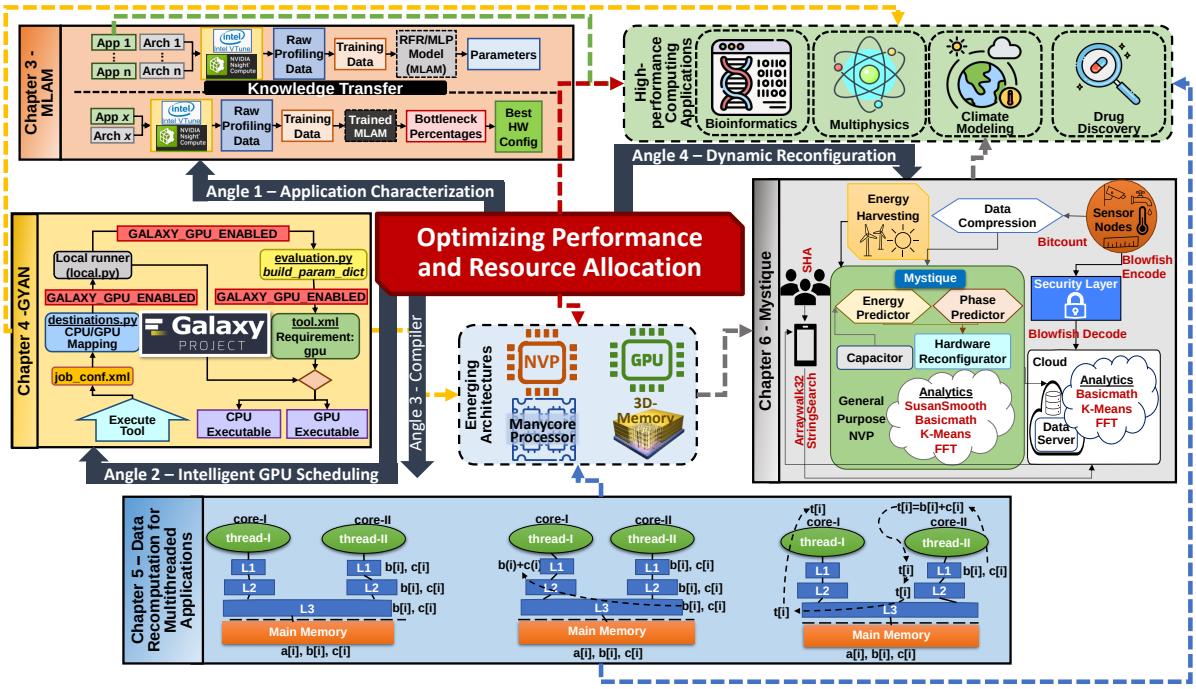


Figure 1.1: The vision of the dissertation.

Contributions: This dissertation targets optimizing the performance and resource management in modern computer architectures for HPC applications from four different angles, as seen in Figure 1.1. In the first angle, we target optimizing the performance and resource management at the application level by statically characterizing different applications using detailed microarchitectural analysis. This goal is pivotal in identifying potential areas for optimization in emerging architectures on which HPC applications are hosted. HPC platforms have become increasingly heterogeneous and typically include multiple GPUs. So, intelligent dynamic scheduling is crucial to optimize the performance of HPC applications in multi-GPU-supported settings. We tackle the performance and resource allocation optimization problem from the GPU job-scheduling angle.

As HPC evolves, applications become more and more dominated by memory access latencies. To solve this, we reduce the number of memory accesses using a compiler strategy specifically targeting multi-threaded applications, thereby improving overall performance. In recent years, the hardware platforms that HPC applications are hosted on started to migrate to green computing, which utilizes renewable energy resources (i.e., energy harvesting). We must consider fine-grained dynamic application phase character-

ization and hardware reconfiguration, particularly for emerging architectures, such as energy harvesting (EH) and non-volatile processors (NVPs). In NVPs, instantaneous energy utilization directly impacts computational progress. We propose dynamically reconfiguring EH NVPs to maximize forward progress.

Contribution I (C-I) Leveraging Machine Learning to Perform Microarchitectural Bottleneck Analysis: The changing computational requirements of HPC systems call for a thorough understanding of microarchitectural features across various hardware platforms. For this purpose and to find optimization areas for performance and resource allocation, employing ML/DL-based approaches for bottleneck analysis to provide the best execution environment for different applications is vital. Even though many bottleneck analysis tools exist [67, 144], the knowledge obtained from these tools is *not* transferable to other hardware or applications. To this end, in *Chapter 3*, we propose **MLAM**, an architectural bottleneck deduction model that employs ML/DL algorithms to identify architectural bottlenecks from raw data and establish connections between the architectural bottlenecks and the performance metrics. With MLAM, the bottlenecks of the incoming application are robustly predicted, allowing opportunities for knowledge transfer to different architectures, personalized architectural configurations, better resource management, and performance optimization.

Contribution II (C-II) On-the-Fly Job Scheduling to Instantaneously Suitable GPUs: As HPC applications become more popular and compute-heavy, the use of heterogeneous compute elements increased significantly. Most of the HPC heterogeneous compute platforms contain multiple GPUs. There are many HPC application domains where GPUs have become indispensable, such as bioinformatics, nuclear physics, and DL [25, 44, 62, 128]. Due to the extensive usage of GPUs, especially in platforms like "Galaxy" Project [46], addressing inefficient resource management is paramount for optimizing the performance of HPC applications. To this end, we propose a strategy in *Chapter 4* that allows efficient allocation of GPUs to jobs in multi-GPU infrastructures.

Contribution III (C-III) Optimizing the Performance of Multi-threaded Applications Using a Compiler-Guided Strategy: As HPC applications solve extensive problems, they are becoming more memory intensive. As memory accesses are dominating HPC application runtimes due to increased memory intensity, data locality has been a pressing problem. While many locality optimization [79, 90, 104, 108, 162, 205] and other solutions such as accelerator design [14, 84, 214] and near data computing [47, 80, 139, 183] have been proposed, due to increasing dataset sizes and complexity of emerging architectures, the benefits of the previously proposed methods

have been limited. Data recomputation is used to mitigate data locality issues. Data recomputation [3, 4, 193] eliminates costly data accesses by replacing each such access with multiple less costly data accesses plus extra computation. Previous approaches for recomputation do not handle multi-threaded applications and do not explore the role of different caching policies/strategies in improving the effectiveness of data recomputation. To fill this gap, in *Chapter 5*, we propose a compiler-guided strategy that balances data locality optimization and recomputation optimization to minimize data access latencies in multi-threaded application executions and a new compiler-guided cache replacement strategy to optimize performance.

Contribution IV (C-IV) Increasing Performance Using On-the-Fly Architectural Resource Reconfiguration: Our first study statically characterizes applications to create a robust bottleneck analysis model. Different portions of domain applications, such as IoT, may have different architectural characteristics. IoT applications leverage Wireless Sensor Networks (WSNs), which should involve low maintenance and strive for continuous forward progress. EH NVPs are suitable for this purpose, but current EH NVPs allow minimal and course-grained dynamism [112, 113], barely explore other dynamic reconfiguration options, and they are program agnostic even though predictable runtime application behavior is observed. This behavior hinders application forward progress. If the WSN gateways were aware of, for example, memory boundedness, dynamically reconfiguring hardware would increase forward progress. Towards this, in *Chapter 6*, we dynamically reconfigure the cache according to the observed microarchitectural behavior of the phases.

Dissertation Organization: The background and related work is presented in Chapter 2. Chapter 3 presents a machine learning-based architectural bottleneck deduction model for application characterization. Chapter 4 presents an intelligent and dynamic GPU allocation strategy for time-sensitive jobs in multi-GPU systems. Chapter 5 proposes a compiler-guided strategy to optimize resource allocation and performance for high-performance computing applications. Chapter 6 presents an application phase characterization framework to reconfigure non-volatile processor architectures dynamically. Lastly, the conclusions, summary of key results, and potential future research are provided in Chapter 7.

Chapter 2
Background and Related Work

This chapter provides a brief background and related work about (i) application characterization, (ii) dynamic application phase characterization and hardware reconfiguration, and (iii) reconfiguring cloud infrastructures.

2.1 Background

First, we must understand how CPU architecture works when performing architectural profiling to characterize applications. The CPU has two parts: *front-end* and *back-end*. The front end is in the initial part of the pipeline and is responsible for feeding operations to the back end, which executes the operations using compute units and memory. The branch predictor predicts the next instruction's address and fetches it from the L1I cache. Later, the instruction is decoded into μOps, and then these μOps are scheduled in the out-of-order scheduler. The scheduler dispatches the ready μOps to the execution units, and when these μOps finish executing, they are *retired*. In this sequence of events, the front-end related bottlenecks are referred to as "`FE`," back-end ones as "`BE`" (where they are divided as "`BE:CORE`" for core/compute related and "`BE:MEM`" for memory-related bottlenecks). The retiring μOp percentage is referred to as "`RET`," and the branch prediction and other speculation-related bottlenecks are referred to as "`SPE`" in the bottleneck analysis and application characterization portion of this thesis.

The next step for characterization is application phase prediction, which tracks the changes in the execution behavior of applications. In this context, a "phase" is defined as a set of contiguous portions of program execution with similar behavior, irrespective of temporal adjacency [99]. Phase classification divides a set of intervals of a specific metric into phase intervals with similar architectural/software behavior. Phase prediction predicts the behavior of the upcoming period in the application runtime [99].

Since the first two studies focus on CPU architectures, our third study also focuses on optimizing performance and resource allocations for platforms that use GPUs. GPUs are powerful computation components for computationally challenging and demanding workloads. GPUs are specialized to deliver remarkable performance, especially in the cases of parallel computations, because GPU is a hardware component with high memory bandwidth, high arithmetic capabilities, and is highly parallel-programmable. Due to these properties, GPUs yield high-performance benefits over CPU counterparts. Since GPUs have substantially developed over the years due to latency and throughput being increasingly important in many application domains, the recent NVIDIA GPUs deliver thousands of GFLOPS [142]. This advancement created a noticeable need to optimize performance and energy efficiency [145].

We then focus on compiler optimizations to improve performance and resource allocation in HPC. Data locality is one of the remaining problems in many application domains, especially in HPC. Even though many optimizations for data locality have been proposed, the increasing complexity of architectures and dataset sizes limit the potential of these optimizations. Data recomputation is utilized to remedy this problem. Data recomputation [3, 4, 193] is an approach that eliminates costly data accesses by replacing them with multiple less costly data accesses plus extra computation. Even though data recomputation has been explored before, the previous approaches have yet to consider inter-thread interactions and the role of different caching policies in improving the effectiveness of data recomputation. Data recomputation may be very beneficial in many-core systems. However, it may not be always legal. Moreover, in many-core systems, threads can recompute data on behalf of each other, which can contribute to significant performance improvements.

2.2 Related Work

Architectural Analysis and Characterization of Workloads Using ML/DL: The first component necessary to perform architectural reconfiguration optimizations from cores to cluster-level infrastructures is offline/static workload characterization using learning. Prior work includes Stengel et al. [180], which employed a cache-memory performance model to quantify the bottlenecks in stencil computations, and Hoefler et al. [59] predicted the performance of parallel applications. Some prior works that leverage ML/DL to perform *bottleneck analysis* for different applications include [34, 209, 210]. Prior characterization works include the following: Prakash et al. [152] presented a

characterization of the SPEC CPU benchmarks targeting an Intel multiprocessor system, and Ryoo et al. [166] analyzed the bottlenecks of DL applications on CPUs, GPUs, and Xeon Phi-based platforms. Zhu et al. [215] performed profiling on DL applications across different infrastructures.

Novelty of Our Study: These works propose architectural analysis frameworks both for processors and cloud computing-level infrastructures. However, none present an architectural profiling deduction ML/DL model that captures the complex mathematical relationships between hardware counters and architectural bottlenecks and speeds up the architectural profiling bottleneck deduction for hardware designers. Moreover, our tool allows knowledge transfer, which is not possible when using VTune [67] or NVIDIA Nsight Compute [143]. To this end, we propose a machine learning-based microarchitectural bottleneck analysis in Chapter 3.

Adding GPU and Multi GPU Support to the Galaxy Framework: Intelligent job scheduling is paramount in large frameworks such as the Galaxy Project [46]. Since Galaxy did not take advantage of GPU resources available on the supercomputers on which it was hosted, we added this support along with intelligent multi-GPU scheduling. Among all other job management features Galaxy has, "Galaxy CloudMan" [2] was proposed to allow researchers to manage an arbitrarily sized compute cluster on Amazon EC2. Even though the EC2 instance has a GPU, Galaxy could not use it without our support. Besides the cloud computing support, there have been several other application-level enhancements for the Galaxy Framework. PyPasWAS [203] is a Python-based multi-core GPU and CPU sequence alignment tool. The Web-based platform [116] is an infrastructure for NGS analyses for Galaxy with GPU support.

Novelty of Our Study: For the CloudMan support, even though the EC2 instance has a GPU, Galaxy could not take advantage of it without our support. While PyPasWAS achieves a 33× speedup in execution time using GPUs for alignment, it is not a platform for running different sequence alignment tools, and it does not provide infrastructure nor a GPU interface with Galaxy. The Web-based platform [116] is an in-house framework that uses Galaxy as a middleware to run jobs along with Slurm [208] scheduler; it does not add a bare-metal GPU interface nor intelligent GPU scheduling. We address this issue by employing on-the-fly job scheduling to instantaneously suitable GPUs, as described in Chapter 4.

Compiler-Guided Strategy to Optimize HPC Workloads: Akturk and Karpuzcu [3] propose replacing a load instruction with a sequence of instructions to recompute a loaded data value. Later, Akturk and Karpuzcu [4] analyze the computation vs.

communication tradeoff of recomputation. Sakalis et al. [168] use recomputation of data that resides in off-chip memory in isolation. On the compiler side, Koc et al. [92] propose recomputing the data required from a bank and avoiding unnecessary bank activations. Later, Koc et al. [91] propose an algorithm to eliminate off-chip memory accesses to Scratch-Pad Memory (SPM) with recomputation. Briggs et al. [19] propose recomputing the value of loaded data when it is cheaper if it cannot be kept in a register to avoid expensive storage. Punjani [153] proposes "rematerialization" of spilling values in register-starved architectures to improve performance. Tang et al. [193] present Computing with Near Data (CND).

Novelty of Our Study: In none of these works mentioned above, threads recompute on behalf of other threads. Moreover, none of the works explore cache policy optimizations to aid recomputation. We address these issues in Chapter 5.

Application Phase Prediction Approaches: Phase prediction is also a crucial improvement component that can be used to reconfigure hardware. It mainly uses online/on-the-fly application characterization for each phase of a specific application. There are many phase prediction strategies [31] including the Basic Block Vectors (BBV) approach, which keeps track of execution frequencies of basic blocks touched in an execution interval, and a phase change happens when the Manhattan distance between two BBVs exceeds a threshold. Another predictor is Working Set Signatures (WSS) [31], where a program phase is defined by thresholding the relative distance between WSSs. In the Conditional Branch Counter predictor, a counter counts the dynamic conditional branches executed, and a *phase change* is defined as the difference in branch counts of any consecutive intervals beyond a threshold [31]. Isci et al. [69] propose the Global Phase History Table Predictor where the phases are categorized according to $memory\_operations/\mu Op$ metric on the fly, and upcoming phases are predicted using a phase history table.

Using Phase Prediction for Processor Reconfiguration: Phase prediction information can be used to reconfigure processors. In [52], a reconfigurable VLIW processor with varying issue widths is proposed based on predicting the number of active data paths. In [13], the authors use a phase detection algorithm that dynamically utilizes hardware counters to reconfigure the on-chip memory hierarchy. In CHAMELEON [94], the authors propose a reconfigurable heterogeneous memory system, where the hardware dynamically switches memory regions between hardware or OS-managed part of memory and cache modes. Ranganathan et al. [158] proposed a reconfigurable cache to support memory functionalities specialized for media processing. Kumar et al. [97] examined

phase characterization-directed timing-associativity-reconfigurable L1 caches. In [179], the authors proposed a reconfigurable cache hierarchy for parallel workloads, targeting CMPs. Kim et al. [89] propose reconfiguring the cache into application-specific functional units for accelerating code patterns. In [93], a Coarse-Grained Reconfigurable Architecture is introduced, which leverages the variable usage of functional units using application phases. In Spendthrift [113], an ML-based frequency and resource scaling policy is presented for improving forward progress by 2.08× in EH NVPs.

Novelty of Our Study: In all of these prior works, the reconfiguration optimizations are forgotten across different executions. The optimizations must be re-calculated, showing how repetitive optimization calculations could waste time and energy. We address this problem in Chapter 6.

Chapter 3
Machine Learning-based Microarchitectural Bottleneck Analysis

3.1 Introduction

Machine Learning (ML) and Deep Learning (DL) recently gained substantial attention, primarily due to the remarkable accuracy improvements they have demonstrated. The effectiveness of ML/DL algorithms has been established in application domains such as image recognition [57], natural language processing [85], autonomous driving [39], bioinformatics [103], and computational/quantum chemistry [157]. As a result of this prominence, most high-performance computing applications contain ML/DL kernels, which causes a shift in computational needs. This shift and the emergence of diverse hardware problems necessitate a deeper understanding of microarchitectural characteristics across these hardware platforms to tune architectures for optimum execution environments for diverse applications. Existing works [24, 74, 198] emphasize the importance of tuning architectural components for the best execution of ML/DL algorithms. There has been much *less* prior work on employing ML/DL-based approaches for bottleneck analysis, i.e., using ML to identify the limiting factors to be tuned to provide the best execution environment for different applications.

An architectural *bottleneck* is a hardware component stalling the execution of an application due to its imbalanced utilization compared to other hardware units. A few existing works in this area (e.g. [34, 70, 209, 210] and the references therein) focus on ML-based architecture profiling models. Even though many bottleneck analysis tools exist [67, 144], the knowledge obtained from these tools is *not* transferrable to other hardware or applications. Moreover, to our knowledge, *no* prior work proposes a bottleneck deduction model that employs ML and DL algorithms and captures the

complex relationship between the hardware metrics and the architectural bottlenecks. This work aims to fill this critical gap. We propose and experimentally evaluate an architectural bottleneck deduction model, *MLAM*, that employs ML/DL algorithms to identify architectural bottlenecks from raw data and establish connections between the architectural bottlenecks and the performance metrics. Our contributions are as follows:

- We analyze 44 applications from different benchmark suites: 30 SPEC CPU 2006 applications, 10 ML applications, and 4 DL applications. We identify the hotspot kernels of each application and their different architectural bottlenecks for a better understanding of the applications as well as their notable functions/kernels[1], which are used as input 'dataset' for MLAM.
- We establish the *relationship* between 'hardware metrics' and 'architectural bottlenecks' by calculating the *correlation coefficients* between them. This information is used to decide the initial feature list for MLAM.
- We design an *architectural bottleneck analysis model* (MLAM) that can *predict* the architectural bottlenecks with an ML strategy Random Forest Regression (RFR) or a DL strategy Multi-Layer Perceptron regression (MLP). Our model can achieve 0.70 (RFR) and 0.72 (MLP) R^2 inference accuracy by updating features iteratively and employing permutation importance.
- We evaluate MLAM's bottleneck percentage prediction and ordering accuracy using our proposed similarity metrics for four different architectures and for training MLAM with multi-architecture data. The predictions result in 0.73 average similarity to the proper ordering and percentages for different architectures, and multi-architecture training results in 0.72 similarity, showing that MLAM helps tune architectures for optimal performance.

We gathered data about bottlenecks and hardware metrics using VTune [67][2], targeting different Intel architectures. By employing MLAM, one could achieve further benefits compared to other profiling methods like the Roofline model [204] or directly using VTune (or a similar tool). So, MLAM allows us to (i) recognize the relationship between the hardware metrics and the architectural bottlenecks, (ii) understand the critical hardware metrics when predicting the architectural bottlenecks, (iii) perform multiple bottleneck predictions for different test applications/architectures using a trained instance of MLAM, and (iv) most importantly, MLAM allows robustly deducing bottlenecks from raw profiling data to tune architectures to suit different application domains for better

[1]We use the terms 'kernel' and 'function' interchangeably.

[2]VTune is a performance analysis tool for x86-based systems running Linux or Microsoft Windows operating systems.

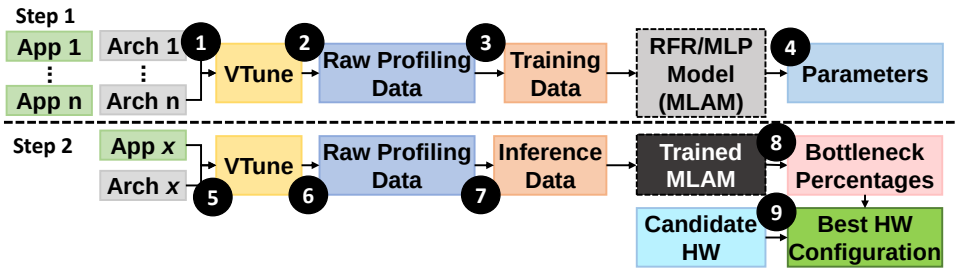


Figure 3.1: MLAM aiding in bottleneck knowledge transfer for unseen applications and unavailable architectures.

performance. So, MLAM would be very useful in generalizing (transferring) bottleneck analysis knowledge to different applications/architectures while learning from a limited data set. Section 3.2 provides more details on the use case.

3.2 Example Use-Case

MLAM can be used in multiple ways. *First*, it can predict the architectural bottlenecks from raw hardware metrics and discover *relationships* among them. These predictions can be beneficial to understand the underlying reasons behind bottlenecks *to develop more optimized libraries*.

Second, unlike existing architectural analysis tools that identify performance-limiting hardware elements, MLAM allows *transferring the knowledge* obtained from these tools to unavailable applications or architectures *to develop more optimized architectures*, as seen in Figure 3.1. So, MLAM can aid in choosing suitable (new or previously encountered) architectures for (new or previously encountered) applications to ensure adequate service without a significant engineering effort margin. With trained MLAM and the test data from an *unseen* target application, which can be gathered using the strategy in Section 3.7.1, the user can perform a fine-grained bottleneck analysis for target applications to optimize them. One *does not need to retrain MLAM once trained*. Therefore, performing architectural profiling on an application and deducing bottleneck information from raw profiling data is slower and more tedious than just inputting the raw profiling data into a trained MLAM instance and robustly obtaining bottleneck analysis with much less effort.

Third, MLAM can be applied to different CPU architectures (refer to Section 3.7.5) and to GPUs. The main change when re-purposing MLAM to a different architecture is in the data collection strategy, e.g., using NVProf [144] or Nsight Compute [143]. Then,

one can efficiently train MLAM for other architectures.

Lastly, if desired, one can use a different ML/DL algorithm by replacing the current algorithms employed in MLAM (RFR and MLP) with a new algorithm (such as SVM, KNN, XGBoost or DL algorithms such as Transformer [197], BERT [85] or EfficientNet [191]) along with the corresponding importance calculation logic.

For example, consider three applications, A, B, and C, and three architectures, X, Y, and Z (①). We perform architectural profiling using VTune with application A running on architecture X, B on Y, and C on Z and obtain raw profiling data (②). We then perform the necessary preprocessing steps and feed this data into MLAM for training (③), which outputs optimal parameters and features (④). Later, when we have a previously unseen application D running on a new architecture W (⑤), we obtain the raw profiling data using VTune (⑥). This data is preprocessed to gather the inference data (⑦). The preprocessed inference data are fed into MLAM to obtain the bottleneck information (⑧), which will then be used to find the best hardware configuration among candidate configurations (⑨). Another use case would be changing the application or architecture on step ⑤. One could use a previously *seen* application running on an *unavailable* architecture or, conversely, an *unseen* application running on a previously *seen* architecture. Since MLAM's goal is to transfer knowledge, all these use cases would be possible.

3.3 Background

3.3.1 Target Architecture Template

While our approach applies to different types of architectures (such as Intel Ice Lake, Cascade Lake, and Broadwell), we discuss one representative architecture in this section. Figure 3.2 describes the microarchitecture of Intel Skylake [33]. This CPU (and the other Intel CPU architectures mentioned before, similarly) consists of two parts: *front-end* and *back-end*. The front-end resides in the early stage of the processor pipeline and feeds operations into the back end. The back-end executes the allocated operations with the help of compute units and the memory subsystem. First, the branch predictor predicts the address of the next instruction and fetches that instruction from the L1I (L1 instruction) cache. Then, the decoders decode the instructions into μOps and fill the allocation queue. The μOps are allocated to the out-of-order scheduler, which dispatches the ready μOps to their expected execution units. When the μOps complete their execution, they are *retired* in program order.

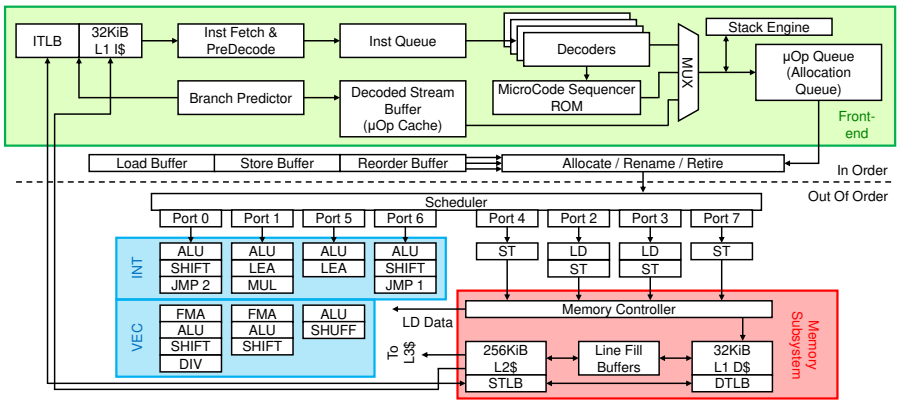


Figure 3.2: Microarchitecture of Skylake CPU (based on [33]).

3.3.2 Target Benchmarks

In this work, we consider *three types of benchmarks*:

3.3.2.1 SPEC CPU 2006

SPEC CPU 2006 benchmark suite [58] is widely used to evaluate the performance of a given target architecture. SPEC CPU 2006 includes representative benchmarks from real-life applications, which makes them suitable for performing a comprehensive performance evaluation for a given architecture. This suite includes *integer* and *floating-point* workloads; we used 30 workloads for our experiments.

3.3.2.2 Machine Learning

We used ten ML applications that are implemented using Python and Scikit-learn API [20, 149]: *Linear Regression* (Amazon Access Samples dataset [136]), *Logistic Regression* (Toxic Comment Classification Challenge dataset [78]), *Naive Bayes classification* (EMNIST [26]), *K-Means* (New York City Taxi Trip Duration [76]), *K-Nearest Neighbors* (EMNIST [26]), *Support Vector Machine* (EMNIST [26]), *Random Forest Classification and Regression* (GTZAN [77] and Craigslist Used Cars [160]), *XGBoost* (GTZAN [77]) and *Light Gradient Boosting Machine* (Home Credit Default Risk [75]).

Table 3.1: Hardware setup

Architecture	**Broadwell**	**Skylake**
Core & Cache	Intel Xeon E5-2686 v4 @ 2.30GHz 36 cores, 72 threads (hyperthreading) L1I cache 18x32 KB 8-way set associative (SA) L1D cache 18x32 KB 8-way SA L2 cache 18x256 KB 8-way SA L3 cache 18x2.5 MB 20-way SA, shared	Intel Core i7-6700K @ 4.00GHz 4 cores, 8 threads (hyperthreading) L1I cache: 4x32KB 8-way SA 1D cache: 4x32KB 8-way SA L2 cache: Unified, 4x256KB, 4-way SA L3 cache: 4x2MB, 16-way SA
Memory System	Memory controller (MC): DDR4, 4 channels Main memory (MM): 512GiB, DDR4	MC: DDR3L, DDR4, 2 channels MM: 2x8GiB, DDR4
SoC	QuickPath Interconnect	Ring Interconnect
Architecture	**Cascade Lake**	**Ice Lake**
Core & Cache	Intel Xeon Platinum 8252C @ 3.80GHz 32 cores, 64 threads (hyperthreading) L1I cache: 12x32 KB L1D cache: 12x32 KB 8-way SA L2 cache: 12x1 MB 16-way SA L3 cache: 24.75 MB 11-way SA	Intel Xeon Platinum 8375C @ 2.90GHz 32 cores, 64 threads (hyperthreading) L1I cache: 32x32 KB L1D cache: 32x48 KB 12-way SA L2 cache: 32x1.25 MB 20-way SA L3 cache: 54 MB 12-way SA
Memory System	MC: 2 DDR4, 3 channels MM: 192GiB, DDR4	MC: 8 DDR4, 2 channels MM: 192GiB, DDR4
SoC	Ultra Path Interconnect	Ultra Path Interconnect

3.3.2.3 Deep Learning

We used Caffe [73][3] to run our DL applications: *Alexnet* (ILSVRC 2012 [165] dataset), *LeNet* (MNIST [100]), *GoogleNet* (ILSVRC 2012 [165]), and *Fine Tune Flickr Style* (Flickr style [81]).

3.4 Experimental Setup

Table 3.1 lists the hardware components used in this study. We use four different architectures: Broadwell, Skylake, Cascade Lake, and Ice Lake. We use Linux 5.4.0 with GCC 9.3.0, Python 3.7.9, MKL 2020.2, VTune 2020.2.0, Caffe 1.0, Scikit-learn 0.24.2, Pandas 1.1.1, and Numpy 1.19.1.

3.5 Benchmark Analysis

Figures 3.3, 3.4, and 3.5 show the hardware boundedness (bottlenecks) brought up by different resources at a function-level granularity. Following the terminology by Intel [67], the term 'boundedness' is equivalent to the term 'bottleneck'; that is, an application/kernel is said to be *bounded* by a specific hardware component if that

[3]Caffe is a framework that provides commonly used computations (convolution, pooling, ReLU, and softmax).

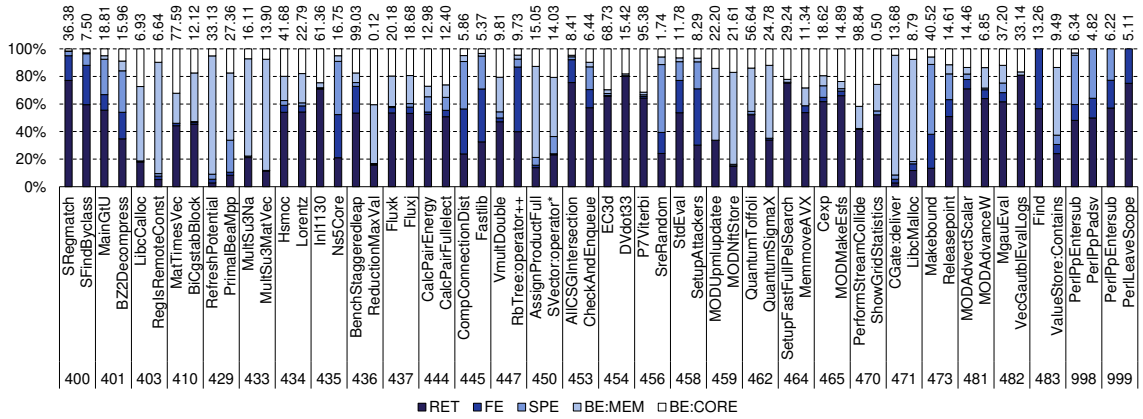


Figure 3.3: Distribution of bottlenecks (y-axis) (RET: retiring instruction percentage; FE: front-end bottlenecks; SPE: speculative bottlenecks; BE:MEM: back-end memory bottlenecks; BE:CORE: back-end core bottlenecks) in the top-2 hotspot functions (x-axis) in SPEC CPU suite. The top axis numbers show the percentage of the kernel's execution time within each benchmark.

specific hardware component is the *bottleneck* for the application/kernel. We also talk about 'fraction/percentage of boundedness,' indicating how much bottleneck a given hardware component is for an application/kernel, i.e., how many of the cycles of the application/kernel experience stall in that component.

The CPU architecture can be divided into *front-end* and *back-end*. The front-end is for instruction fetch, decode, and handover μOps to the back-end. The back-end communicates with the cache and the memory controller for the data and executes the instructions by utilizing ALUs, vectorization units, and memory resources. *FE* and *BE* in Figures 3.3, 3.4, and 3.5 correspond to the percentage of stalls caused by the front-end and back-end, respectively. The back-end-related bounds include the memory-related (`BE:MEM`) and core-related (`BE:CORE`) bottlenecks. *SPE* captures the stalls caused by wrong speculation, such as branch misprediction or machine clearing, and retiring (`RET`) corresponds to the pipeline slots that have been used for performing meaningful work. A high value for the *RET* metric means that the architecture's theoretical μOps retired per cycle is almost achieved (i.e., *resources are well-utilized, higher the better*).

3.5.1 SPEC CPU Benchmarks

Figure 3.3 shows the distribution of the architectural bottlenecks of the top two hotspot kernels in the SPEC CPU benchmarks. For instance, SRegmatch and SFindByclass are the top two hotspot functions in `400.perlbench`. The execution of these functions is relatively simple, which leads to a high retiring percentage and almost negligible back-end

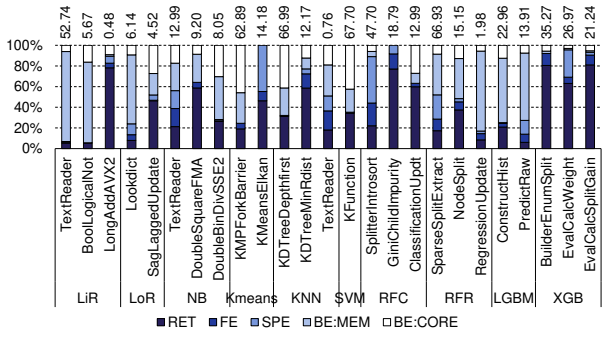


Figure 3.4: Distribution of bottlenecks (y-axis) in notable ML kernels (x-axis) and their execution time in percentage (top axis) within each application.

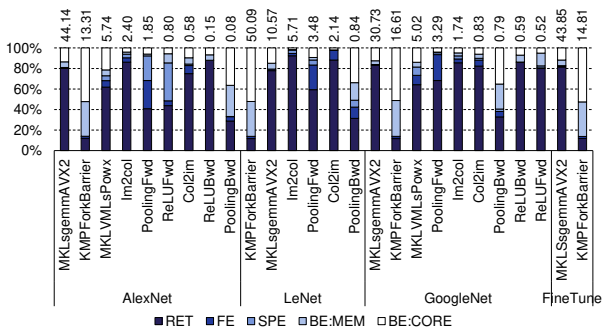


Figure 3.5: Distribution of bottlenecks in notable DL kernels and their execution time percentage within each application.

issues. They spend more time on the front-end side, particularly on instruction decode, so they are FE-bound. For `401.bzip2`, which performs compression and decompression of input files, both of its top hotspot functions – `MainGtU` and `BZ2Decompress` – suffer from a severe SPE bottleneck. MainGtU is used in block-sorting lossless data compression algorithm, and `BZ2Decompress` is for the `bzip2` decompression. Both functions include many conditional statements, which increase misspeculation rates. The `MatTimesVec` function in `410.bwaves` (fluid dynamics) and the top two hotspot functions from 470.lbm (Lattice Boltzmann Method to simulate incompressible fluids) all experience the `BE:CORE` problem because they perform heavy, high-density matrix computations. In comparison, the top two hotspot functions in `429.mcf` (mass public transportation scheduling), `433.milc` (physics/quantum chromodynamics), and `471.omnetpp` (large Ethernet network simulation) are all `BE:MEM` bound. The hotspot functions of `429.mcf` update the node's potential value or cost by traversing the nodes. The `MultSu3Na` and `MultSu3MatVec` functions in `433.milc` take two `su3_matrix` (3×3 matrix) as input and return another `su3_matrix` after matrix computations. All these kernels perform massive data accesses, so they are highly `BE:MEM` bounded.

3.5.2 Machine Learning (ML) Benchmarks

Figure 3.4 shows the notable kernels in ML applications and their bottlenecks. The notable kernels are hotspot functions directly related to the core algorithm or the data-loading phase of the application. For example, in Linear Regression (LiR), since the computation volume is low, reading input data (`TextReader`) occupies most of its execution time. `TextReader` only reads the data from memory; so, it is `BE:MEM` bound. LongAddAVX2

(AVX2 addition) exhibits a very high instruction retire rate since AVX2 kernels are well-optimized for this architecture. LongAddAVX2 has a high `BE:CORE` bottleneck due to its high number of ALU operations. KFunction (single kernel evaluation for SVM training) from SVM is `BE:CORE` bound. KFunction implements different types of kernel computations such as *dot product* for linear, *pow* for polynomial, *Radial Basis Function (RBF)*, and *sigmoid*. Due to these intensive computations, there has mainly been a `BE:CORE` bottleneck. It can be observed that the kernels in our ML applications are mostly BE-bound, specifically 48% of notable ML kernels have high `BE:MEM` and `BE:CORE`, bottlenecks. The reason is that many ML kernels need to utilize the ALU/vectorization units for their computations and load/store data to the cache and memory, which creates back-end issues. Also, half of our ML kernels (including those from the ML benchmarks using the AXV2, SSE2, FMA, and Scikit-learn libraries) have high `RET` because these libraries are well-optimized.

3.5.3 Deep Learning (DL) Benchmarks

Figure 3.5 shows the notable kernels from our DL benchmarks. The "notable kernels" are directly related to the main compute performed by the layers (matrix multiplication, pooling, ReLU) or data processing (data load, data rearrangement). Unlike the SPEC CPU/ML benchmarks, the notable functions are relatively similar across DL benchmarks. As shown in Figure 3.5, the notable kernels from AlexNet are `AVX2 SGEMM`, `KMP barrier`, `MKL VML pow`, `Im2col/Col2im`, `Pooling fwd/bwd` and `ReLU fwd/bwd`, which are similar across benchmarks.

DL applications have different model designs in terms of the layer deployment, activation functions, and connections of the layers to one another; however, the underlying kernels that they use are similar, which makes the DL applications experience similar behavior and bottlenecks compared to the other types of benchmarks tested in this study. Another difference is that many hotspot kernels are `AVX2 SGEMM` type, a well-optimized kernel that results in high `RET`. However, except for the `RET` part of `MKLsgemmAVX2`, it mostly experiences `BE:CORE` bottleneck because of the intensive VPU usage.

3.6 Hardware Metrics and Selection Methodology

3.6.1 Hardware Metrics

Multiple architecture components interact with one another during execution, and various hardware metrics can be captured dynamically via performance counters (cf. Section 3.3.1). Intel [66] specifies all hardware-related metrics that can be collected by VTune [67] for different Intel architectures. There are 300 metrics; choosing the 'right metrics' to employ in our MLAM is crucial. Below, we discuss our metric selection strategy.

3.6.2 Metric Selection Methodology

In MLAM, the hardware metrics are used as 'features.' However, there are many metrics, and it would not be a good idea to use them all because (i) some metrics are not directly related to the bottlenecks, so using them as features could decrease the accuracy of MLAM, and (ii) a large number of feature sets can decrease the performance of MLAM. So, it is important to select 'most beneficial' metrics as features. We employ two metric selection strategies – *correlation-based* and *importance-based*.

The correlation coefficient measures the degree to which a pair of variables are linearly related. With random variables X and Y, correlation coefficient $\rho_{X,Y}$ can be calculated as: $\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y}$, where μ_X and μ_Y are expected values and σ_X and σ_Y are standard deviation. $\rho_{X,Y}$ can be $-1 \leq \rho_{X,Y} \leq 1$. If $\rho_{X,Y} = 0$, X and Y are uncorrelated; if $\rho_{X,Y} = +1$, they have perfect relationship; and, if $\rho_{X,Y} = -1$, they have perfect inverse relationship.

We can calculate the correlation across 'bottlenecks' (Figure 3.6) as well as between 'bottlenecks' and 'hardware metrics' (Figure 3.7). From Figure 3.6, it can be found that `RET` has a negative correlation with all other bottlenecks and a strong negative correlation with `BE:MEM`. Most of the benchmarks experience `BE:MEM` bottlenecks, and this is why there is a strong negative correlation between `RET` and `BE:MEM`. `FE` and `SPE` are somewhat positively correlated since the `SPE` issue is mainly raised as a result of branch misprediction, and it affects the front end (regarding re-fetching/decoding/allocating). Likewise, `BE:MEM` is positively correlated with the caches and the DRAM, as they are the main components that create the `BE:MEM` bottleneck. As explained earlier in Section 3.5, a significant fraction of the `BE:MEM` boundedness is due to the DRAM, precisely due to the DRAM bandwidth (`DRAM:BW`). Thus, `BE:MEM` strongly correlates with DRAM and `DRAM:BW`.

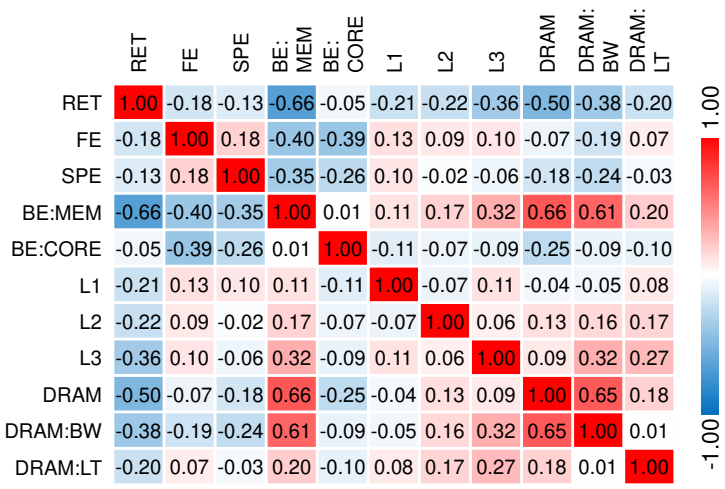


Figure 3.6: Correlation across bottlenecks, calculated as described in Section 3.6.2 (less translucent colors mean higher negative/positive correlation).

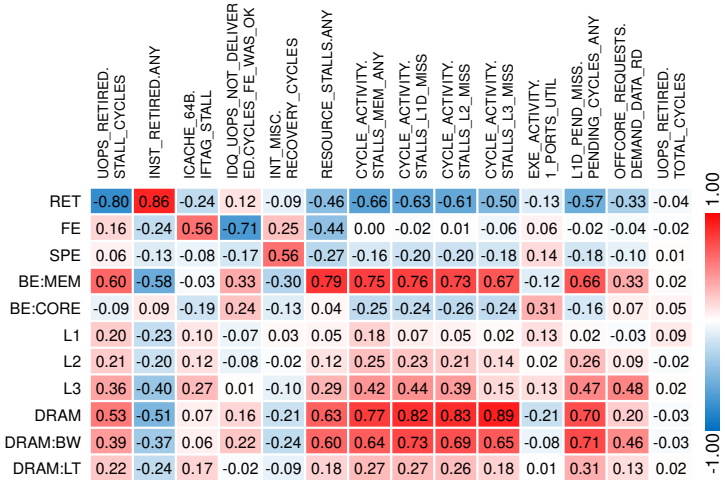


Figure 3.7: Correlation between architectural bottlenecks and hardware metrics.

3.6.2.1 Correlation-based Strategy

We also calculate the correlation coefficients between bottlenecks and hardware metrics, and the results are presented in Figure 3.7. `UOPS_RETIRED.STALL_CYCLES` has a strong negative correlation with `RET` since it captures how many execution cycles are stalled. It positively correlates with other bottleneck reasons and strongly correlates with `BE:MEM` since many kernels experience the `BE:MEM` issue. `ICACHE_64B.IFTAG_STALL` exhibits a strong correlation with `FE` and a slight positive correlation with the caches and the DRAM since it relates to the stalls during fetch due to the L1 instruction cache tag misses.

Likewise, `INT_MISC.RECOVERY_CYCLES` seems to be positively correlated to FE and SPE because of the branch misprediction recovery, and `EXE_ACTIVITY.1_PORTS_UTIL` has a positive correlation with `BE:CORE`. Other cache and memory-related metrics have a strong positive correlation with the memory bottleneck. However, metrics like `UOPS_RETIRED. TOTAL_CYCLES` are not much correlated to any bottleneck reasons, so we do not use them.

3.6.2.2 Importance-based Strategy

The importance-based approach is used for high performance and accuracy. We used *permutation feature importance* [5, 18] to identify how important a given feature is for the ML/DL model. With a fitted predictive model m and a dataset D, we can calculate a reference score s (accuracy) and the permutation importance i_j of column j can be calculated as: $i_j = s - \frac{1}{K}\sum_{k=1}^{K} s_{k,j}$, where $s_{k,j}$ is the score of model m with corrupted data $\tilde{D}_{k,j}$.

The permutation importance specifies the importance of each feature (column). We can use it to select the features that are *not* essential to train the model and remove them. How we use the importance to select the features and the importance changes over iterations are explored in Section 3.7.2 and Section 3.8.

3.7 Analysis Model

MLAM can transfer bottleneck analysis knowledge to new architectures and applications. Specifically, it could be trained using VTune bottleneck analysis data with many applications and architectures (Step 1). Whenever a new application and/or architecture is encountered, the new application could be analyzed using VTune on the new architecture, and the raw data from this analysis could be fed into the trained MLAM model for obtaining an optimal hardware configuration for that application (Step 2). As depicted in Figure 3.8, MLAM has four major components – *data preparation and preprocessing*, *metric selection*, *model training*, and *inference*. The overall flow of MLAM includes (1) gathering the data and preprocessing it; (2) selecting the features using correlation-based or importance-based methodologies (see Section 3.7.2); (3) training the ML/DL models with training data and chosen features; (4) validating the ML/DL model with the validation set; (5) if the validation accuracy improves or similar, going back to the feature selection for the next iteration if the model fitting is not finalized; and (6) inferencing with the test dataset.

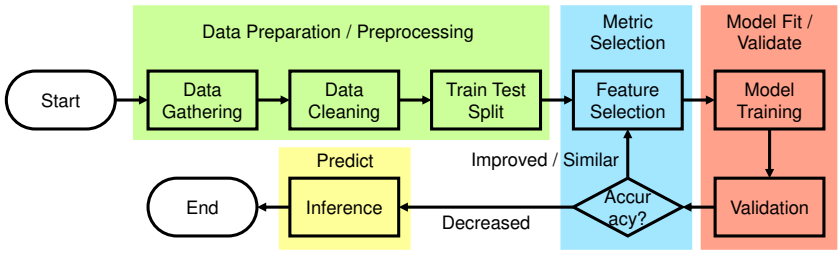


Figure 3.8: End-to-end view of MLAM.

3.7.1 Data Preparation/Preprocessing

3.7.1.1 Data Gathering and Cleaning

Numerous hardware metrics are related to architectural bottlenecks (see Section 3.6.1). Each kernel with hardware metric values can be one data point, and the metrics are 'features.' For supervised learning, we need a correct label for each data. As shown in Section 3.5, kernels can have a fraction of their bottlenecks in `RET`, `FE`, `SPE`, `BE:MEM`, and `BE:CORE`, which can directly act as the response value.

We used VTune [67] to collect the data points. It periodically checks the status of the architecture and kernels and collects statistics. If the kernel execution is too short, the data does not correctly reflect the kernel's state. So, careful 'data cleaning' is needed for better accuracy. We dropped kernels that contained less accurate information and retained primary hotspots. We dropped the duplicated data points and the ones with zero values that exceeded the threshold.

3.7.1.2 Normalization

For MLAM, normalization is crucial to improve accuracy because the metric values are mostly cycles or occurrences. Therefore, the longer the kernel is, the larger the metric value is. Thus, to eliminate the effect of the length of the kernel, we *normalize* the value of all metrics by each kernel's number of cycles. Another normalization we apply is column-level standardization. Each metric can have a different range, and as a result, combining all of them for a feature can cause a problem by giving a wrong impression to the model, like large values being important. This normalization aims to change the column values into a 'common scale' without distorting the range differences or losing any information. For this, we apply z-score ($z = (x - \mu)/\sigma$) transformation. We obtain the values of μ and σ from the training data and apply the same z-score transformation to test/validation data to prevent corruption.

3.7.1.3 K-Fold

K-Fold cross-validation [184] is used to improve the precision of a model's accuracy, especially with small-sized datasets. It (1) divides the dataset into k parts; (2) uses $k-1$ parts as training data and 1 for validation; (3) repeats step (2) k times while rotating the validation set; and (4) determines the expected accuracy based on the results across the iterations. We used the K-Fold method with $k = 10$ for a more accurate permutation importance value, which increased the final accuracy of MLAM.

3.7.2 Feature Selection

We employ two feature selection strategies: *correlation-based* and *importance-based* (see Section 3.6.2). Apart from the differences in their mathematical formulation, these two strategies also have a procedural dissimilarity. The correlation can be known *before* running the ML/DL model, whereas the importance can only be known *after* running the model. We calculate the correlation between the hardware metrics and the bottlenecks, which correspond to the features and labels of our dataset. However, the importance, by definition, needs the fitted model as its input, so we can only know the importance after we train the model. Considering these, there are three sub-parts in our metric selection process: *feature list initialization*, *feature list update*, and *termination*.

3.7.2.1 Feature List Initialization

There are around three hundred hardware metrics. Not all of these metrics would be helpful in *predicting* the bottlenecks for a given kernel. Including all metrics as a feature list could decrease the performance of MLAM (due to the repetitive feature list update iterations). Choosing a reasonable 'initial feature list' is essential for accuracy *and* performance. For this, we use the correlation-based methodology. We choose the subset of Φ (whole feature set) as the initial feature set (Φ_{init}), which can be obtained from the following equation:

$$\Phi_{init} = \{f \in \Phi : \exists y \in Y \ s.t. \ |\rho_{f,y}| > \gamma\}, \tag{3.1}$$

where Y is the label set, $\rho_{f,y}$ is a correlation between a feature (f) and a label (y), and γ is the correlation threshold. The expression indicates that if the absolute value of the correlation between a feature and any of the labels is larger than the threshold, then the feature becomes an element of Φ_{init}. The next challenge is to pick reasonable γ, whose effect will be studied later in Section 3.8.

3.7.2.2 Feature List Update

After we have Φ_{init}, the model is trained using the train data with the initial feature list. Using the fitted model, we can calculate i_f (importance of each feature). As discussed in Section 3.6.2.2, 'importance' indicates *how vital a given feature is to the model*, and with it, we can remove the 'not-so-important' metrics from our feature list. However, it is possible that some features, classified as unimportant with a low cross-validation score, could be essential for a good model. That is why we do feature selection in an *iterative fashion* instead of fitting the model once and choosing features based on importance right from that. We gradually remove the features with regular ratio (α) from the feature list and use the updated feature list in the next iteration ($rank(i_f)$ in descending order):

$$\Phi_{iter+1} = \Phi_{iter} \setminus \Phi_{exc},\ where,\ \Phi_{exc} = \{f \in \Phi_{iter} : rank(i_f) > |\Phi_{iter}| \times (1-\alpha)\}. \quad (3.2)$$

3.7.2.3 Termination

The termination stage decides whether to continue the feature list update or not. To determine that, we use the validation score, R^2, which can be calculated as $R^2 = 1 - \frac{SSE}{SST}$. R^2 can have a value less than or equal to 1, and a higher R^2 value indicates a better model. After the cross-validation, we can identify whether the model has improved ($R^2_{iter} - R^2_{iter-1} > \epsilon$), remained similar ($|R^2_{iter} - R^2_{iter-1}| \leq \epsilon$), or worsened ($R^2_{iter} - R^2_{iter-1} < -\epsilon$). There could be room for further improvement if it improved, so we continue. Likewise, if R^2 remained similar, we continue the feature selection because (1) the model might be improved in the future or (2) a shorter feature list can increase performance. When R^2 starts decreasing, it is likely that we removed valuable metrics, so we stop and use the previously trained model for the inference.

3.7.3 ML Model Fitting/Validation

MLAM employs two different ML algorithms: *Random Forest Regressor (RFR)* and *Multi-Layer Perceptron (MLP) Regressor*. We selected regression-based algorithms since our target values (architectural bottlenecks) are fractions/percentages, as shown in Section 3.5. When we consider the nature of the bottlenecks, kernels are not 100% core or memory bound. Instead, they experience a mixture of bottlenecks.

In MLAM, it is possible to use any supervised learning algorithm. Among the choices, RFR and MLP are good representative models for exploring the effectiveness of MLAM. RFR is robust in noisy data and relatively simple to fine-tune, which suits our purposes well, considering that our dataset is somewhat noisy and that a long fine-tuning process

would degrade the performance of MLAM. MLP is a simple but very effective DL model commonly used in many domains; thus, by using MLP, we can show the effectiveness of DL models in bottleneck analysis.

To fine-tune MLAM, we used the *successive halving hyperparameter tuning method* [71, 102] that searches the 'hyperparameter space' for the best hyperparameter combination. For RFR, the hyperparameters are the number of estimators, the number of features for the best split, the minimum number of data to split the node, and the minimum number of data a leaf node should have. For MLP, the hyperparameters are the size of the hidden layer, the L2 penalty (regularization) parameter, and the initial learning rate. We use separate validation scores to decide whether to repeat the metric selection iteration. This score becomes an input to the metric selection stage as explained in Section 3.7.2.

3.7.4 Inference

After MLAM terminates iteration, it proceeds to the inference with the test data. We calculate the R^2 score to evaluate its effectiveness. Inference can be executed with the previously trained, saved model, which helps avoid re-training the model whenever we predict the bottleneck.

3.7.5 Multi-Architecture Support

In our evaluations, we train and perform inference using the proposed MLAM using the Intel Skylake architecture, as mentioned in Section 3.4. However, we also train our MLAM using the Intel Broadwell, Icelake, and Cascadelake architectures individually to show that our model helps avoid performing cumbersome architectural profiling data deduction to perform bottleneck analysis in *different architectures* and for *many applications*. One will not need to profile many applications to train the model since trained models could be reused for each architecture.

We also propose a *multi-architecture-trained model*. We train our model with multiple architectures and perform inference with one architecture. Multi-architecture training is aimed at the case where the end-user (hardware designer) does not have enough historical hardware profiling data for the target (inference) architecture, so the large amount of data coming from other architectures and a small amount of data coming from the target architecture will be used to predict bottlenecks for a new workload running on the target architecture. Finally, MLAM aims to reduce the time required for *comprehending* the architectural profiling data, which slows down any progress in research and development

for designing better-suited hardware configurations for different application domains. Once MLAM is trained, it can perform verification multiple times to reduce the required effort to reduce architectural profiling raw data deduction time.

3.8 Experimental Results

In this section, we quantify (1) the effect of correlation-based feature selection, (2) performance and accuracy with importance-based feature selection, (3) the importance of the metrics to predict bottlenecks, and (4) fine-tuning.

3.8.1 Correlation-Based Feature Selection

Figure 3.9 plots the changes in performance (speedup) and accuracy when using different correlation threshold (γ) values in the feature list initialization. The bar graph shows the performance variations in training and inference concerning the cases with $\gamma = 0$ to $\gamma = 0.85$. The line graph shows the model score, R^2. The x-axis indicates γ and the size of Φ_{init} derived from Equation 3.1.

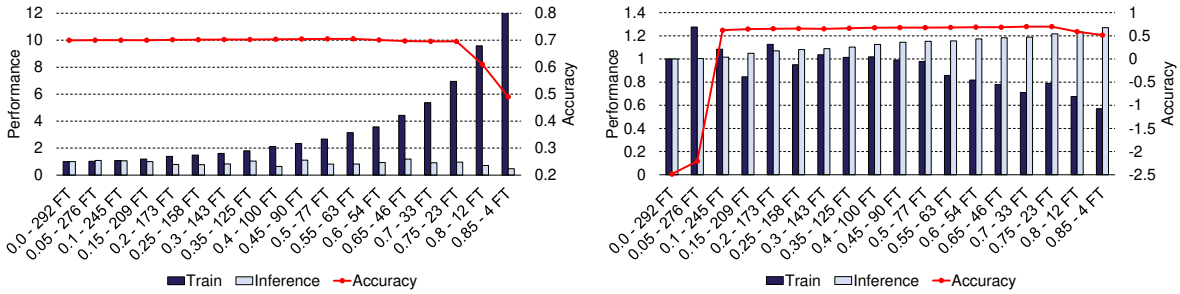


Figure 3.9: Performance (bars) and accuracy (line) of RFR (left) and MLP (right), with correlation-based metric selection.

Following observations can be found: (1) The RFR model works well with $\gamma = 0$, whereas the MLP model has inferior accuracy with low γ values; (2) In the range of $0.1 \leq \gamma \leq 0.75$, the accuracy of RFR is stable (around 0.70); however, MLP achieves a slight accuracy improvement (0.62 to 0.70); (3) In both RFR and MLP, the accuracy drops when employing a very high γ value (i.e., $\gamma > 0.75$); (4) When the value of γ increases, the RFR training performance improves dramatically, whereas that of MLP decreases; and finally, (5) As γ is increased, the execution time of RFR inference exhibits a stable behavior, whereas the MLP inference execution time decreases (gradual performance improvement till 27.06%).

Regarding observations (1) and (2), RFR works well with $\gamma \leq 0.75$ since RFR randomly selects the features to create the forest, and as a result, if a feature is not decided to be 'good' to be added, RFR does *not* select that feature. Therefore, as long as the metrics are in the feature list, RFR is expected to result in robust accuracy. However, MLP uses *all* the features to train its hidden layers; it is more sensitive to the features than RFR. Therefore, if it gets not-very-related data, the accuracy can drop drastically, as in the case where $\gamma = 0$. This behavior is also why MLP gradually improves accuracy as we select good features.

Regarding observation (3), if we set γ too high, some essential features are excluded, so the accuracy drops in both. Regarding observation (4), the performance of the RFR training improves as the number of feature sizes decreases since the RFR can quickly build a forest with a smaller number of feature sets. However, for MLP, we use early stopping when MLP converges; so, even though the execution time of one MLP iteration gets reduced with fewer features, the total iteration count increases, leading to performance degradation. Finally, for (5), the execution time of the RFR inference is short, and it depends on the depth of the tree. Thus, the performance of RFR does not change much. In contrast, for MLP, with a smaller feature list, the number of computations needed for inference decreases, which makes MLP inference performance better.

3.8.2 Importance-Based Feature Selection

Figure 3.10 plots the performance and accuracy changes per iteration with feature list update. We set $\alpha = 0.1$ and $\gamma = 0.5$ so we do not prematurely exclude any critical feature. At each iteration, we only choose the features that satisfy Equation 3.2. The training and inference performance is similar to Figure 3.9, and the validation performance is comparable to that of inference. The performance of importance calculation improves *exponentially* since we use permutation importance. The validation accuracy drops beyond iter11 for RFR and iter12 for MLP, primarily due to removing essential features. However, RFR shows steady validation and inference accuracy (around 0.7), whereas the validation and inference accuracy of MLP improves, which was also captured during the correlation-based approach. One difference is that MLP shows a higher peak validation (0.71) and inference accuracy (0.72) compared to the correlation-based one (0.70), mainly because of the better feature selection strategy it employs. Based on this experimental data, we have selected a feature list in iter11 for RFR and iter10 for MLP.

Figure 3.11 gives the permutation importance of the hardware metrics over iterations in the feature update. The 25 metrics in the graph are from the feature list created

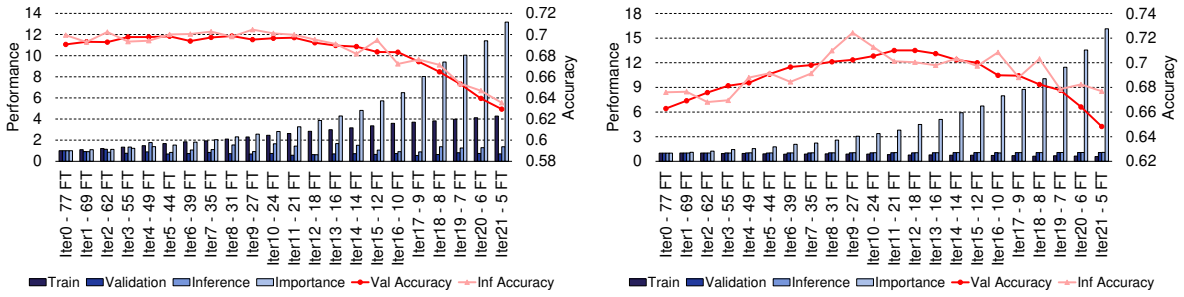


Figure 3.10: Performance (bars) and accuracy (lines) of (a) RFR and (b) MLP with importance-based metric selection.

during iter11 in RFR and iter10 in MLP (in total 32). The lines show the importance changes from iter0 to iter21, and if the feature is excluded, the line ends there (the longer the line is, the longer the feature lasts). RFR and MLP utilize different metrics to train the model. In our selected feature list, 40.6% of the features are used in both RFR and MLP, 25.0% of them are used only in RFR, and 34.4% only in MLP. Additionally, 21.9% of them are related to the number of instructions, μOps, and overall stalls, 25.0% are related to core cycles and stalls, 28.1% are for front-end, 6.3% are from misprediction, 15.6% are from cache and memory, and finally 3.1% are for retire slot. We see that the metrics capturing the number of inst/μOps, stall-related metrics, core-related ones, or retirement slot usage are the 'most valuable' hardware metrics for MLAM.

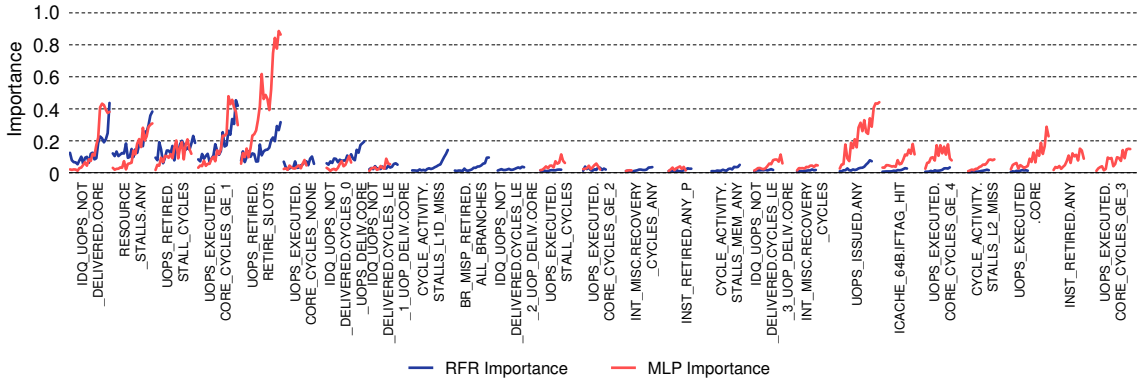


Figure 3.11: Permutation importance changes over feature selection iterations on selected metrics.

3.8.3 Fine-Tuning

Figures 3.12 and 3.13 plot the accuracy variation during the fine-tuning. Each line represents one possible candidate of the hyperparameter combination. Also, the x-axis

captures the iteration in the halving process with (amount of resources, number of candidates). Initially, the accuracy is low since a small fraction of the dataset is used. We can obtain the best hyperparameter combination by adding more resources and eliminating unpromising candidates. The candidates that reach the last iteration with the best accuracy are depicted in red with an accuracy of 0.70 (RFR) and 0.72, 0.71 (MLP). Variances in accuracy between the candidates of RFR are minimal, whereas the variances in accuracy between the candidates of MLP are significant. Because of the characteristics of RFR (robust and easy to fine-tune), there is not much variance between the candidates, which makes any hyperparameter combination perform reasonably well. At the same time, the effect of fine-tuning (accuracy improvement) is relatively small. MLP shows variations between the candidates, making it hard to fine-tune, but it improves accuracy (0.71 → 0.72).

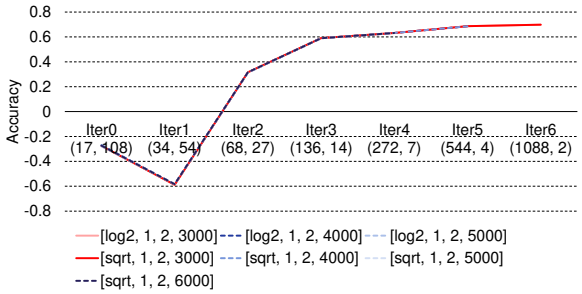


Figure 3.12: Fine tuning for RFR model with [max_features, min_leaf, min_split, n_estimators] hyperparameters.

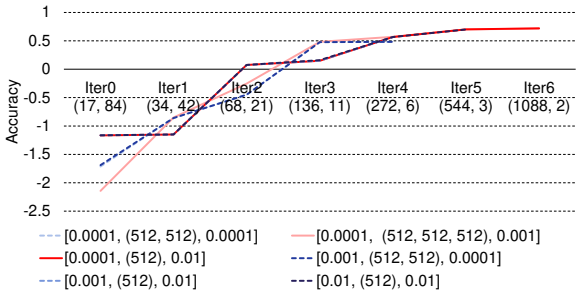


Figure 3.13: Fine tuning for MLP model with [alpha, hidden_layer_sizes, learning_rate_init] hyperparameters.

3.8.4 Results for Different Architectures

To perform a multi-architecture evaluation of MLAM, we collected comprehensive and detailed architectural profiling data for the Broadwell, Icelake, Skylake, and Cascadelake architectures using their hardware counters, *for only ML workloads*, due to time constraints. We used *correlation-based feature selection* with the *Random Forest Regression model* because this set of experiments is more reliable with fewer data. We also perform multi-architecture training to evaluate our model regarding bottleneck ordering and magnitude prediction in multi-architecture settings. For this, MLAM was trained using three of the four architectures along with a portion of the remaining one and validated using the remaining portion of the remaining architecture.

3.8.4.1 Ordering and Value-Based Similarity Metrics

MLAM helps deduce bottlenecks robustly from raw architectural profiling data to eventually create suitable architectures for different applications. Therefore, instead of obtaining the architectural bottleneck prediction accuracy, obtaining the bottleneck ordering accuracy and some influence of the absolute bottleneck percentage values is crucial to optimizing performance-limiting hardware components robustly. To ensure this, we propose our own ordering and magnitude similarity metric. Our similarity metric was initially based solely on the 'ordering' of the bottlenecks according to predicted bottleneck values, as shown in Definition 1. However, we improved this metric by adding the influence of the absolute percentage values of bottlenecks as shown in Definition 2.

Definition 1 (Ordering-Based Similarity) Let n_j be the number of intersections between the first j elements of the true ordering of bottlenecks, t, and predicted ordering of bottlenecks, p, then

$$sim\_ord\_based(t,p) = \frac{1}{m}\sum_{j=1}^{5}(6-j)\left(\frac{n_j}{j}\right),$$

where m is a constant to normalize $sim\_ord\_based(t,p)$ to lie between 0 and 1, $\frac{n_j}{j}$ is the ratio of correctly guessed number of bottlenecks in the first j most important (highest percentage) ones, $(6-j)$ is weighting to increase the importance of the most important bottlenecks.

Definition 2 (Ordering- and Value-Based Similarity) Let $t = (t_1, \ldots, t_5)$ and $p = (p_1, \ldots, p_5)$ be the percentage values of `RET`, `FE`, `SPE`, `BE:MEM`, and `BE:CORE` for true and predicted bottlenecks, respectively. Assume $\tilde{t} = (\tilde{t_1}, \ldots, \tilde{t_5})$ is the descending ordered version of t, whereas $\tilde{p}$ is the ordered version of p where the order respects the mapping from t to $\tilde{t}$. That is, $\tilde{t_1}$ is the highest bottleneck value, while $\tilde{p_1}$ is the prediction for this value. Observe that $\tilde{p_1}$ is not necessarily the largest among $p_1, \ldots, p_5$. Similarity between t and p is

$$sim\_val\_based(t,p) = 1 - \frac{1}{m}\sqrt{\sum_{i=1}^{5}(6-i)^2(\tilde{t_i}-\tilde{p_i})^2},$$

where m is a constant to normalize $sim\_val\_based(t,p)$ to lie between 0 and 1, and $(6-i)$ are added weights to increase the importance of the most important bottlenecks.

3.8.4.2 Single-Architecture Train Results

We present the single architecture model training and bottleneck prediction results using our similarity metrics. As seen in Figure 3.14, the *Ordering and Value-Based*

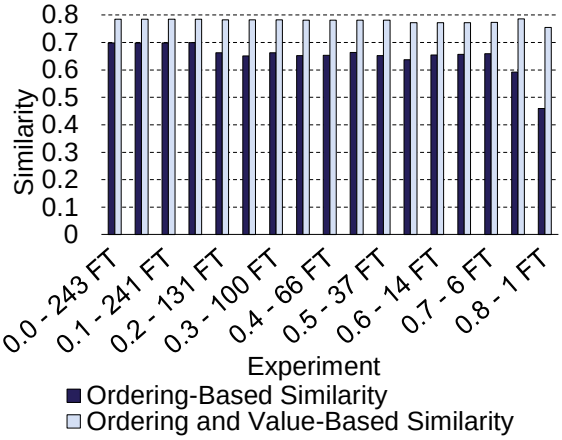


Figure 3.14: Similarity results for the Broadwell architecture experiments.

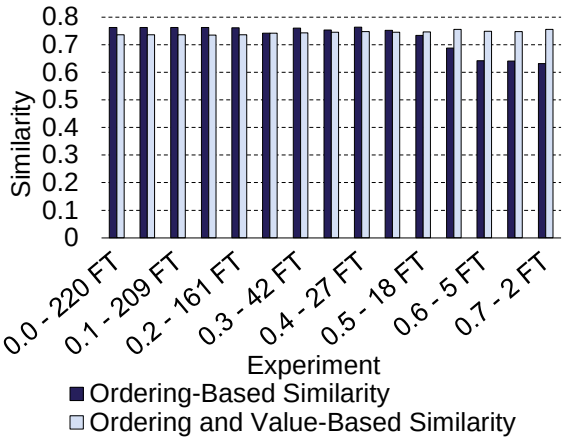


Figure 3.15: Similarity results for the Cascade Lake architecture experiments.

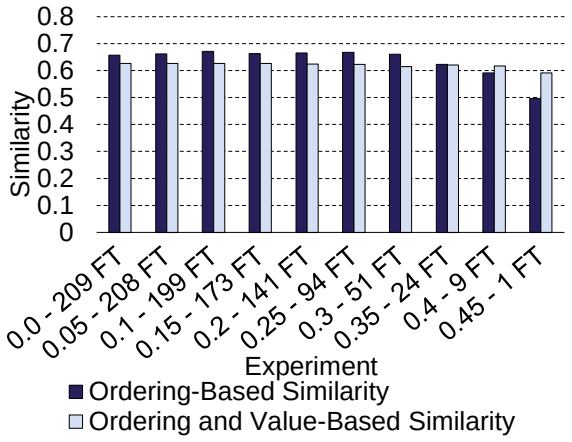


Figure 3.16: Similarity results for the Ice Lake architecture experiments.

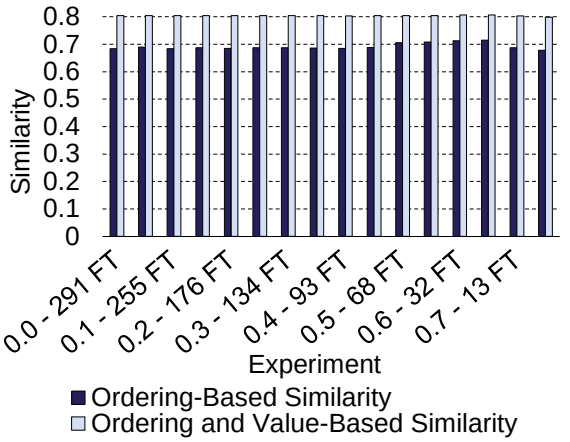


Figure 3.17: Similarity results for the Skylake architecture experiments.

Similarity metric was consistently higher than the *Ordering-Based Similarity* in all correlation threshold and feature size experiments on the Broadwell architecture. This result indicates that the ordering and the percentage values of bottlenecks were also predicted accurately. This result was the same for most of the correlation threshold and feature size experiments of the Cascade Lake architecture (Figure 3.15) and all experiments of the Skylake architecture (Figure 3.17). In some Ice Lake experiments, the Ordering-Based Similarity metric had slightly higher results (Figure 3.16). On average, the difference between the two metrics was only 2%. This result shows that the orderings were accurate; however, there was a slight difference in bottleneck percentage value predictions due to having fewer data than needed. In the case of a larger dataset, the

Ordering and Value-Based similarity metric will give a higher similarity result due to the model being trained better. On average, the Ordering-Based and Ordering and Value-Based Similarity for the Broadwell architecture were 65% and 77%, respectively; 72% and 74% for Cascade Lake; 64% and 62% for Ice Lake; and 69% and 80% for Skylake. Overall, when the bottleneck value information was used to calculate the similarity between the actual and predicted bottlenecks, this similarity was 6%

3.8.4.3 Multi-Architecture Train Results

For multi-architecture model training evaluation, we performed a different set of experiments. The performance counters were different across different architectures. So, we identified the same ones with different names and performed a mapping between them. Moreover, if an architecture did not have a counter that another had, we filled in missing values using the mean or median strategy. Specifically, we varied the merging strategy of the different architecture datasets (union, intersection), the tested architectures (Broadwell, Cascade Lake, Ice Lake, and Skylake), and the missing hardware counter data-filling method (mean, median). As seen in Figure 3.18, the Ordering and Value-Based Similarity results were consistently higher for all of the experiments conducted (by 4% on average). The similarity values were, on average, higher than 70%

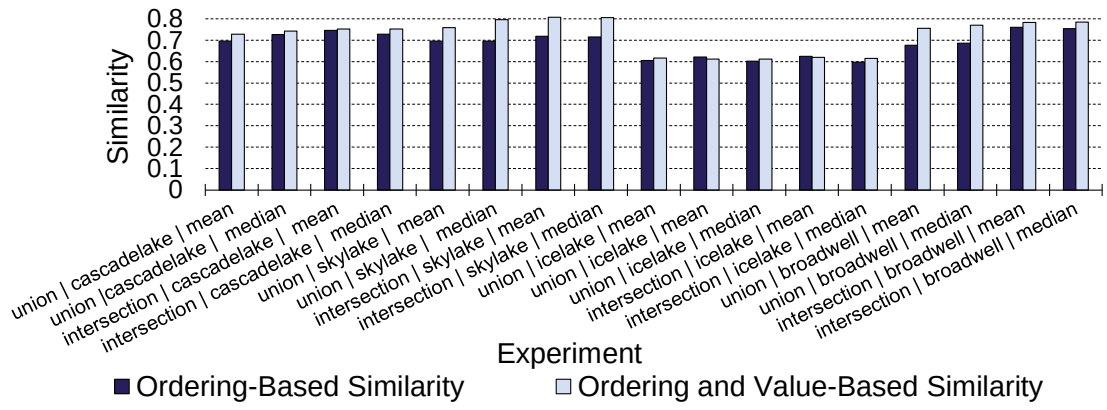


Figure 3.18: Similarities for multi-architecture experiments.

In both sets of experiments, Ordering and Value-Based Similarity results were higher (72.4%), indicating that both ordering *and* bottleneck values that MLAM outputs are accurate to create an application- or domain-specific architecture, showing MLAM will aid in robustly reconfiguring the hardware according to the predicted bottlenecks, creating a custom architecture, for the target workloads.

3.8.5 Summary of the Results

RFR vs MLP: RFR shows a more stable accuracy than MLP, and MLP gives a higher peak accuracy than RFR.
Single Architecture Train Similarity Results: The Ordering and Value-Based Similarity for the bottleneck ordering **and** value prediction was higher than the ordering-based similarity for four architectures and the multi-architecture training, indicating that not only the bottleneck ordering but also the bottleneck percentages were predicted accurately.
Correlation vs Importance Based Feature Selection: Correlation- and importance-based feature selection strategies exhibit similar behavior. However, eliminating features solely based on the correlation coefficient is faster than importance-based elimination but risks excluding "important" features (impacting accuracy).
Important Hardware Metrics: RFR and MLP share some metrics (40.6%); however, they also independently use different metrics to predict the bottlenecks (RFR: 25.0%, MLP: 34.4%). Generally, the metrics for the number of inst/μOps, stall-related, core-related, and retirement slot usage are the most *important* metrics for MLAM.

3.9 Discussion of Related Works

General Performance Modeling: Williams et al. [204] proposed a roofline model to improve the performance of parallel floating-point applications. Stengel et al. [180] employed a cache-memory performance model to quantify the bottlenecks in stencil computations, and Hoefler et al. [59] predicted the performance of parallel applications. Unlike these works, we focus on using ML/DL to perform architectural bottleneck analysis.
ML-Based Modeling: Recent works proposed ML models to predict the behavior of applications running on different hardware: Yoo et al. [209] proposed to use ML to identify the hotspot of functions in an application. Yoo et al. [210] suggested a job status prediction strategy for scientific clusters. Jaggard et al. [70] implemented a framework to monitor the network performance for ML applications. Compared to such works, MLAM employs ML/DL algorithms and shows the relationship between the hardware metrics and architectural bottlenecks. To our knowledge, using ML/DL models to explain the relationship between metrics and bottlenecks has *not* been explored before.

3.10 Chapter Summary

In this chapter, we proposed MLAM, a machine learning-based architectural analysis tool. MLAM allows the transferring of the knowledge obtained from architectural analysis tools to different applications and architectures to tune hardware. We benchmark programs from different domains and identify their architectural bottlenecks. Further, we identify the relationships between the hardware metrics and the bottlenecks from the correlations.

Chapter 4
On-the-Fly Job Scheduling to Instantaneously Suitable GPUs

4.1 Introduction

As HPC applications become more popular and compute-heavy, the use of heterogeneous compute elements increased significantly. Most of the HPC heterogeneous compute platforms contain multiple GPUs. Offering an intelligent job scheduling algorithm for GPU-supported jobs within multi-GPU hosts could improve the efficiency of resource allocation on GPU infrastructures, increasing the performance of high-performance computing (HPC) applications.

One of the most popular frameworks in the bioinformatics domain that hosts many HPC applications, Galaxy, is a web-based open-source framework widely used by thousands of researchers [124] for a variety of compute-intensive applications, including computational chemistry [82], genome assembly, epigenetics, metagenomics, ML, and drug discovery [156]. Galaxy can be installed on various computing platforms, such as local clusters, public datacenters, and national lab supercomputers [123], and is also available as a worldwide network of managed free services (known as *usegalaxy.**). As a result of its accessibility, Galaxy has been cited over 10,000 times in the last decade [125]. Galaxy allows users to access tools, manage workflows, reproduce, store, and share experimental results with the community with these diverse deployment options.

As evidenced by prior research [145], many of the tools used in Galaxy have parallelization opportunities, potentially enabling substantial performance improvements when executed on hardware accelerators. An example is PyPaSWAS [203], a sequence alignment application showing a 33× speedup with GPU than CPU. Within the broad spectrum of hardware accelerators, including GPUs, Field-Programmable-Gate-Arrays (FPGAs),

and Application-Specific Integrated Circuits (ASICs), GPUs have been dominating the landscape due to their multi-faceted nature of supporting modern-day applications [145]. GPUs have become an essential part of modern computing systems, with their use reaching far beyond their initial target domain (computer graphics) to many parallel application domains such as bioinformatics, nuclear physics, deep learning, and others [25, 44, 62, 128]. With the rapid increase in programmability and compute and storage capabilities of GPUs, there has been ongoing development of a growing set of applications and problems mapped to GPUs. For instance, Argonne National Laboratory's researchers have accelerated a COVID-19 vaccine study that simulates an essential part of a protein spike on coronavirus made up of 1.5 million atoms. Using the latest V100 GPUs, they achieved a 5× speedup compared to CPU-only execution [56].

With the proliferation of parallel bioinformatics applications/tools, deployment platforms like HPC supercomputers and public data centers have begun to include GPUs as a part of their mainstream hardware infrastructure. For instance, the Brookhaven National Laboratory has expanded its supercomputers to include a 200-node GPU cluster [117]. Despite GPUs being available in today's hardware infrastructure, the current Galaxy framework does not support GPUs. This apparent deficiency in Galaxy motivates the central premise of our work: *can we make the Galaxy framework "GPU-aware" such that GPU-enabled tools can leverage the flexibility offered by executing in Galaxy for enhancing performance?*

However, it is non-trivial to integrate GPUs into the current Galaxy framework without affecting the user experience (i.e., users should be able to retain their original tool deployment method, while the tools should be able to leverage GPUs when applicable). To address this problem, we present GYAN, an enhanced Galaxy framework with support for executing GPU-enabled tools. We achieve this through minimal code enhancements and user-agnostic modifications to the Galaxy framework, and we further test our improvements on an in-house Galaxy deployment.

We make the following key contributions:

1. We make Galaxy GPU-aware such that the tools with GPU capability can seamlessly execute in Galaxy alongside CPU-enabled tools. We support bare-metal (locally running) and containerized tools that can be executed in Galaxy.

2. We design an intelligent GPU-aware orchestration policy such that a given tool can be executed on a CPU or a GPU based on availability. Furthermore, we provide multi-GPU support to spread highly compute-intensive tools across multiple GPUs.

The GPU selection is designed based on the availability and utilization of all GPUs in a cluster.

3. We evaluate the effectiveness of the proposed GPU support in Galaxy using two widely used tools: *Racon* and *Bonito*. Racon is a genome consensus [196] tool, and Bonito performs base calling [172].

4. The results from our experiments indicate that the GPU versions of these two tools show ∼2× (for Racon) and ∼50× (for Bonito) speedups over their original CPU-based counterparts.

The remainder of this chapter is structured as follows. Section 4.2 presents the necessary background information on Galaxy, bioinformatics tools, GPUs, and containerization. In Section 4.3, we then explain the motivation behind this work and our challenges. The details of implementing the enhancements we made to the Galaxy framework to allow GPU-aware job mapping and orchestration are given in Section 4.4. This section also discusses our multi-GPU support. The experimental results are presented and discussed in Section 4.5, followed by the conclusions.

4.2 Background and Related Work

We present background on Galaxy, containerization support, parallelism, and GPUs.

4.2.1 Galaxy Software Framework

Galaxy is an open-source web-based framework maintained by a large, worldwide community. Galaxy enables thousands of researchers without informatics expertise to perform computational analyses [123]. Galaxy framework consists of two main components. The first is *The Galaxy Software Framework*, a web-based computational analysis application. The framework interacts with the underlying computational infrastructure hidden from the user. This infrastructure can be a conventional cluster, cloud, or a hybrid system that combines the two. The second component, *usegalaxy.**, is a set of hosted, free servers around the works that allow thousands of users to use various tools. These servers are typically hosted on national cyberinfrastructure (e.g., in the US, at Texas Advanced Computing Center (TACC) as part of the CyVerse project [123], and in Australia, on the NeCTAR academic cloud).

The hosted Galaxy instances offer a set of popular, commonly used tools. *Tools* are the applications run on Galaxy instances. These tools are used by an end-user, installed as a "Galaxy Admin", and developed by a tool developer. When a user wants to execute a tool, it is submitted as a "Galaxy Job". A single job can be a single tool instance or a workflow consisting of a sequence of multiple tools. Galaxy tools have XML files called "tool configuration files" or "wrapper files", automatically rendered into the tool's web user interface. The wrapper files create a bridge between the tools and Galaxy to inform Galaxy on how to execute the tool, what options to pass as parameters, and what output file(s) will be generated [126].

4.2.2 Containerization Support in Galaxy

Galaxy can also use containerized tools by launching jobs as "containers." A container is a "standard unit of software that packages up code and its dependencies" [121]. Containerization allows seamless, reliable execution from one platform to another and makes end users' jobs easier by managing all the dependencies. Docker containers [121] are instantiated, at runtime, from Docker Container Images, which are lightweight and standalone packages that include the implementation, tools, libraries, and other settings necessary for that application to be executed independent of the operating system [121]. Singularity is a tool that works almost the same as Docker; however, it has different permission configurations, which allow it to be executed on HPC clusters easily [132]. Galaxy currently supports both Docker and Singularity-containerized tools. Galaxy also supports "Biocontainers". Biocontainers [150] is an open-source project that helps manage bioinformatics packages for applications and allows us to deploy them as containers. Biocontainers include Docker containers built from Dockerfile recipes and Conda-based containers that first develop a Conda package and then build a Docker container from the package [150].

4.2.3 Parallelism and GPUs

GPUs are designed to be powerful engines for computationally demanding applications. They deliver an excellent performance for many types of parallel computations. A GPU is a highly parallel programmable processor with extensive arithmetic capabilities and memory bandwidth. Therefore, it is not only meant for graphics purposes. So, it yields high-performance advantages for many parallel applications over its CPU counterpart. Further, GPU architectures have developed substantially over the years, with parallelism

and throughput becoming increasingly important than latency in many application domains. Especially the recent NVIDIA GPUs can achieve thousands of GFLOPS (giga-floating point operations per second) [145].

In an NVIDIA GPU architecture, an application code is parallelized using the CUDA parallel programming model [115], which maps threads, blocks, grids, and warps to the GPU architecture. In this model, GPUs are referred to as "devices", and CPU is referred to as "host". The functions that run in GPUs are called "device kernels". In a device kernel, the threadIdx gives the ID of the current thread, blockIdx gives the ID of the current block, blockDim gives the size of each dimension of the current block, and lastly, gridDim gives the size of each dimension of the current grid [115]. GPUs can run many device kernels in parallel, and each device kernel accesses a grid to access threads and blocks. The number of grids, blocks per grid, and threads per block depends on the GPU's compute capability. Threads are organized in 1, 2, or 3-dimensional blocks, and similarly, blocks are organized in 1, 2, or 3-dimensional grids [115]. Within a block of threads, the threads are executed in groups of 32, named "warps", and all threads in a warp execute the same thing [45]. Each thread in a warp accesses a shared memory, which can introduce bank conflicts.

NVIDIA GPUs contain Streaming Multiprocessors (SMs), and each SM contains Streaming Processors (SPs) or CUDA cores. SMs execute the device kernels in block(s) (therefore several warps) one after another. In the current GPU architectures, each SM has several warp schedulers [129], showing these GPUs' dynamic scheduling nature, allowing scalability. So, a higher number of blocks used in a device kernel allows better scaling across any GPU architecture.

To evaluate GYAN, we used a machine with two Tesla K80 GPUs. The Tesla K80 GPU has two Tesla GK210 GPUs, as seen in Figure 4.1. Both GPUs have 2,496 processor cores with a core clock of 560 MHz to 875 MHz; the memory bandwidth is 480 GB/sec, and the total board memory is 24 GB [130]. In this GPU, the number of threads per warp is 32; the maximum number of threads per block is 2048, and the maximum number of warps per SM is 64. There are 15 SMs, each containing four warp schedulers, allowing four warps to be executed simultaneously. The SMs in this GPU have improved performance for double-precision workloads [129].

4.2.4 Related Work

We briefly describe the infrastructure and software enhancements to Galaxy over the years. On the infrastructure front, cloud computing can offer on-demand access to elastic

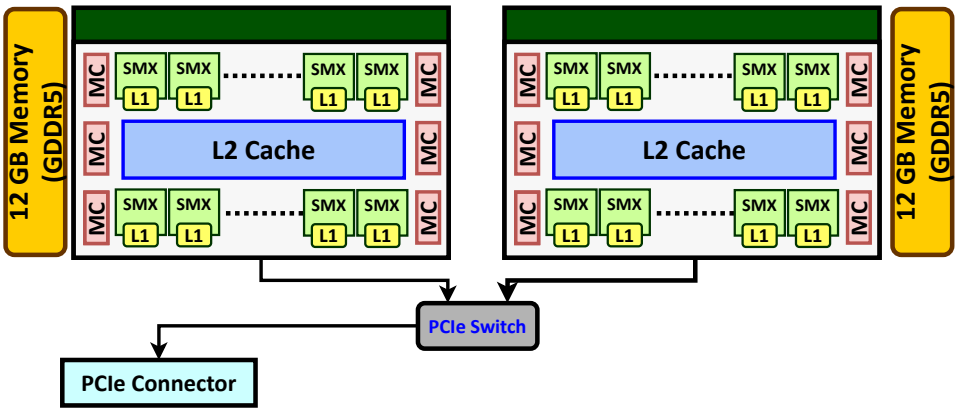


Figure 4.1: Tesla K80 GPU architecture.

computational infrastructure; however, it is not available for biologists to use "as is." "Galaxy CloudMan" was developed [2] to allow researchers to manage an arbitrarily sized compute cluster on Amazon EC2. This system does not require informatics knowledge, allows the creation of a configured compute cluster in less than five minutes, and makes the entire biological tools suite available for immediate usage.

Besides the cloud computing support, there have been several other application-level enhancements for the Galaxy Framework. One is related to expanding Galaxy's reference data [199]. For many bioinformatics analyses, properly managing reference datasets is an important task. Refgenie [199] is a reference management system that enables this task, and it is integrated into the Galaxy platform with a graphical user interface. Similarly, Galaxy was recently enhanced with another platform related to long-read sequencing, which has become popular and allows sequencing long contigs at low cost and minimal preparation. "NanoGalaxy" [30] was developed, and it is a freely available Galaxy-based toolkit for analyzing long-read sequencing data. PyPasWAS is a Python-based multi-core GPU and CPU sequence alignment tool. While PyPasWAS achieves a 33× speedup in execution time using GPUs for alignment, it is not a platform for running different sequence alignment tools, and it does not provide infrastructure nor an interface with Galaxy. Another recent work [116] designed infrastructure for NGS analyses using Galaxy with GPU support. While providing limited implementation details, it is an in-house framework that uses Galaxy as a middleware to run jobs along with the Slurm scheduler.

4.3 Motivation

While the current implementation of Galaxy does not allow GPU-enabled tools to run, prior research demonstrates that the GPU versions of many tools currently running in Galaxy can potentially achieve significant speedups (compared to their CPU versions). The speedups for a few life sciences applications are as follows: Direct Coulomb Summation ∼45× [145]; Cutoff Pair Potentials application ∼17× [161]; Fluorescence Microphotolysis ∼11× [10]; and Multi-Level Summation Method Short-Range application ∼25× [55].

These examples show that GPUs can improve performance in critical research areas for human life, which has shaped accelerator development in recent years. At Argonne National Laboratory, researchers study a COVID-19 vaccine, where a 24 DGX A100 system cluster empowers them to accelerate the simulations, enabling a faster understanding of how this virus infects humans [56]. Furthermore, The National Energy Research Scientific Computing Center uses A100 for AI-based simulations [56]. They had speedups up to 5× (V100 GPU vs. CPU) in different areas, and they expect more gains with A100 [56]. These results show that these use cases and tools are highly parallelizable and yield significant performance improvements when coupled with GPUs. Further, these tools are often embarrassingly parallel, allowing them to scale across multiple GPUs.

4.3.1 Challenges in Bringing GPU Support

To design and develop the GPU support, we first investigate the four main steps involved with Galaxy's tool execution flow. As seen in Figure 4.2, first, users trigger a job submission through the Galaxy web interface. Galaxy parses the tool requirements from the wrapper file, including references to the required libraries and hardware. Second, Galaxy executes the submitted job via a *runner*, which maps the job to a destination using a job configuration file. Third, Galaxy submits the job to a job scheduler or executes it locally as a dedicated process. Finally, Galaxy collects the results from the job execution and presents them to the user via the web interface. While inspecting this tool execution workflow, we identified four critical challenges in the workflow's first two steps.

Scientific Impact: Galaxy is used by thousands of researchers who significantly contribute to various important domains, executing hundreds of thousands of compute and data-intensive experiments. With GPU infrastructure support, these experiments will benefit from massive speedups due to the inherent parallelism they offer. This

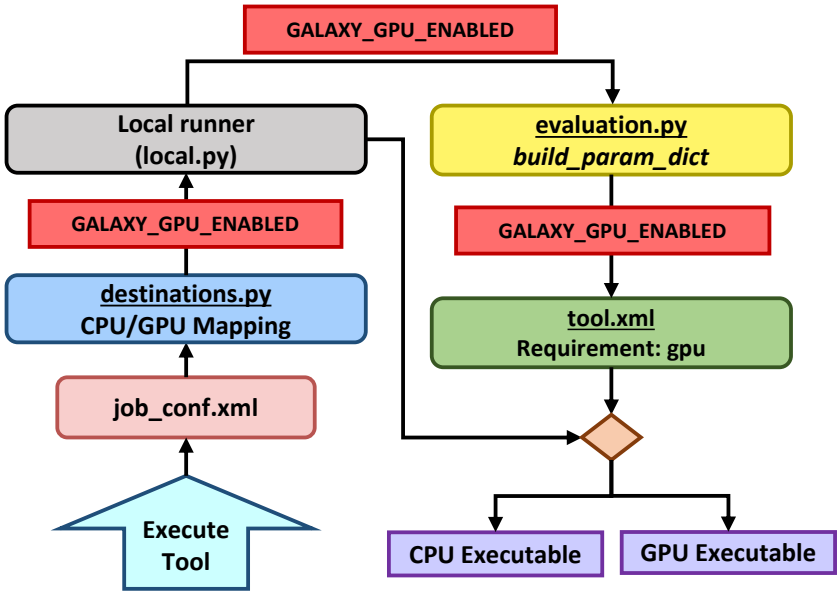


Figure 4.2: The system flow diagram for Galaxy tool execution.

advantage unilaterally motivates the need to enable GPU-aware tool execution in the Galaxy framework.

The first challenge, *Challenge-I*, is to include a new compute requirement for GPU (in the form of XML tags) in the tool configuration file. The current Galaxy implementation does not provide explicit hardware requirement specification tags. Hence, it is non-trivial to include a new hardware specification while retaining the original Galaxy execution flow. The second challenge, *Challenge-II*, lies in exposing the GPU availability to the Galaxy runner. The runner uses dynamic destination mapping to map jobs into physical hosts (destinations). The runner needs to extract information from the new GPU requirement, such as GPU availability and GPU model, to map GPU jobs alongside CPU jobs dynamically. Additionally, if GPUs are unavailable, the runner needs to switch jobs to CPU nodes in a user-agnostic fashion.

Since we need to support both bare-metal and containerized tools, *Challenge-III* is to enable GPU support for containerized tools. Although Galaxy supports launching tools like Docker containers, it does not support NVIDIA Docker-based GPU containers. The primary challenges to exploit them lie in defining a new compute requirement (in *Challenge-I*) as a part of the existing container launch script.

The final challenge, *Challenge-IV*, concerns the design of multi-GPU-aware computation mapping support. To efficiently enable multi-GPU support, we need the following: (i) allow the end-user to specify the IDs of GPUs as a requirement; (ii) obtain the real-time

GPU information such as GPU IDs, the number of executing processes, and memory usage, and (iii) design a GPU device allocation strategy without (or minimally) affecting the currently running processes. These four challenges highlight the design complexities of enabling GPU support in Galaxy.

4.4 Design and Implementation of GYAN

To address the challenges in Section 4.3.1, we design and implement GYAN – a GPU-aware computation mapping support for Galaxy. While retaining the original execution flow of Galaxy, the essential features of GYAN are (i) minimal to no user involvement, (ii) easily extensible code enhancements to Galaxy's core framework, (iii) minimal overheads for cluster administrators, and (iv) under-the-hood automated decision-making process. Below, we explain the individual design components of GYAN in detail.

4.4.1 GPU-Aware Computation Mapping

Challenge-I requires designing a new compute requirement of type GPU in XML tags. To overcome this challenge, we designed and implemented a new *parser*, which interprets the new requirement type. The new requirement type will be used in the tool wrapper file as shown in Code 4.1. The new requirement type is utilized when deciding if GPU is required in the upcoming steps of the Galaxy execution flow. The values of the compute requirement type can be "gpu" or "cpu" (default).

```
1 <macros>
2   <xml name="requirements">
3     <requirements>
4       <requirement type="package" version="@VERSION@">racon</requirement>
5       <requirement type="compute">gpu</requirement>
6     </requirements>
7   </xml>
8   ...
9 </macros>
```

Code 4.1: The `macros.xml` file. It specifies the tool's requirements and is imported into the racon.xml file. The type "gpu" requirement is at line 5, allowing Galaxy to recognize that the tool required GPU to be executed.

Recall that *Challenge-II* exposes the GPU availability to the Galaxy runner. We added a new job rule to address this challenge, which allows us to dynamically map between CPU or GPU destinations according to different conditions. The job rule obtains GPU availability and the number of GPUs using the "`pynvml`" Python library. If the tool's wrapper file has the compute requirement of type "gpu" and at least one GPU is

available, the destination is configured to be "local GPU." At the same time, a boolean environment variable called "GALAXY_GPU_ENABLED" is introduced to Galaxy; it is set to "true" if the "local GPU" destination is configured and "false" otherwise. The Galaxy administrators configure the job execution destinations using the "job_conf.xml" file. The new job rule is used in the "job_conf.xml" file as a "destination" (see Code 4.2).

```
<job_conf>
  <plugins>
    ...
    <plugin id="dynamic" type="runner">
      <param id="rules_module">galaxy.jobs.rules</param>
    </plugin>
  </plugins>
  <destinations default="gpu_cpu_decision">
    <destination id="gpu_cpu-decision" runner="dynamic">
      <param id="type">python</param>
      <param id="function">dynamic_map</param>
    </destination>
  </destinations>
</job_conf>
```

Code 4.2: The "job_conf.xml" configuration file that specifies the dynamic job destination decision making dynamic_destination.py script as the runner.

In the Galaxy framework, the backend Python variables are exposed to the tool developer with the dictionary data structure, which is the output of the "build_param_dict" function. This function resides in the "evaluation.py" script and bridges the Galaxy backend and the tool developer. Using this information, we exposed the "GALAXY_GPU_ENABLED" environment variable to the tool wrapper file by inserting a dictionary entry. We assign the "GALAXY_GPU_ENABLED" value to ‘‘__galaxy_gpu_enabled__" in the tool-config file.

Code 4.3 shows how the tool wrapper file utilizes the "__galaxy_gpu_enabled__" key from the parameter dictionary. It the value of "__galaxy_gpu_enabled__" and decides which executable to use. By default, in CUDA programming, if the tool does not specify any GPU device preference, all the GPUs are made available. However, if the tool has a specific GPU preference within the requirements XML, then that GPU is used. If the GPU is busy, the tool will be offloaded to another GPU device based on availability.

```
1  <tool id="racon" name="Racon" version="@VERSION@">
2    ...
3    <command detect_errors="exit_code"><![CDATA[
4      ...
5      #if $__galaxy_gpu_enabled__=="true":
6        racon_gpu
7      #else:
8        racon_cpu
9      #end if#
10     ...
11   ]]></command>
12 </tool>
```

Code 4.3: The "`racon.xml`" wrapper file for the Racon tool. The wrapper files allow users to specify the executable and include the parameters that the end user can set. This file is necessary for Galaxy to recognize the tool. The "`GALAXY_GPU_ENABLED`" environment variable is accessed via the parameter dictionary entry "`galaxy_gpu_enabled`" (line 5).

4.4.2 GPU-Awareness for Containerized Tools

Challenge-III is about enabling GPU support for containerized tools. Although Galaxy supports launching tools as containers and a tool's container has support for GPU, Galaxy does not launch the containers with GPU support. To overcome this challenge, we must modify the container launch script to utilize the GPU compute requirement that the tool developer specifies using the wrapper file.

In the original Galaxy framework implementation, when the `job_conf.xml` file has the "`docker_enabled`" parameter set to "`true`" [122], the Docker runner takes effect. Next, the Galaxy container launching script reads the required container ID from the tool wrapper file and pulls it from the docker hub or `bioconda` [50]. Subsequently, the script executes the container by assembling a bash command. While assembling this command, GPU support for Docker and Singularity-containerized tools can be added using an additional flag. The machine or cluster hosting Galaxy must have NVIDIA-Docker installed so that the user driver components and the GPU devices in the container are mounted to the container at launch.

To add the GPU support, we must first obtain information about GPU availability and the tool's GPU requirement. To this end, we use a similar approach to the one described in Section 4.4.1. The destination to execute the tool is changed to the "docker" destination in the job_conf.xml configuration file. If there is no GPU available, the NVIDIA-Docker library will not work. When we are adding the new GPU flag with the `command_part.append(''--gpus all'')` line to the Docker run command, we first check the environment variable that is set according to the GPU availability and requirement using

the `if os.environ[‘GALAXY_GPU_ENABLED’] == ‘‘\texttt{true`" statement.

We added GPU support to Singularity-containerized tool execution, similar to the GPU support design for Docker-containerized tool execution. If the "`GALAXY_GPU_ENABLED`" environment variable value is "`true`", the GPU support is added to the Singularity container launch command using the `command_part.append(‘‘--nv’’)` statement. Theoretically, this addition should suffice for GPU support for Singularity containers. However, in the current Galaxy framework implementation, when volumes are mounted to the Singularity image, the "rw" and "ro" flags are given for the read-write or read-only permissions. These two flags are removed with the GYAN enhancements because Singularity's new version (Version 3.1) does not support these flags when adding the GPU flag.

4.4.3 Multi-GPU-Aware Computation Mapping

In addition to the single GPU support, we also provide a multi-GPU computation mapping support, which executes a given tool using multiple GPUs, provided that there is more than one GPU available on the host. To this end, the first part of *Challenge-IV* enables the end-users to specify the IDs of GPUs for tools as requirements in the wrapper files. The non-trivial challenge here is identifying the GPU ID and the already-defined GPU compute requirement. We used the existing "version" XML tag of the tool wrapper file's required objects to solve this problem. Therefore, the "version" tag corresponds to our design's GPU minor ID(s). The second part of this challenge involves obtaining real-time GPU information such as GPU IDs, executing processes, and memory usage for each GPU. To solve this part of the challenge, we propose an algorithmic design that determines the processes running on each GPU. This information is obtained by executing a GPU query command from the Python local runner script ("`local.py`"). This command uses the "`nvidia-smi`" query and returns the output as XML. The "`BeautifulSoup`" library processes the XML output to extract the GPU IDs and PIDs of the processes executing on each GPU. As shown in Algorithm 4.1, a dictionary with the key-value pairs is created by processing the output, where keys are GPU minor IDs and values are process IDs.

The next part of *Challenge-IV* is to design a GPU device allocation strategy without affecting the currently executing processes. To solve this challenge, we introduce two methodologies, which are explained below.

Algorithm 4.1 The "`get_gpu_usage`" function in the "`local.py`" script. This function captures the executing processes for each device and returns a list of available GPUs and all GPUs in the system.

```
Input : None
Output : avail_gpus, all_gpus
proc_gpu_dict ← {}
 avail_gpus ← []
 all_gpus ← []
 bash_cmd ← "/bin/bash -c 'nvidia-smi –query -x' "
 out,  err ← subprocess.Popen(...)
 soup ← bs(out, "lxml")
 gpu_find ← soup.find("nvidia_smi_log").find_all("gpu")
 process_find ← p.find("processes").find_all("process_info")
 for p in gpu_find do
 |   minor_id ← p.find("minor_number")
 |    for proc in process_find do
 |    |   pid_proc ← proc.find("pid")
 |    |    proc_gpu_dict[minor_id].append(pid_proc)
 |    |
 |   end
end
for x,y in proc_gpu_dict do
 |   all_gpus.append(x)
 |    if y is empty then
 |    |   avail_gpus.append(x)
end
```

4.4.3.1 Process ID Approach

The "`__command_line`" function in the "`local.py`" script assembles a command to execute the tool submitted by the end-user and launches the command as a subprocess. The "`get_gpu_usage`" function is called in the "`__command_line`" function. It returns the lists of available GPUs and all the GPU IDs on the host machine. If the list of available GPUs contains the required tool's GPU IDs, the value of the "`CUDA_VISIBLE_DEVICES`" environment variable is set to those GPU IDs. Next, it is exported to the local runner as shown in Algorithm 4.2.

Like the earlier approach, GPU support for containerized tools with multi-GPU support is developed by exposing GPU information to containers. Note that we have not used the "`-gpus x`" command, which is meant to expose the desired GPUs because it did not work as intended. Instead, we have exported the "`CUDA_VISIBLE_DEVICES`" environment variable according to the GPU availability. Then, we used the "`-gpus all`" flag to obtain all GPU IDs that the environment variable exposed. These GPU IDs are determined with Algorithm 4.2. This flag is necessary for the Docker runtime to support the GPU-enabled containerized tools.

Algorithm 4.2 The "`__command_line`" function in the "`local.py`" script.

```
input  : self, job_wrapper
output : CUDA_VISIBLE_DEVICES
if job_wrapper.tool exists then
    for req in reqmnts do
        if req.type = "compute" and req.name = "gpu" then
            if req.version and req.version != "" then
                gpu_id_to_query ← req.version
            flag ← 1
    end
    if gpu_flag and gpu_count > 0 and flag then
        GALAXY_GPU_ENABLED ← "true"
    avail_gps, all_gps ← get_gpu_usage()
     for dev in all_gps do
        all_gps_str += dev
    end
    if gpu_id_to_query in avail_gps then
        gpu_dev_to_exec ← gpu_id_to_query
    else
        gpu_dev_to_exec ← ""
         for dev in avail_gps do
            gpu_dev_to_exec += dev
        end
    CUDA_VISIBLE_DEVICES ← gpu_dev_to_exec
```

4.4.3.2 Process Allocated Memory Approach

The "Process ID Approach" explained may not be efficient for some scenarios. For example, if all GPUs execute at least one process and an incoming task is distributed to all GPUs, some GPUs can have very high memory utilization. This situation may cause stalling due to context switching between tasks. Instead, with the "Process Allocated Memory Approach," we place the upcoming job on a GPU with the least instantaneous device memory allocated for executing the process(es). This approach uses the "`nvidia-smi`" calls mentioned in Section 4.4.3.1. Instead of obtaining the process IDs from this query, we get each GPU's "`fb_memory_usage.used`" value. We then expose the GPU ID with the minimum memory usage to the upcoming job.

4.5 Evaluation Framework

In this section, we discuss the evaluation of GYAN using the experiments conducted in Galaxy hosting on a Chameleon Cloud testbed, which has GPU nodes. We compared runtimes of CPU-only executions and GPU-supported executions along with the multi-GPU enhanced Galaxy executions for two tools. With the use of GYAN, running GPU-supported tools on Galaxy does not introduce any extra overhead because GYAN

executes and schedules jobs to GPUs without adding another layer of the software stack. Therefore, to evaluate GYAN, we showed how GPU-supported tools have speedup over CPU-only versions to motivate GPU computation mapping for Galaxy. These speedups can increase the number of computing and data-intensive experiments, leading to increased research bandwidth.

4.5.1 Workloads

We focus on two significant workloads for testing GYAN: *Racon* and *Bonito*. These are popular tools for processing next-generation sequencing data [127] from the two most popular long-read technologies – PacBio [131] and Oxford Nanopore Technologies [146]. We chose them because they capture a broad range of sequencing data analysis, covering multiple platforms and stages of the data processing pipeline. Besides, they are highly amenable to GPU-based parallelization.

Sequencing data typically goes through a processing pipeline before it can lead to biological insight. One of the earliest steps in the pipeline is "basecalling" [101]. A basecaller converts sequencing data from its raw form, capturing the complex combination of optical sensors, hardware, and chemistry underlying the technology into a sequence of individual nucleotides. Oxford Nanopore Technologies provides a PyTorch-based basecaller for its data, Bonito [172]. Bonito is inspired by convolutional neural networks (CNNs) in speech recognition. It has several functionalities, like training a bonito model (*bonito train*), converting an `hdf5` training file into a bonito format (*bonito convert*), evaluating a model performance (*bonito evaluate*), downloading pre-trained models and training datasets (*bonito download*), and basecaller that obtains a fasta format output from "`.fast5`" files (*bonito basecaller*). Bonito supports both GPU and CPU execution. It has automatic mixed-precision support for accelerating the training tool [172].

Basecalled reads are then often used to perform a de novo assembly. An *assembler* is a program that outputs long reference sequences for shorter read segments as it predicts the sources of these reads. To this end, the assembler first constructs a draft backbone sequence of the reference. It then aligns the reads to that backbone and corrects each position in the backbone according to the consensus of the nucleotides that align with it. *Racon* is a module that performs this step for PacBio sequencing data. While this is a compute-intensive process, it leads to significantly better quality assemblies. Racon uses the mapping data to construct a partial-order alignment with single-instruction, multiple-data (SIMD) support to accelerate the consensus generation order of magnitude faster than state-of-the-art methods [196].

4.5.2 Experimental Setup

We used a machine with an Intel Xeon E5-2670 processor with 48 CPUs and two NVIDIA Tesla K80 GPUs to conduct our experiments. The GPU driver version is 455.45.01, and the CUDA version is 10.2. We used Python, Version 3.6.9, to execute Galaxy. We created a different set of experiments to analyze all of the contributions. The first set of experiments executes the Racon tool with the local runner and compares the performance with various parameters and the GPU vs. CPU execution. Next, we created a Docker image for the Racon-GPU tool and used that to compare the execution times of runs with different parameters and GPU vs. CPU. Lastly, to evaluate the multi-GPU functionality, we created various cases to test the multi-GPU process scheduling.

4.5.3 GPU Hardware Usage Script

We implemented a script that allows us to collect statistics about the executing jobs. This script obtains the GPU utilization, GPU memory utilization, and PCIe link generation information for every second, including minima, maxima, and average. It is executed when a job is submitted and stopped when a job is either killed or stopped. Whenever it stops, a post-processing function is executed, generating `.csv` files and other log and statistic files aggregated from the chronological data for each job. Code 4.4 shows the GPU query used for this script. This script captures the increasing/decreasing trends of the GPU memory usage and GPU utilization of the executing tool. It demonstrates how beneficial GPU usage is and leads to the multi-GPU computation mapping support design (cf. Section 4.4.3).

```
1 bash_command = "/bin/bash -c 'nvidia-smi query-gpu=utilization.gpu, utilization.memory,
      memory.total, memory.free, memory.used, pcie.link.gen.max, pcie.link.gen.current
     --format=csv -l 1'"
2 sp = subprocess.Popen(bash_command, shell=True, stdout=File_object, stderr=subprocess.
     PIPE).pid
```

Code 4.4: GPU metric query for obtaining hardware usage metrics. This code snippet is added to the "`queue_job`" function of the "`local.py`" runner script.

4.6 Results and Analysis

4.6.1 GPU-Aware Computation Mapping

To test the impact of GYAN, we ran the Racon-GPU tool on Galaxy. We used a 17 GB Alzheimers NFL Dataset containing the polished sequencing results for the Alzheimer

human brain transcriptome [147]. After experimenting with all of the batch sizes and the number of CPU threads with the Racon tool, the configuration that gave the best performance was four threads and one batch without banding approximation, and it took 1.72s. Among the experiments that use banding approximation, four threads and 16 batches performed the best with 1.67s. The CPU-only execution using four threads took 3.22 seconds, or nearly 2× slower than the GPU execution, as seen in Figure 4.3.

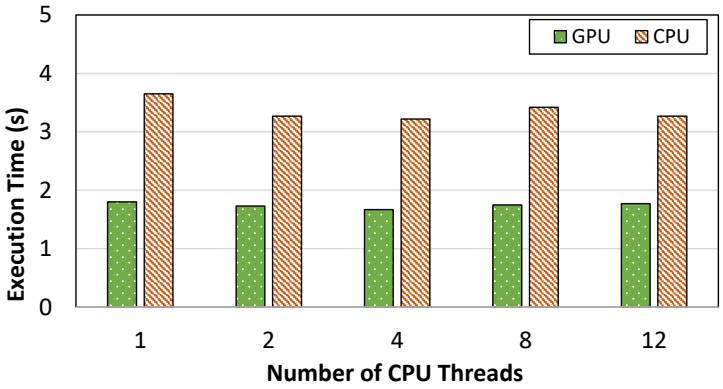


Figure 4.3: Performance across different thread numbers for the Racon tool comparing the GPU and CPU-only versions.

To understand the reason for the speedup and the nature of the Racon-GPU tool, we used the GPU hardware usage script given in Section 4.5.3. We found that the Racon-GPU tool does the "polishing" portion of the consensus generation in GPU. While the CPU-only polishing execution takes 117 seconds, using GPU, this execution time is reduced to 15 seconds (2 seconds for GPU memory allocation + 13 seconds of GPU polishing + 0.0001 seconds of additional CPU polishing for the remaining portion of the reads that could not be polished in GPU). Thus, the CPU end-to-end execution takes ∼410s for the Isoseq NFL Alzheimers dataset, and the GPU end-to-end execution takes ∼200s. So, even though polishing time is reduced from 117 to 13 s, there is an overhead of ∼40s due to CUDA API calls to transfer input data and results from and to GPU for kernel computation of polishing and CUDA kernel synchronization.

To see the hotspots and understand how much the API calls affect the performance, we performed NVProf analysis on the running job. As plotted in Figure 4.4, most calls are kernel synchronization and memory transfer API calls (which send the 17 GB dataset in chunks that fit in GPU memory to the device and back to the host). Lastly, the ClaraGenomics library kernel calls, which are "`generatePOAKernel`" and "`generateConsensusKernel`." To understand the bottlenecks, we did an NVProf stall analysis on Racon. We found out that there is ∼70% memory dependency stall and

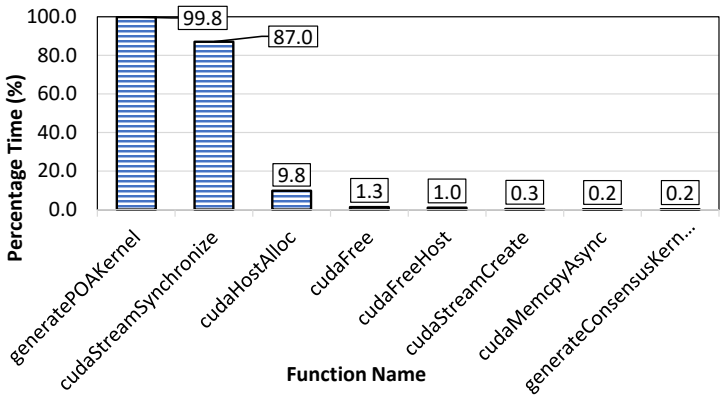


Figure 4.4: Hotspot functions obtained by doing NVProf analysis on the Racon-GPU workload.

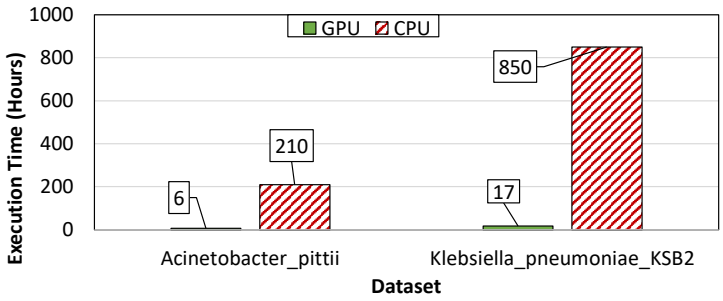


Figure 4.5: Bonito CPU vs. GPU execution times for two datasets.

∼20% execution dependency stall, which is why we cannot get further performance improvements.

We experimented with the Bonito Basecalling tool (pip package, version 0.3.2) using `Acinetobacter_pittii` (1.5 GB) and `Klebsiella_pneumoniae_KSB2` (5.2 GB) datasets [135]. Figure 4.5 shows that the GPU support reduced the execution time significantly. The figure shows the execution times for CPU as approximate results because the CPU execution time for the smaller dataset (`Acinetobacter_pittii`) lasted more than 210 hours, and the larger dataset is approximated to last 4× longer than the smaller dataset (more than 850 hours). Therefore, GPU's speedup vs. CPU is more than 50× for the Bonito Basecaller tool. We did an NVProf analysis for the Bonito Basecaller tool. Figure 4.6 shows the Bonito hotspot functions, which are CUDA kernel launcher, kernel synchronizer functions, and GEneral Matrix to Matrix Multiplication (GEMM) functions; these are a critical part of neural networks. Bonito Basecaller uses a pre-trained network.

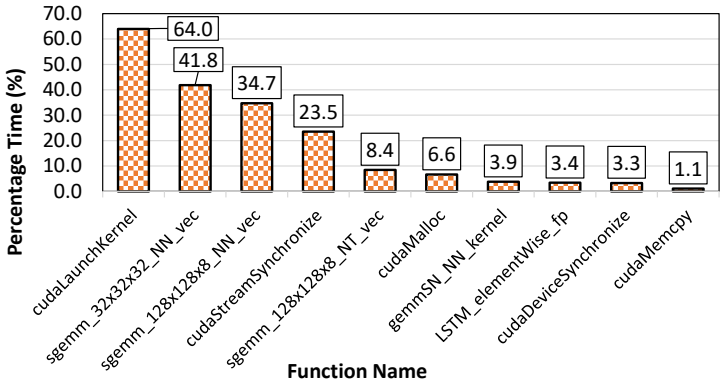


Figure 4.6: Bonito hotspot functions obtained by doing NVProf analysis.

4.6.2 GPU-Awareness for Containerized Tools

We used the Racon-GPU tool and the Alzheimers NFL Dataset to test the GPU support for containerized tools. We experimented with the same parameters and settings to compare the bare-metal and containerized versions of Racon to infer the container launching overhead and the speedup between the CPU and GPU-aware containerized execution of Racon. We created a Racon-GPU Docker container that can be accessed online [51] or can be pulled using the "docker pull gulsumgudukbay/racon_dockerfile" command. We used this container as a requirement for the tool wrapper. The best performance configuration for the Racon experiments that did not use banding approximation was with 2 CPU threads and four batches for accelerated GPU polishing. The best configuration for the experiments that used banding approximation was using 2 CPU threads and eight batches for accelerated GPU polishing, as seen in Figure 4.7. We compared the two best-performing configuration performance numbers and calculated that approximately 0.6s (36%) of the time was spent on container launching and cold start overhead.

4.6.3 Multi-GPU-Aware Computation Mapping

We used Racon-GPU and Bonito Basecaller tools to evaluate the multi-GPU-aware computation mapping and executed them on Galaxy. We used the Bonito pip package version 3.2.0 and assembled several different experiments.

Case 1: Two Different Tools This experiment tested the most basic functionality of multi-GPU-aware computation mapping, checking whether two jobs will be scheduled to their required devices. If the user specified Device 0 as the Racon tool's compute requirement and Device 1 as the Bonito tool's compute requirement, the experiment

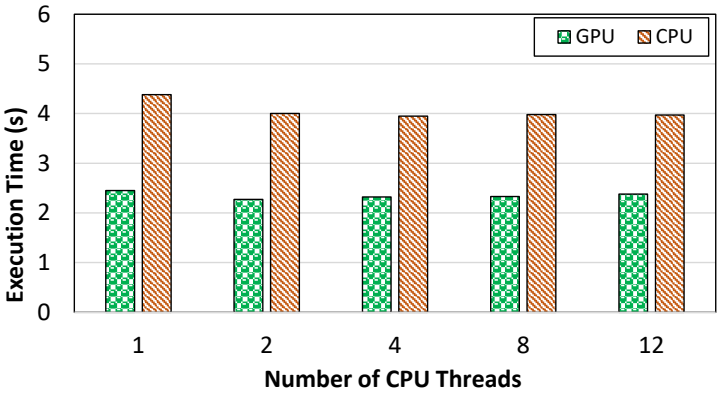


Figure 4.7: Performance across different thread numbers and batch sizes for Racon-GPU with banding approximation executed on Docker containers.

```
Every 0.1s: nvidia-smi                                  k80-gpu: Tue Dec 15 0

Tue Dec 15 08:45:14 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 455.45.01    Driver Version: 455.45.01    CUDA Version: 11.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 00000000:05:00.0 Off |                    0 |
| N/A   48C    P0    60W / 149W |     63MiB / 11441MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K80           Off  | 00000000:06:00.0 Off |                    0 |
| N/A   40C    P0   145W / 149W |   2734MiB / 11441MiB |     95%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A     27755      C   ...axy/tools/racon/racon_gpu       60MiB |
|    1   N/A  N/A     27039      C   .../galaxy/.venv/bin/python3     2731MiB |
+-----------------------------------------------------------------------------+
```

Figure 4.8: Multi-GPU support Case 1 console output.

checked if the jobs were scheduled to their correct GPUs. Figure 4.8 shows the console output for this case. Figure 4.9 shows this case under Case 1 along with the "`nvidia-smi`" query output of the experiment, which shows the processes and the GPUs allocated to them. The Racon process runs on GPU 0, and the Bonito process runs on GPU 1, indicating that the basic functionality works as intended. Therefore, we can conclude that two different tools can be executed in parallel in separate GPUs without performance degradation, running at their original execution times.

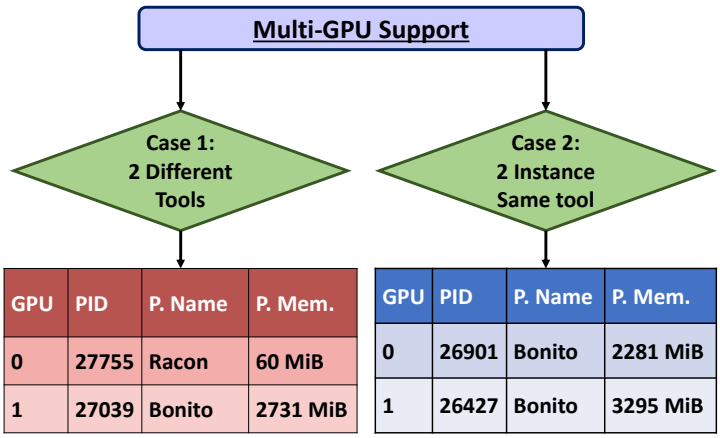
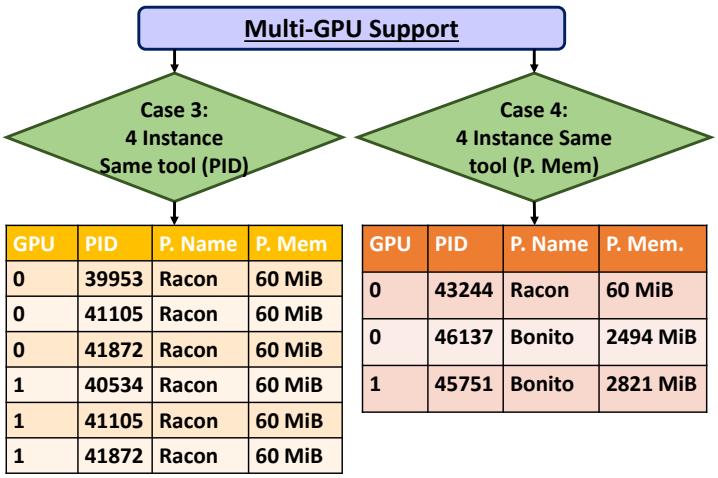


Figure 4.9: Multi-GPU support Cases 1 and 2.

Case 2: Two Instances of the Same Tool This experiment involved scheduling a second instance of the same tool to another GPU if the first required GPU is busy with an instance of the tool. If Bonito's required GPU ID is 1, the instance should be scheduled in Device 1. If another instance of Bonito is started with the same GPU ID 1, it will be scheduled to a not-busy GPU (GPU 0), as seen in Figure 4.9, Case 2.

Case 3: Four instance of the Same Tool GPU Allocation Using PID This experiment involved executing more than two instances of the containerized Racon-GPU tool (showing that both "PID GPU allocation" and "multi-GPU support for containerized tools" are working). As can be observed from Figure 4.10 and the console output in Figure 4.11, the scheduling works as intended: the first Racon instance (PID 39953) is scheduled to GPU 0, the second (PID 40534) to GPU 1, and then, since both GPUs are busy, the upcoming processes with PIDs 41105 and 41872 are scattered to both GPUs.

Case 4: Four instances of the Same Tool GPU Allocation Using Process Memory This experiment involved executing instances of Racon and Bonito tools and another instance of Bonito. Figure 4.10, Case 4 shows that Racon (PID 43244) is scheduled to GPU 0, Bonito (PID 45751) is scheduled to GPU 1, and the second instance of Bonito (PID 46137) is scheduled to GPU 0. When the user executes the second instance of Bonito, the GPU with the minimum memory usage is GPU 0 (with 60 MiB usage). This case is a better approach than the previous one because it allows more efficient device allocation than distributing the third process to all GPUs and introducing multi-GPU overhead for tools that do not have multi-GPU support.



GPU	PID	P. Name	P. Mem
0	39953	Racon	60 MiB
0	41105	Racon	60 MiB
0	41872	Racon	60 MiB
1	40534	Racon	60 MiB
1	41105	Racon	60 MiB
1	41872	Racon	60 MiB

GPU	PID	P. Name	P. Mem.
0	43244	Racon	60 MiB
0	46137	Bonito	2494 MiB
1	45751	Bonito	2821 MiB

Figure 4.10: Multi-GPU Support Cases 3 and 4.

```
+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A     39953      C   /usr/bin/racon_gpu                60MiB |
|    0   N/A  N/A     41105      C   /usr/bin/racon_gpu                60MiB |
|    0   N/A  N/A     41872      C   /usr/bin/racon_gpu                60MiB |
|    1   N/A  N/A     40534      C   /usr/bin/racon_gpu                60MiB |
|    1   N/A  N/A     41105      C   /usr/bin/racon_gpu                60MiB |
|    1   N/A  N/A     41872      C   /usr/bin/racon_gpu                60MiB |
+-----------------------------------------------------------------------------+
```

Figure 4.11: Multi-GPU support Case 3 console output.

4.7 Chapter Summary

The explosion of parallel bioinformatics applications has driven deployment platforms like HPC supercomputers and public data centers to embrace GPUs as a part of their hardware ecosystem. Nevertheless, the most popular bioinformatics application framework, Galaxy, does not support GPU-based applications. To fill this void, in this chapter, we propose GYAN, an enhanced version of Galaxy with GPU support, which allows the GPU-capable tools to execute in Galaxy. More specifically, we augment an intelligent GPU-aware computation mapping and orchestration support to Galaxy so researchers can execute the tools in both CPU (or GPU) based on the tool requirements. The GPU-aware computation mapping was extended to multi-GPU application mapping and orchestration.

Furthermore, we also enabled GPU containerization support for both Docker and Singularity containers. We performed experiments using two tools. Our experiments revealed that the GPU support through GYAN leads to ∼2× improvement in performance using the Racon-GPU tool in a 17 GB dataset and ∼50× improvement for the Bonito

base calling tool. Also, the intelligent multi-GPU-aware computation mapping allowed us to efficiently allocate GPUs to jobs according to the states/occupancy of each GPU and execute many instances of different tools simultaneously without degradation in performance. GYAN's source code is merged into Galaxy's main repository.

Chapter 5
Multithreaded Performance Optimization Using a Compiler-Guided Strategy

5.1 Introduction

As HPC applications have increased dataset sizes and computational demands, their memory accesses dominate end-to-end execution times. Due to the QoS requirements, optimizing the number of memory accesses and data locality is of paramount interest. In this chapter, to tackle maximizing the performance of multithreaded applications, we propose a novel compiler-guided strategy to balance data locality and recomputation optimizations to minimize data access latencies in multi-threaded applications.

Data locality remains among the most pressing problems in many HPC application domains. While numerous data locality optimizations have been proposed in the past (e.g., see [79, 90, 104, 108, 162, 205] and the references therein), ever-increasing dataset sizes and architectural complexity of emerging multicore and manycore systems limit the potential of these optimization techniques, demanding more radical solutions to the data locality problems. Recent works on accelerator design [14, 84, 214], near data computing [47, 80, 139, 183], and 3D memories [88, 105, 106] are examples such new approaches.

Data Recomputation [3, 4, 193] is one approach geared towards eliminating costly data accesses by replacing each such access with multiple less costly data accesses plus some (extra) computation. For example, if accessing variable "a" is predicted to be very expensive (e.g., it is in off-chip memory) but its value can be (re-)computed using values

of two variables, "b" and "c," which are less costly, it may be more profitable to *recompute* "a" using "b" and "c," also incurring some extra computations along the way. Existing works in recomputation (including hardware [91,92] and software-centric [3,4,19,168,193] ones) have focused on the exploration of the potential of recomputation in single-threaded application programs and conventional cache/memory hierarchies. Such efforts leave two aspects of data recomputation unexplored: (i) *They need to handle multithreaded applications*, that is, in pursuing data recomputation opportunities, they need to consider inter-thread interactions. We believe significant performance gains can be achieved if one can devise techniques that allow a thread to recompute on behalf of other threads when it is profitable. More generally, can we look at a multi-threaded application as a whole and, for each data recomputation opportunity, identify the thread that is best suited for it?, and (ii) *They do not explore the role that the different caching policies/strategies can play in improving the effectiveness of data recomputation.* Our results indicate that tuning the cache replacement decisions according to the needs of the data recomputation paradigm can further boost application performance.

Figure 5.1 depicts various scenarios for which data recomputation may be desirable in a 6×6 manycore system. For example, in Figure 5.1 (a), instead of accessing $a[i]$ from main memory, it uses on-chip elements $b[i]$ and $c[i]$ to recompute the value of $a[i]$, assuming it is possible to do so.[1] Figure 5.1 (b) illustrates a similar situation, but this time all three data elements are on-chip, except $a[i]$ is a faraway location, from the perspective of the computing thread, compared to the other two data elements ($b[i]$ and $c[i]$) which are close-by. In Figure 5.1 (c), instead of letting the computing thread (marked red) access $a[i]$ from main memory, we ask another thread (its location is marked blue) to perform recomputation, *on behalf of* the red thread. Figure 5.1 (d) shows a similar scenario, except that all data elements involved are on-chip. Finally, in Figure 5.1 (e), the value of $a[i]$ needed by the red thread ($(b[i]+c[i])$-$d[i]+f[i]$) is collaboratively computed by two threads (yellow and purple). Our contributions are as follows:

- We propose a compiler-guided strategy that balances data locality and recomputation optimization to minimize data access latencies in multi-threaded application executions. We argue that, in addition to the opportunities provided by data recomputation within each thread separately, in an isolated fashion, one can achieve further performance benefits from data locality-aware recomputation by allowing threads to employ recomputation *not* only for their own sake but also for other threads' sake, i.e., we propose "recomputation across threads."

[1] Data recomputation may not always be legal; we discuss its legality conditions in the sequel.

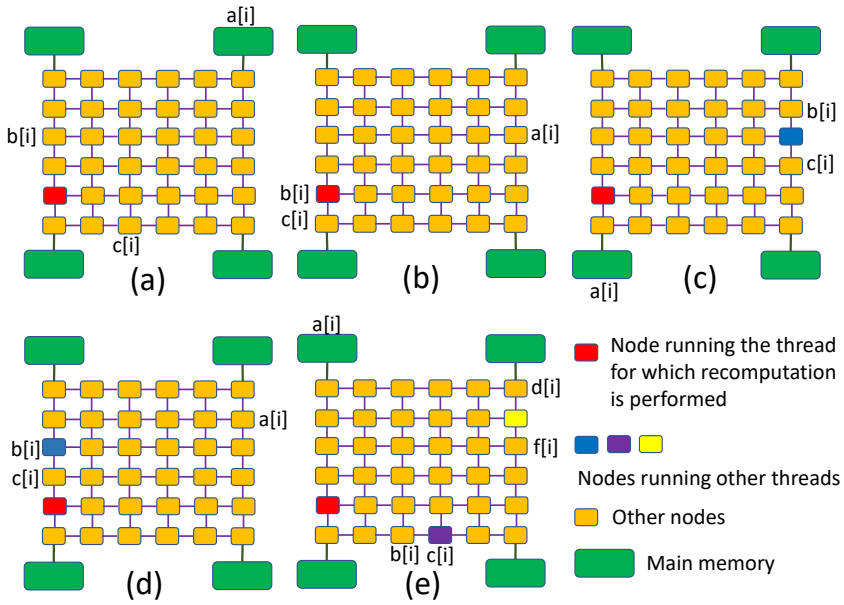


Figure 5.1: Different scenarios showing how data recomputation can be beneficial in a 6×6 manycore system.

- We demonstrate that significant performance gains are possible by designing a new compiler-guided cache replacement strategy. The unique aspect of our strategy is that it makes its replacement decisions based on *not* only recency information (as in the case of traditional LRU) but also future/perceived data recomputation opportunities. This new approach is called "recomputation-conscious caching."

- We evaluate our approach with 14 multithreaded applications. The results show that our approach improves performance by an average of 13.25% over the conventional optimizations that do not use recomputation and 7.68% over single-thread centric recomputation. The corresponding improvements when employing recomputation-conscious caching are 19.12% and 11.63%, respectively.

5.2 Background and Target Architecture

We focus on loop-intensive applications programs where loop bounds and array references are "affine functions" of enclosing loop indices and loop-independent variables/constants. Note, however, that the application program being optimized can have non-affine data references (e.g., non-affine subscript expressions) as well, but our recomputation optimization focuses on the affine data references and makes "conservative assumptions" on non-affine data references. A loop nest is represented by an iteration vector $\vec{I} =$

$(i_1, i_2, \cdots, i_n)^T$, where i_1, i_2, $\cdots$, i_n are loop iterators from the top and a system of inequalities representing the loop bounds $\zeta\vec{I} \leq \varphi$. An array reference within such a loop nest is represented by $\Omega\vec{I} + \omega$ where Ω is referred to as "access matrix" and ω as "offset vector." As an example, if a loop with two iterators, i_1 (outer) and i_2 (inner), contains an array reference such as $X[i-2][i+j+1]$, we have

$$\Omega = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \text{ and } \omega = \begin{pmatrix} -2 \\ 1 \end{pmatrix}.$$

We focus on many-core systems where multiple cores are connected using an on-chip network. In this work, we perform experiments on Intel Sky Lake architecture (*mobile* Core i7-6700HQ CPU) and a simulated version of it (the latter is for our recomputation-conscious caching experiments). We assume that the cache hierarchy (which consists of L1, L2, and L3) is "non-inclusive" (a data element in Li does not necessarily mean that it is also available in Lj where i<j). We focus on "shared-memory" systems; hence, we assume our applications are manually parallelized using multiple threads (targeting a logically shared memory system) by the programmers themselves or automatically by a parallelizing compiler. Our approach also makes use of a "latency estimator", which estimates, at compile time, the cost of accessing a given data element at any given point in execution at runtime, taking into account its location as well as the distance to it on the on-chip network. The primary purpose of this estimator is to enable the compiler to decide when it is beneficial/profitable to employ data recomputation. While different ways exist to implement such a latency estimator, ours is built upon the cache miss equations (CME) [43]. In the subsequent sections, we explain our changes to the original implementation of CME to accommodate the data recomputation-specific needs.

5.3 Motivation

5.3.1 Why Recomputation?

We start by considering the code fragment in Code 5.1 (in a hypothetical block-based imperative language), assuming that a, b, c, d and e are data arrays and *for* and *endfor* are the language-supplied constructs that specify the boundaries of the loop body.

Let us assume the accesses to other arrays between lines 3-5 displace $a[i]$ from the caches closer to the core to those away from the core (perhaps to main memory) before the execution reaches line 6. After the execution of line 6, both $b[i]$ and $c[i]$ are expected to be

```
1 for i := 1, N
2   a[i] = b[i] + c[i]
3   ...  = ...   # accesses to other arrays
4   ...  = ...   # accesses to other arrays
5   ...  = ...   # accesses to other arrays
6   ...  = b[i] - c[i]
7   ...  = a[i] / e[i]
8 endfor i
```

Code 5.1: Recomputation potential.

in the cache closest to the core executing this program. So, in line 7, instead of accessing $a[i]$ from a faraway location (and spending lots of cycles for that), we can *recompute* its value using $b[i]$ and $c[i]$ which are closer to the core, that is, as the right-hand side (RHS) of the assignment statement in Line 7, we re-execute $(b[i] + c[i])/e[i]$, instead of $a[i]/e[i]$. To be more concrete, when the execution reaches line 7, if the (estimated) access costs for $a[i]$, $b[i]$, and $c[i]$ are 30 cycles, two cycles, and two cycles, respectively, and performing an addition operation ("+") takes one cycle, we can save 30-(2+2+1)=25 cycles by using recomputation. For this recomputation to be legit (semantic-preserving), neither $b[i]$ nor $c[i]$ should be updated between lines 2 and 7. Besides, employing recomputation causes problems, e.g., potentially negative interactions with cache locality, which we will discuss in the following parts.

5.3.2 Recomputation across Different Threads

While we have focused on recomputation opportunities from a single-thread perspective in the code example above, threads can also perform recomputation on behalf of one another. Consider two threads that are scheduled to execute (in parallel) the code fragments in Code 5.2: In this scenario, when thread-I reaches line 6 to access $a[i]$, it may be able to recompute $a[i]$'s value (by performing $b[i] + c[i]$) because both $b[i]$ and $c[i]$ are expected to be in some "shared cache" between these two threads, as this thread-II has just accessed them in its lines 4 and 5. This scenario is called "recomputation enabled by other threads" (shortened as "`REOT`"). Alternatively, if it is more beneficial than the strategy above, thread-II can itself perform $b[i] + c[i]$ *on behalf of* thread-I and assign the result to a temporary variable $t[i]$, which thread-I can subsequently access. This scenario is called "recomputation on behalf of other threads" (shortened as "`RBOT`"). These two cases are depicted in Figure 5.2: (a) shows the situation just before the recomputation (of $b[i] + c[i]$ to replace access to $a[i]$) takes place, whereas (b) and (c) illustrate REOT and RBOT, respectively. Note that, in both (b) and (c), the off-chip memory is not accessed; so, both these options are expected to lead to a lower latency than accessing $a[i]$ directly from the off-chip memory.

```
1 thread-I:                  thread-II:
2
3 a[i] = b[i] + c[i]         ... = ...
4 ...  = ...                 ... = b[i] - d[i]
5 ...  = ...                 ... = c[i] - e[i]
6 ...  = a[i] / f[i]         ... = ...
```

Code 5.2: Recomputation across two threads.

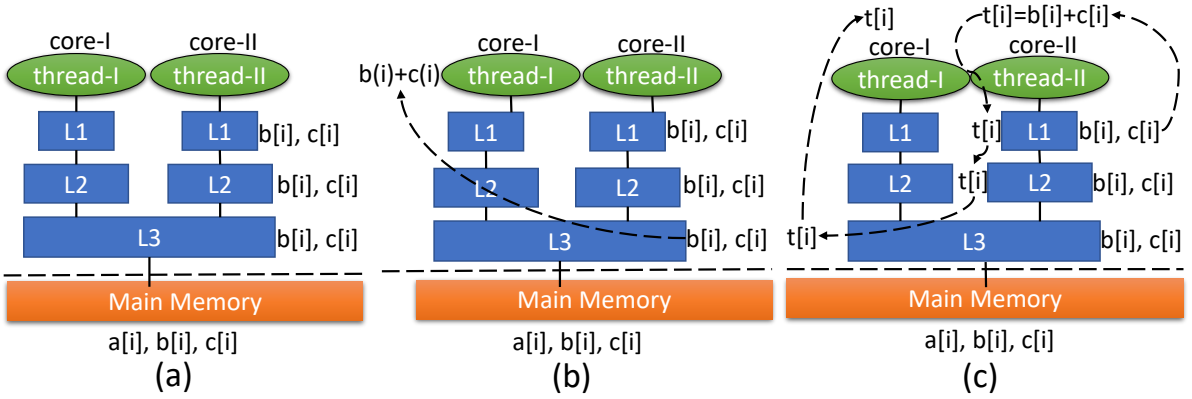


Figure 5.2: Data locations and computations for REOT (b) and RBOT (c), assuming the initial state shown in (a).

5.3.3 Recomputation–Data Locality Tradeoffs

While the examples discussed above highlight the potential opportunities brought by recomputation in single-threaded and multi-threaded cases, it may not be a good idea to take advantage of every recomputation opportunity presented by the program code. In particular, there exist cases where data locality optimization and recomputation optimization can conflict. To illustrate, let us consider the code in Code 5.3, assigned to a thread in a multithreaded application. While this scenario is similar to the abovementioned scenario, there are further reuses of $a[i]$ following line 7. As a result, directly accessing $a[i]$ in line 7 (as opposed to recomputing its value using $b[i]$ and $c[i]$) can be a better option here as doing so would also enable fast (probably L1) accesses to $a[i]$ in lines 8 and 9. However, to make such decisions, one needs to consider the big picture, which involves all accesses to the data element for which we are considering recomputation ($a[i]$ in the example) as well as to those that will be used in recomputation ([b] and $c[i]$), as the best option could be exploiting data locality in some cases (by directly accessing $a[i]$) and exploiting recomputation in others (by using $b[i]$ and $c[i]$).

Furthermore, even if lines 8 and 9 do not exist in the code of this thread, they may exist in the code of some other thread. In such a scenario, considering the performance of *both* the threads together, it may be more profitable to skip recomputation and bring $a[i]$ into the cache hierarchy so that the other thread can access it from some shared

```
1  for i := 1, N
2    a[i] = b[i] + c [i]
3    ...  = ...   # accesses to other arrays
4    ...  = ...   # accesses to other arrays
5    ...  = ...   # accesses to other arrays
6    ...  = b[i] - c[i]
7    ...  = a[i] / e[i]
8    ...  = a[i] * (a[i] + d[i]) / 2
9    ...  = a[i] - 3f[i]
10 endfor i
```

Code 5.3: Conflict between data locality and recomputation optimizations.

layer in the cache hierarchy. Performing such recomputation-data locality tradeoffs can be challenging in the case of multithreading and is further discussed in Section 5.4.

5.3.4 Recomputation-Conscious Cache Replacement

One can envision further benefits from data recomputation by modifying the existing cache replacement policies. Current cache replacement policies are generally based on the idea of least recently used (LRU), which is based on discarding, from the cache, the least recently used data element first to make room for the new data element to be brought in. Since implementing the exact LRU in hardware can be costly, most modern architectures employ an "approximate" version. While LRU has proven to be quite successful in managing limited cache space for diverse programs and most existing compilers are tuned to exclusively work with the LRU policy, in a compilation flow that employs data recomputation (like ours), we may want to change it. We consider the code fragment in Code 5.4 to demonstrate the need for a new replacement policy.

```
1  for i := 1, N
2    a[i] = b[i] + c[i]
3    d[i] = b[i] - c[i]
4    e[i] = b[i] x c[i]
5    ...  = ...   # accesses to other arrays
6    ...  = ...   # accesses to other arrays
7    ...  = ...   # accesses to other arrays
8    ...  = b[i] + 1
9    ...  = b[i] - 1
10   ...  = a[i] + d[i] + e[i]
11   ...  = a[i]
12 endfor i
```

Code 5.4: Example showing the need for a new cache replacement policy.

At the beginning of this code fragment, $a[i]$, $d[i]$, and $e[i]$ are computed, and later towards the end of the loop body, they are reused. When the execution reaches line 10, if both $b[i]$ and $c[i]$ are in the cache closest to the core executing this thread, we can recompute the values of $a[i]$, $d[i]$ and $e[i]$ (using $b[i]$ and $c[i]$) instead of directly accessing them from far away caches or main memory (assuming that they are displaced from the L1 cache due to the accesses to other data elements in lines 5-9). However, $c[i]$ may also be replaced from the L1 cache as lines 5 through 10 do not access it. By adopting a

cache replacement policy that *prevents* $c[i]$ from being displaced from the cache until the execution reaches at least line 11, we can still entertain the recomputation option. Hence, the general idea would be to ensure that the data elements that would be used in a future recomputation of other data elements must be kept in the cache till the recomputation takes place, even if doing so goes against the LRU policy.

Consider the code segments from two threads in Code 5.5. Thread-II would benefit, when it comes to line 9 of its code, from $c[i]$ residing in the cache hierarchy (i.e., not replaced); so, a cache replacement policy that would keep this data element in a cache layer shared by the two cores that run these two threads would be very desirable from a recomputation viewpoint (even if we allow it to be replaced from the private cache of the core running thread-I). In Section 5.4.2, we give our algorithm that issues pin/unpin "hints" to the cache to increase recomputation opportunities (via data element pinning in the cache) in multi-threaded application programs, and in Section 5.5, we evaluate it.

```
1  thread-I:                thread-II:
2
3  a[i] = b[i] + c[i]       ...  = ...
4  d[i] = b[i] - c[i]       ...  = ...
5  e[i] = b[i] x c[i]       ...  = ...
6  ...  = ...               ...  = ...
7  ...  = ...               ...  = b[i] + f[i]
8  ...  = ...               ...  = b[i] - g[i]
9  ...  = ...               ...  = a[i] + d[i] + e[i]
```

Code 5.5: Potential benefits of recomputation-conscious caching across threads.

5.4 Compiler Algorithms for Data Recomputation

Our compiler algorithms are glued together as a single "pass" ("phase"). Our pass is invoked after coarse-grain code parallelization to distribute computations/iterations across available cores (unless the code has already been hand-parallelized) and conventional data locality optimizations employed by commercial optimizing compilers such as loop permutation, iteration space tiling, data prefetching, and loop unrolling, as well as data layout optimizations such as array padding [60]. Consequently, it starts with an *already-optimized* program code from *both* the coarse-grain parallelism and data locality perspectives. On the other hand, low-level code optimizations (including fine-grain parallelism/SIMD optimizations) are carried out after our pass. If desired, select data locality optimizations can be applied again following our pass. Our pass takes, as input, the program code to be optimized and a "description" of the target manycore architecture for which the input code is to be compiled. This description includes but is not limited to, the number/types of cores, details of cache hierarchy and cache policies, details of

the underlying on-chip network and its routing policy, and details of the memory request scheduling policies.

5.4.1 Identifying Recomputation Opportunities

Recall from Section 5.2 that an array reference within a loop nest is represented by $\Omega\vec{I} + \omega$. We use $S_k(\vec{I})$ to denote the "instance" of an assignment statement S_k scheduled to be executed in iteration $\vec{I}$ in a loop nest; here, we have $1 \leq k \leq \kappa$, where κ is the total number of assignment statements in the loop body. We sometimes use the notation $S_k(\vec{I}) \in t_p$ to indicate that $S_k(\vec{I})$ is assigned to thread t_p (e.g., as a result of code/loop parallelization).

For a given loop iteration $\vec{I}$ and assignment statement S_k, $\alpha(S_k(\vec{I}))$ gives us the left-hand-side (LHS) reference (i.e., the reference to be updated/re-written by the assignment statement) and $\beta(S_k(\vec{I}), h)$ gives us the h^{th} reference on the right-hand-side (RHS). As an example, in a statement like $S_1(\vec{I}) : a[i_1][i_2] = a[i_1 - 1][i_2 + 1] + b[i_1][i_2]$ (assuming $\vec{I} = (i_1\ i_2)^T$), we have $\alpha(S_k(\vec{I})) = a[i_1][i_2]$, $\beta(S_k(\vec{I}), 1) = a[i_1 - 1][i_2 + 1]$, and $\beta(S_k(\vec{I}), 2) = b[i_1][i_2]$. When no confusion occurs, we use $\{\beta(S_k(\vec{I})\}$ to denote all references on the RHS of statement $S_k(\vec{I})$. Sometimes, we use S_k without mentioning any iteration vector or thread if they are clear from the context.

5.4.1.1 Legality of Data Recomputation

As observed by prior research [3, 193], recomputation is not always applicable. Prior work focuses mainly on single-threaded centric optimizations and hence misses important opportunities that could be enabled by considering multiple threads and their interactions, e.g., data sharing. Consider the code fragment in Code 5.6 scheduled to be executed by thread t_1. In this example, we cannot use recomputation for $a[i]$ in line 6 since $c[i]$ got updated in line 4 (as a result, $b[i] + c[i]$ would generate a different result). Even if the assignment in line 4 was not there, some other thread (say, t_2) can have a similar assignment (i.e., $c[i] = ...$), which would still prevent recomputation of $a[i]$ in line 6 of the code of thread t_1. Thus, to be able to employ recomputation in a multithreaded execution, we need to ensure either (i) such assignments do not exist or (ii) they are enforced not to execute before the desired recomputation takes place. Formulating these constraints can be challenging in multithreaded execution and will be discussed later.

Figure 5.3 depicts recomputation opportunities in a *multithreaded code*. We begin by assuming that statement $S_1(\vec{L})$, which is $a[] = b[] + c[]$ is scheduled to be executed by

```
for i := 1, N
  a[i] = b[i] + c[i]
  ...  = ...   # accesses to other arrays
  c[i] = ...   # accesses to other arrays
  ...  = b[i] - c[i]
  ...  = a[i] / e[i]
endfor i
```

Code 5.6: An example showing that it is not always possible to employ data recomputation.

thread t_n. Suppose, later in execution, a statement $S_2(\vec{I})$, of the form $d[] = a[] + 1$, is to be executed by thread t_p. We want to check whether it is possible to replace the direct $a[]$ access in $S_2(\vec{I})$ of t_p with a recomputed version in $S_{extra}(\vec{J})$: $d[] = b[] + c[] + 1$ in thread t_q, i.e., we would like to replace $a[]$ by $b[] + c[]$ and execute the resulting statement $S_{extra}(\vec{J})$ by thread t_q (instead of thread t_p executing $S_2(\vec{I})$). Some of the involved threads in this process may be the same; in the extreme case, we have $t_p = t_n = t_q$.

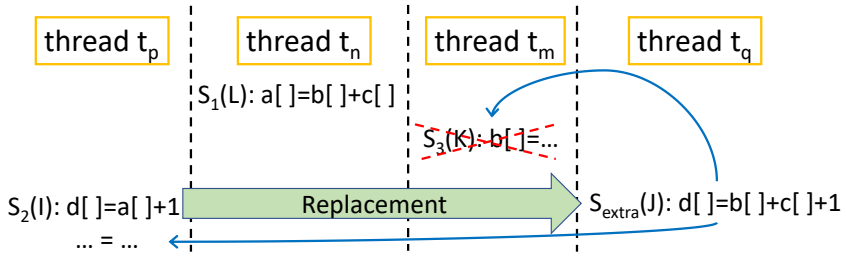


Figure 5.3: Recomputation opportunities in a multithreaded scenario. Blue arrows denote point-to-point (thread-to-thread) synchronization.

For such a replacement of direct data access with its "recomputed value", there must not be any intervening statement $S_3(\vec{K})$, where $\vec{L}<\vec{K}<\vec{J}$, to be executed by thread t_m, updating one of the data elements (e.g., $b[]$) to be used in the recomputation of $a[]$. We want to point out that t_m can be the same as any of t_p, t_n, or t_q. We set `rcmp?`$(< S_2(\vec{I}), t_p, R >, < S_{extra}(\vec{J}), t_q >)$ to "true" if the following conditions are satisfied:

1. $\exists$ s_1, $\vec{L}$, and n such that $s_1(L) \in t_n$ and $\alpha(s_1(\vec{L})) = R$ and $\vec{L} < \vec{I}$, and
2. !$\exists$ s_3, $\vec{K}$, and m such that $\alpha(s_1(\vec{L})) \in \beta(S_2(\vec{I}))$ and $S_2(\vec{L}) \in t_m$ and $\vec{L} < \vec{K} < \vec{J}$.

The first condition conveys that there must be an assignment to reference R ($a[]$ in the example in Figure 5.3) previously in execution. The second condition indicates that there must *not* be an intervening assignment, by any thread, to any of the operands/data elements that will be used in the recomputation of R. For this second constraint, if there is no such assignment to any operand for recomputing R, nothing extra needs to be done,

and we insert S_{extra}. If such an assignment/update exists, we still insert S_{extra} to ensure that it is executed before S_3, and we insert a synchronization between S_{extra} and S_3. Similarly, we need to ensure that any statement that follows S_2 must be executed after S_{extra}, and a compiler-inserted synchronization enforces this. These two compiler-inserted thread-to-thread synchronizations, needed to ensure correct execution for recomputation-optimized code, are shown using blue arrows in Figure 5.3. The second synchronization can be eliminated if t_q is the same as t_p. We insert S_{extra} in t_p whenever possible. In many cases, this may not be preferable from the overall performance viewpoint, as t_q can be closer to the data elements that are needed to recompute the value of the target reference (see the examples in parts (c) and (d) of Figure 5.1). In general, as far as the legality of recomputation of a given reference R is concerned, one can identify all threads t_q that can legally accommodate $S_{extra}(\vec{J})$.

5.4.1.2 Benefits of Data Recomputation

While the formulation above captures the constraints that must be satisfied for the legality of recomputation (i.e., for `rcmp?`$(< S_2(\vec{I}), t_p, R >, < S_{extra}(\vec{J}), t_q >)$ to be true), it does not say anything about whether employing it would be beneficial from a performance viewpoint.

For a given array reference $R(\vec{I})$ in the code (where $R(\vec{I})$=$\Omega\vec{I} + \omega$), we define $\gamma(R(\vec{I})$ as the "estimated cost" of accessing it. This estimation is performed at compile-time based on the "predicted location" of $R(\vec{I})$ at run-time. To do this estimate, we use prior work, called cache miss equations [43], which predicts whether a given data access/array reference will hit or miss in a given cache. Once this prediction is made for each layer in a cache hierarchy, we can estimate the "base latency" of the access. For instance, if the access $(R(\vec{I})$ is predicted to miss in the L1 cache but hit in the L2 cache, we estimate $\gamma(R\vec{I})$ as the access latency of the L2 cache. However, this base latency does *not* consider the specific location of the data. For example, in a modern on-chip network based manycore architecture such as Intel's Sky Lake, the cost (latency) of a last-level cache (LLC) access is a function of not only the cache parameters but also the *distance* (in the on-chip network) between the core issuing the data request and the location of the LLC bank that holds the requested data.[2] We extended the original CME formulation in [43] by i) taking into account an entire (L1-L2-L3) cache hierarchy of a manycore system, ii) considering conflict and coherence misses in addition to cold and capacity

[2]For example, there are typically multiple L3 banks distributed across different nodes, and the latency is a function of the distance between the requesting core/node and the L3 bank that holds the data.

misses, and iii) predicting on-chip network latency based on the distance between the core and the target LLC bank. Our latency predictor is accurate; its estimate is always within ±10% of the actual latency.

We define the following gain function that quantifies the benefit/profitability of replacing a given reference R appearing in $S_2(\vec{I})$ of thread t_p by the RHS of $S_1(\vec{L})$ (originally appeared in thread t_n) by creating a new statement $S_{extra}(\vec{J})$ in thread t_q, i.e., via recomputation:[3]

$$gain(< S_2(\vec{I}), t_p, R >, < S_{extra}(\vec{J}), t_q >) = \\ \gamma(R(\vec{I}) - \left(\left(\sum_{x \in \beta(S_{extra}(\vec{J}))} \gamma(x)\right) + \pi(\beta(S_{extra}(\vec{J}))) + \sigma(t_q, t_p) + \sigma(t_q, t_m)\right),$$

where $\pi(\beta(S_{extra}(\vec{J})))$ is the "computational cost" of recomputing the right-hand side of $S_1(\vec{L})$ in $S_{extra}(\vec{J})$ in thread t_q; $\sigma(t_q, t_p)$ is the "synchronization cost" between threads t_q and t_p; and $\sigma(t_q, t_m)$ is the "synchronization cost" between threads t_q and t_m. These synchronization costs may or may not materialize in specific cases, depending on where the recomputation takes place and on behalf of which thread it is performed.

5.4.1.3 Code Flattening

The above discussion focuses on one reference and discusses the potential benefits of its recomputation. However, even the references used in recomputing a given reference R can be recomputed. Consider the code fragment in Code 5.7. In line 8, the code accesses $d[i]$, which can be recomputed as $e[i] - a[i]$ if it is beneficial. This recomputation may return a negative gain, assuming that $a[i]$ was not accessed in lines 3 and 8 and was removed from the cache. We can go one step further and recompute $a[i]$'s value to be used in the recomputation of the value of $d[i]$ as $b[i] + c[i]$, noting that both $b[i]$ and $c[i]$ are accessed in line 7. They will likely be in the caches close to the core. We can recompute the value of $d[i]$ using $e[i] - (b[i] + c[i])$, instead of as $e[i] - a[i]$. While this example focuses on a "two-deep" recomputation, we can apply this idea hierarchically (in a chained fashion) for all references involved in the original recomputation, obtaining a "recomputation tree". Although recomputing $d[i]$ using $a[i]$ would serve the purpose, going deeper by recomputing $a[i]$ would still be beneficial when $b[i]$, $c[i]$, and $e[i]$ are in the L1 cache, $a[i]$ is in the L2 cache, and $d[i]$ is in the L3 cache.

In the above expression, a positive gain function value indicates a "benefit"/"profit"

[3]Assuming that `rcmp?`$(< S_2(\vec{I}), t_p, R >, < S_{extra}(\vec{J}), t_q >)$ returns true; we consider the benefits/profitability of a recomputation *only if* the recomputation passes the legality test.

```
1 for i := 1, N
2   a[i] = b[i] + c[i]
3   d[i] = e[i] - a[i]
4   ...  = ...   # accesses to other arrays
5   ...  = ...   # accesses to other arrays
6   ...  = e[i] + 1
7   ...  = b[i] + c[i]
8  ...  = d[i] x f[i]
9 endfor i
```

Code 5.7: An example illustrating the possibility of chained recomputations.

from the recomputation being considered, whereas a non-positive value means no benefit. What this gain formulation tells us is that recomputation is "beneficial" if the cost of directly accessing R is higher than the cost of recomputing it – by performing the needed synchronizations, accessing all right-hand side references, and performing computations, i.e., including *all* the overheads that come with the recomputation. While our recomputation legality and benefit analysis has focused on a single reference (R), it can be repeated for each reference for which recomputation is considered. For instance, if two references are considered, we can apply the above analysis to each one by one and select the best option (latency-wise). The next section discusses data recomputation in the context of an entire program segment.

It is better not to use recomputation in some cases (even if it is legal and beneficial– when considered in isolation–to do so) as it may conflict with data locality (cf. Section 5.3). For example, in Code 5.8, if there were repeated uses of $d[i]$ following line 8, one would choose not to recompute it and instead access it directly from the costly location. This is because this costly access would occur only once, and all subsequent reuses would find $d[i]$ in cache locations closer to the core accessing it.

```
1 thread-I:                thread-II:
2
3 d[i] = g[i] + h[i]        ... = ...
4 e[i] = l[i] + k[i]        ... = ...
5 c[i] = d[i] + e[i]       l(i) = ...
6 ... = ...                 [i] = b[i] + c[i]
7 ... = ...                 ... = ...
8 ... = ...                 ... = a[i]
9
```

Code 5.8: Possibility of chained recomputations.

Unfortunately, both aspects— the possibility of applying recomputation hierarchically and interaction with data locality—make it challenging to develop an analytical formulation capturing all benefits and all costs. Motivated by these observations, we adopt a strategy called "Code Flattening." In this strategy, the code is progressively *flattened* via repeated recomputations until all options are checked for the best-performing one. Let us focus on Code 5.8, which illustrates hierarchical/chained recomputations for reference $a[i]$ that appears in line 8 of thread-II's code.

Figure 5.4(a) depicts the "recomputation tree" for this example, and Figure 5.4(b) lists the corresponding potential unfolded recomputations. Note that we cannot recompute $e[i]$, which is assigned by thread-I, as one of the operands to be used in its recomputation ($l[i]$) is updated in line 5 of thread-II. This process can be viewed as progressive code flattening until no further flattening is possible.

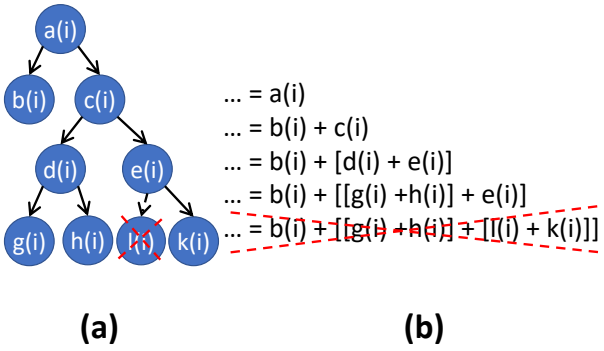


Figure 5.4: Example recomputation tree and corresponding potential recomputations. (a) Recomputation tree for reference $a[i]$ in Code 5.7 and (b) corresponding potential recomputations.

This second point also means that, while replacing R in $S_2(\vec{I})$ of thread t_p by its "recomputed version" $S_{extra}(\vec{J})$ in thread t_q, we evaluate the value of the gain function (and determine if it is profitable to do so), such a replacement (if found profitable) can also have other impacts that influence the behavior of the application code, e.g., (i) it can cause the execution to miss data reuse opportunities if there are subsequent accesses to R (after recomputation is performed), and (ii) by not accessing R we are also not bringing the data elements that reside in the same line (block) as R, into the cache. As a result, although a positive gain function value indicates a "potential benefit" from the replacement, i.e., using the recomputed version (local view), whether such benefit translates to a "program-wide" saving (global view) may not be evident. Motivated by this observation, our approach, after replacing the reference with its recomputed version, invokes a "cost function", which estimates the total latency of the entire program (using γ functions for data accesses, π functions for computations, and σ functions for synchronizations). If the recomputation leads to an overall (program-wide) saving (when including all the costs it entails), it is committed, and then the algorithm explores the next recomputation opportunity.

Algorithm 5.1 gives our compiler algorithm for employing recomputation in multi-threaded codes. Here, T is the total number of threads, Σ is the total number of assignment statements in the code, and Π is the total number of references considered for

recomputation. At any point in the execution of this compiler algorithm, *curver* holds the current "minimum-cost" code version, and *mincost* is its cost (estimated latency). Within the innermost loop of this algorithm, we try to find a legal and profitable recomputation and execute it. The algorithm continues as long as a reference exists for which recomputation could be beneficial (i.e., *expand* is 'true').

Algorithm 5.1 Compiler for employing recomputation in multi-threaded codes.

```
curver ← input program
mincost ← cost(curver)
while expand = 'true' do
    for t_p ← 1 to T do
        for S_2 ← 1 to Σ do
            for R ← 1 to Π do
                for t_q ← 1 to T do
                    expand ← 'false'
                    if ∃I⃗, J⃗ and rcmp?(< S_2(I⃗), t_p, R >, < S_extra(J⃗), t_q >) and
                    gain(< S_2(I⃗), t_p, R >, < S_extra(J⃗), t_q >) > 0 then
                        ▷ Expand curver by adding a recomputed version of S_2 as S_extra into t_q
                        S_extra ← recompute(S_2)
                        curver' ← S_extra in curver in place of S_2
                        if cost(curver') < mincost then
                            curver ← curver'
                            mincost ← cost(curver)
                            expand ← 'true'
                        end
                    end
                end
            end
        end
    end
end
```

5.4.1.4 Discussion of Algorithm 5.1

First, this algorithm expands the recomputation tree at each iteration (see Figure 5.4(a) for an example). Second, although the algorithm above implies that, at each iteration of the while loop, the threads and statements are always traversed/checked in the same order, in our actual implementation, we use a "rotating traversal", e.g., while in the first invocation of the loop that goes over the threads, we start with the first thread, in the following invocation we start with the second thread, and so on. Consequently, the recomputation opportunities are checked across threads (as well as statements) in a breadth-first manner. Third, to check the legality and profitability of recomputation in thread t_p, our approach checks all threads t_q where $1 \leq t_q \leq T$ (total number of threads in the application). This is because, while the gain function may not return a positive value for a given t_x, it may return a positive value for another thread t_y where $1 \leq t_x \neq t_y \leq T$. This approach is one of the significant ways in which recomputation in

a multi-threaded context differs from recomputation in a single-threaded context, as in the latter case, we always have $t_p = t_q$.

5.4.2 Compiler-Guided Cache Management for Recomputation

In this section, we propose a novel caching strategy to increase the effectiveness of data recomputation. We use recomputed value instead of performing the original data access if $gain(< S_2(\vec{I}), t_p, R >, < S_{extra}(\vec{J}), t_q >) > 0$, and the latency of the original access R as well as the accesses in $\{\beta(S_{extra}(\vec{J}))\}$ are estimated using the γ functions. Without loss of generality, let us assume that the accesses on the right-hand side $\{\beta(S_{extra}(\vec{J}))\}$ are Q_1, Q_2, ..., Q_M (M being the total number of accesses). Any particular Q_i or a subset of Q_1, Q_2, ..., Q_M can be responsible for $gain(< S_2(\vec{I}), t_p, R >, < S_{extra}(\vec{J}), t_q >) \leq 0$, thus ending up not performing recomputation. Motivated by this observation, our approach in this section checks the elements on the right-hand side, namely, Q_1, Q_2, ..., Q_M, and pins in the cache(s), the element(s) that leads to $gain(< S_2(\vec{I}), t_p, R >, < S_{extra}(\vec{J}), t_q >) > 0$. The salient aspects of our approach are as follows:

- We provide novel hardware support, namely, two new instructions pin and unpin to provide hints to the cache(s):
  - When executed, pin Q_i, L_j pins block (line) that contains data reference Q_i in cache L_j, where $j = 1$, 2 or 3, until it is explicitly unpinned. Note that if Q_i is not in L_j after executing this instruction, it is brought into L_j. Moreover, once a line is pinned in a cache, it is not displaced from that cache (i.e., excluded from LRU-based replacement) until explicitly unpinned.
  - When executed, unpin Q_i, L_j unpins block (line) that contains data reference Q_i from cache L_j, where $j = 1$, 2, or 3. After executing this instruction, the cache line is not immediately displaced from the cache; it becomes a potential replacement candidate, subject to underlying LRU policy.
- Everything else being equal, we want to minimize the number of data elements pinned in cache(s) in the manycore system. This is because each such pinning is a kind of "disturbance" to the LRU-based replacement policy and can lead to degradation in cache performance. Furthermore, we apply pinning *only if* Algorithm 5.1 cannot find any recomputation opportunity for a given reference.

In this new cache management implementation, we *pin* a data element during the execution of the recomputation-insertion algorithm (Algorithm 5.1) and *mark* the instruction that will use the pinned data element (in implementing recomputation) and *unpin* the element

right after the marked instruction has completed its execution.

Algorithm 5.2 gives a modified version of Algorithm 5.1, which accommodates "data element pinning" and "instruction marking" ("unpinning" is not shown). For each reference R that we cannot find a profitable recomputation, Algorithm 5.2 constructs a list of "potential pinnings", selects the minimum-cost one among them, and marks the associated statement. It uses a cache-pinned element to mark the S_{extra} program statement. This marking is *propagated* from the program statement to individual assembly instructions during the instruction selection phase in the compiler's back-end. This "marking-propagation" process is highly mechanical.

5.5 Experimental Evaluation

Setup: To evaluate the effectiveness of our approach, we used programs from the SPEC OMP2012 benchmark suite [137]. We used the system Intel *mobile* Core i7-6700HQ manycore CPU in our experiments. In our default setting, each application program has been executed using 16 threads (one thread per node). We later changed the number of threads used for execution to conduct a sensitivity study. In implementing our approach, we used LLVM version 14.0.0 [98] as a source-to-source translator (Clang); the resulting optimized sources are compiled using Intel's Fortran Compiler Classic 2021.5.0 and Intel's C++ Compiler Classic 19.1. The above setting is used to evaluate Algorithm 5.1. We used a simulation-based environment based on the gem5 simulator [16] infrastructure to evaluate Algorithm 5.2, modeling our Intel Sky Lake system as accurately as possible.

5.5.1 Evaluation of Algorithm 5.1

Figure 5.5 presents two graphs quantifying the metrics to show how effectively the compiler identifies recomputation opportunities. The first metric, *legality*, gives the fraction of times the `rcmp?` function returns true, i.e., it is legitimate to perform data recomputation for the targeted reference. The second metric, *benefit*, captures the fraction of times the gain function returns a positive number, indicating profitability for data recomputation (when considering only the cases `rcmp?` returns true). We report three numbers for each application program in our benchmark suite for each metric. The first number is when only intra-thread recomputation opportunities are taken advantage of (though the execution is still multithreaded), i.e., no recomputation opportunities across threads are considered. The second and third numbers represent REOT and RBOT, as described in Section 5.3. Note that REOT+RBOT represents the potential of

Algorithm 5.2 Modified compiler for employing recomputation in multi-threaded codes, which accommodates "data element pinning" and "instruction marking".

```
curver ← input program
mincost ← cost(curver)
potentials ← ∅
while expand = 'true' do
    for t_p ← 1 to T do
        for S_2 ← 1 to Σ do
            for R ← 1 to Π do
                for t_q ← 1 to T do
                    expand ← 'false'
                    if ∃I⃗, J⃗ and rcmp?(< S_2(I⃗), t_p, R >, < S_extra(J⃗), t_q >) and
                    gain(< S_2(I⃗), t_p, R >, < S_extra(J⃗), t_q >) > 0 then
                        ▷ Expand curver: add recomputed S_2 as S_extra to t_q
                        S_extra ← recompute(S_2)
                        curver' ← S_extra in curver in place of S_2
                        if cost(curver') < mincost then
                            curver ← curver'
                            mincost ← cost(curver)
                            expand ← 'true'
                        end
                    end
                    else
                        ▷Find minimum subset of {Q_i} ∈ {β(S_extra(J⃗))} and corresponding caches {L_j} such that
                        gain(< S_2(I⃗), t_p, R >, < S_extra(J⃗), t_q >) > 0
                        potentials ← potentials ⋃ < S_extra(J⃗), t_q, {Q_i} >
                    end
                end
                if !(expand) then
                    ζ ← minimum-cost-one(potentials)
                    ▷ Expand curver: add recomputed S_2 as S_extra (of ζ) to t_q
                    pin {Q_i} in caches {L_j}
                    mark statement S_extra for pinning
                end
            end
        end
    end
end
```

"inter-thread" data recomputation.

We can make several observations from Figure 5.5. First, the legality metric (Figure 5.5(a)) averages[4] around 11.2%, 12.1%, and 13.7% for intra-thread, REOT, and RBOT, respectively, indicating that there are many cases in which intra-thread data recomputation would be illegal. We can find a legal data recomputation when other threads are considered. Second, we observe a similar trend when looking at the results of the benefit metric Figure 5.5(b)). Specifically, when considering *only the cases where data recomputation is legal* (i.e., `rcmp?` returns true), data recomputation can lead to potential benefits of approximately 20.4%, 16.9%, and 18.9% for intra-thread, REOT, and RBOT, respectively. These results underline the importance (and feasibility) of recomputation across threads. Third, the fraction of legal and beneficial recomputations

[4]In this section, the term "average" refers to "geometric mean."

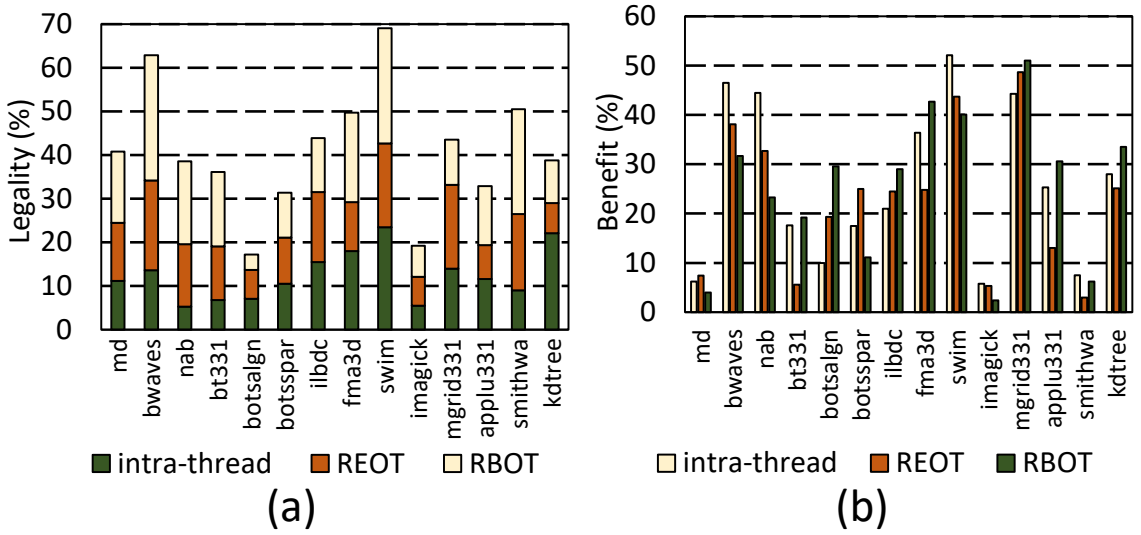


Figure 5.5: Legality and benefit evaluation of Algorithm 5.1.

change from one benchmark to another. For example, in benchmarks like `swim`, we see a significant opportunity and potential saving, whereas, in `smithwa`, which has a large legality percentage, the potential benefits are much lower. This behavior is primarily because, in `smithwa`, data locality in upper-level caches (L1 and L2) is not good, making recomputation more costly, in many cases, than directly accessing the target reference. On the other hand, in applications such as `bothsalgn` and `imagick`, most recomputation opportunities are not legal because of the frequent updates to data elements that would be needed in recomputation. Such updates are challenging for Algorithm 5.1.

Figures 5.6 and 5.7 plot the execution time improvements brought by Algorithm 5.1 compared to the original applications on real hardware and on the simulator, respectively. The original codes against which we compare our recomputation-based versions have already been heavily optimized from data locality and parallelism (SIMD) perspectives using the highest compilation flag. It can be seen from these results that data recomputation brings 13.6% performance savings when averaged over all 14 programs tested. Thus, we can conclude that our compiler-based approach to data recomputation is very successful in practice.

5.5.2 Evaluation of Algorithm 5.2

As shown in Figure 5.8, Algorithm 5.2 improves execution cycles by 19.1%, on average, over the original executions (improvements range between 5.5% and 34.2%), over the original applications. Figure 5.9 shows the recomputation benefits of Algorithm 5.2

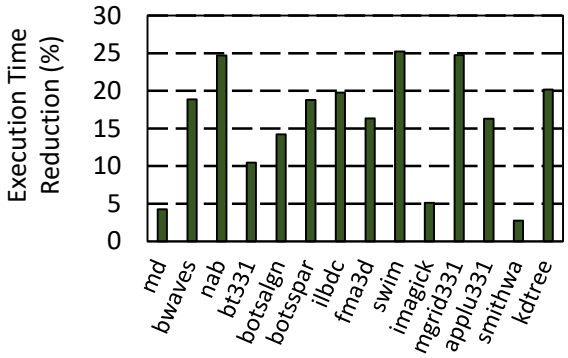


Figure 5.6: Execution time improvements brought by Algorithm 5.1 (real hardware).

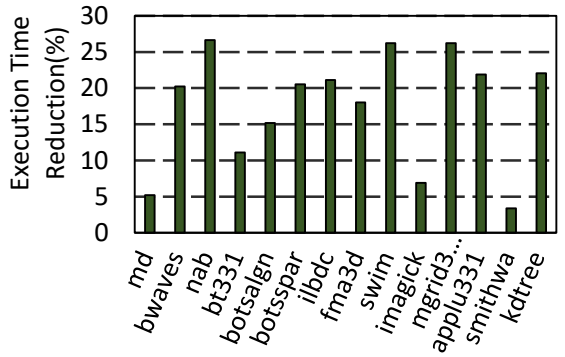


Figure 5.7: Execution time improvements brought by Algorithm 5.1 (simulator).

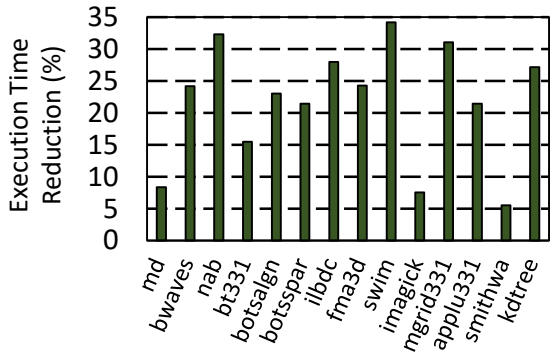


Figure 5.8: Execution time improvements brought by Algorithm 5.2 (simulator).

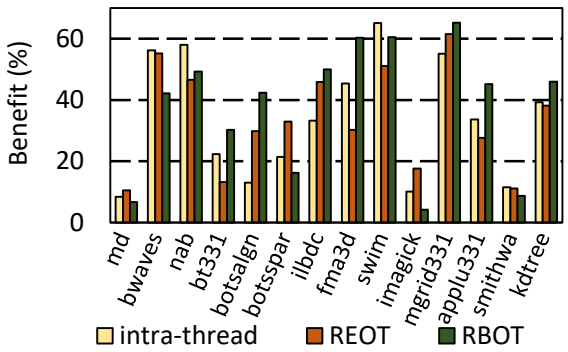


Figure 5.9: Quantification of data recomputation benefits when using Algorithm 5.2.

when using our simulator (as data pinning/unpinning in caches cannot be tested on real hardware). Note that the legality results of Algorithms 5.1 and 5.2 are the same. As seen from these results, data pinning improves the benefit metric significantly by 27.5%, 29.1%, and 28.8% for intra-thread, REOT, and RBOT, respectively. On the other hand, the additional compilation time increase brought by Algorithm 5.2 over Algorithm 5.1 was around 10%, when averaged across all benchmark programs.

While the results in Figure 5.6 are collected from real hardware, those in Figure 5.8 are collected from the simulator. To check whether the simulator accurately models the manycore hardware used in our experiments, we collected the execution time savings brought by Algorithm 5.1 using the simulator as well. The results in Figure 5.8 indicate an average execution time reduction of 14.9%. Comparing these results to those in Figure 5.6 for Algorithm 5.1 (with an average improvement of 13.6%) reveals that the simulator results follow the actual hardware experiments very closely. Figure 5.10 shows

the distribution of the benefits from REOT and RBOT across all 16 threads (the results represent average values over all benchmarks). The benefits are distributed across the threads in a balanced fashion, meaning each thread takes advantage of recomputation.

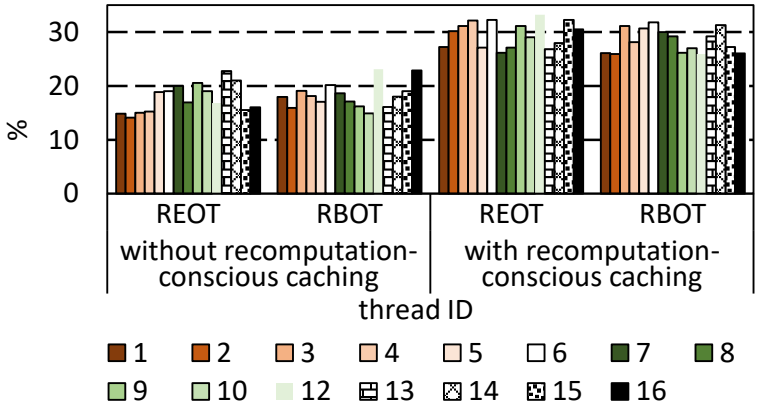


Figure 5.10: Distribution of the benefits coming from REOT and RBOT.

5.5.3 Sensitivity to the Number of Threads

Recall that the results presented and discussed have been collected using 16 threads. We also collected results with varying thread counts for the cases with and without recomputation-conscious caching. We observed that when the thread count varies from 1 to 28, more and more benefits come from REOT and RBOT (in the extreme case of 1 thread [sequential execution], all recomputation benefits are from intra-thread). The corresponding execution time savings with different thread counts are plotted in Figure 5.11. Each point on these plots represents an improvement over the original code under the *same thread count* when averaged (geometric mean) across all 16 applications. Our approach's performance improvements remain stable across different thread counts.

5.5.4 Quantitative Comparison against Related Work

The most closely related work is CND (computation with near data) [193]. CND reduces data movement costs for single- and multi-threaded applications using the "recomputation" concept, where costly data access is replaced by a few less costly data accesses and extra computation if such replacement is cost-efficient. Compared to ours, this work focuses only on single-threaded execution and does not consider recomputation-aware caching. Figure 5.12 depicts the results collected using a 16-thread execution, with intra-thread

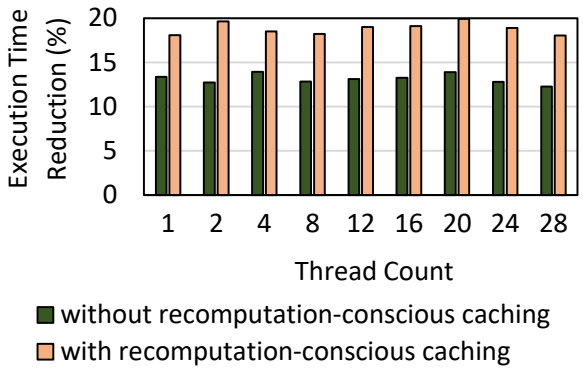


Figure 5.11: Execution time improvements different thread counts.

alone, intra-thread + REOT, and intra-thread + REOT + RBOT, where the incremental contribution of each aspect on execution time can be seen. Our main observation from Figure 5.12 is that the savings achieved by CND and our intra-thread are similar as they catch similar recomputation opportunities. However, REOT and RBOT have a significant impact, demonstrating the importance of exploiting data recomputation opportunities *across threads* (in addition to intra-thread recomputation opportunities).

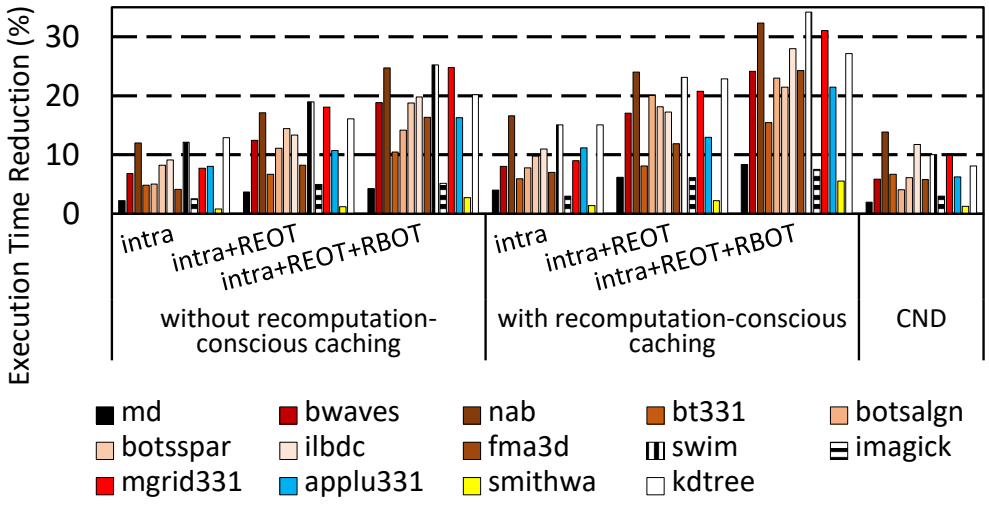


Figure 5.12: Execution time savings for different policies (16 threads).

5.6 Discussion of Related Work

In [3], the proposed idea is to replace a load with a sequence of instructions (called "recomputation slices") to recompute the loaded data value. This work requires architectural support. Akturk and Karpuzcu [4] analyze the computation vs. communication

tradeoff when recomputation replaces data load(s) from memory with reproducing that data. Sakalis et al. [168] use recomputation of data that resides in off-chip memory in isolation. In none of these works, threads recompute on behalf of other threads. On the compiler side, Koc et al. [92] propose improving the energy effectiveness of a banked memory architecture by recomputing the data required from a bank and avoiding unnecessary bank activations. Later, Koc et al. [91] propose an algorithm to eliminate off-chip memory accesses to Scratch-Pad Memory (SPM) with recomputation. Briggs et al. [19] offer a solution to the problem that arises when a value cannot be kept in a register during global register allocation (*rematerialization*), which allows the allocator to recognize when it is cheaper to recompute the value of a loaded data if it cannot be kept in a register to avoid expensive storing and reloading. Punjani [153] shows that, in register-starved architectures, spilling values that can be "rematerialized" causes a loss in performance, so rematerialization aids in the efficient usage of registers. Tang et al. [193] present Computing with Near Data (CND), which has been compared against in Section 5.5.

5.7 Chapter Summary

This chapter proposes a novel recomputation paradigm that (i) exploits recomputation opportunities not only within each thread but also across the threads of a multi-threaded program and (ii) pins select data elements in various caches in memory hierarchy if doing so helps us increase intra-and inter-thread data recomputation opportunities. Our experiments reveal that the proposed strategy that performs data recomputation across threads improves performance by an average of 13.25% over the conventional optimizations that do not use recomputation and 7.68% over a single-thread centric recomputation. The corresponding improvements when employing recomputation-conscious caching are 19.12% and 11.63%, respectively.

Chapter 6
Performance Optimization Using On-the-Fly Architectural Resource Reconfiguration

6.1 Introduction

In our first study (see Chapter 3, we propose an ML model that *statically* characterizes applications from different domains to perform bottleneck analysis for personalized hardware. However, different portions of different applications may have different microarchitectural characteristics and bottlenecks. Moreover, as HPC centers leverage renewable energy resources such as solar energy, energy harvesting processors emerge and require better (i.e., dynamic and fine-grained) resource allocation and configuration. To meet this requirement, this chapter, *we propose dynamically reconfiguring the cache on the fly, using the microarchitectural behavior of application phases*, which are different portions of an execution. We propose this specifically for *energy-harvesting processors* first to solve the smallest case of optimizing hardware resource allocation and performance.

With the increasing use of Internet of Things (IoT) devices in different areas of computing, many applications depend on wireless sensor networks (WSNs) in critical application domains such as smart farming, incident monitoring, and health monitoring [68, 95, 107, 175, 206]. The variety of these areas indicates that IoT devices have become integral to our lives by addressing pressing problems.

For example, to meet the food production needs of a growing world population [68], smart farms that use large-scale WSNs [72], and allow remote crop and livestock monitoring [133] are in great demand [95]. Similarly, WSNs are used in detecting fires [107]

and supplementing visual incident detection with audio [175]. WSNs are also employed in continuous health monitoring, especially for assisted-living patients and remote health monitoring [206]. Such WSNs provide continuous medical monitoring (including sleep monitoring for sleep apnea [140], activity monitoring for detecting falls [64, 159], heart arrhythmia detection [178]), emergency communication, and memory enhancement [206].

WSNs for these domains should involve low maintenance and be self-sufficient since they should strive for continuous "forward progress" (e.g., number of successfully executed and persisted tasks/instructions or transmitted datagrams), even in electrical outages, due to their vital roles in human lives. Smart farms have continuous energy requirements for farmers' market and production needs [95]. Fire detection and incident monitoring networks should have an extensive area coverage [107, 175]. Reliability and continuous forward progress are necessary for health monitoring WSNs [206]. To meet these requirements, Energy Harvesting (EH) Non-Volatile Processors (NVPs), as general-purpose processor gateways (GPGs) for WSNs, can be instrumental.

EH devices perform computation by harvesting energy and storing it in a capacitor, then *intermittently* making forward progress on the application whenever there is "enough" energy stored in the capacitor [109] to begin executing instructions and to persist their outputs. While recent years have seen continued improvements in energy storage devices, the development of NVP GPGs has not been the focus of these improvements [109–111]. Notably, current EH NVPs allow minimal and course-grained dynamism in the system states, often in terms of frequency of operation or size of resources like on-chip caches [112, 113], but barely explore other dynamic reconfiguration options like varying cache ways or operational memory. Further, they are program agnostic even though predictable runtime application behavior is observed, which is the case in many execution environments where NVPs are employed. This behavior hinders progress in application forward, leading to low quality of service (QoS). Instead, if the sensor nodes of WSNs could offload computations to the gateways and if these gateways could achieve enough forward progress by being aware of, for instance, the memory boundedness of an application and EH environment, dynamically reconfiguring hardware would save energy. It would avoid/reduce the need for costly/frequent maintenance of hundreds of distant WSN nodes.

We aim to solve low "forward progress" and high maintenance cost challenges in the EH NVP WSNs employed in application domains, including smart farms, incident monitoring, and health monitoring. One crucial factor that decreases the forward progress in EH NVPs is the high percentage of energy spent on on-chip cache memories (10%-30% of the total energy) [28, 181, 182, 212]. So, focusing specifically on the on-chip cache

memories employed in EH systems and encouraged by prior results [96, 181, 182, 212] demonstrating the importance of on-chip caches for energy optimization, the specific challenges we address in this work include:

- Predicting sudden drops in unreliable and intermittent ambient energy when harvesting;
- Predicting the memory boundedness of the future phases of a given application;
- Using these predictions to determine configurations for L1-instruction, L1-data, and L2 caches, i.e., dynamically reconfiguring them, to decrease cache energy utilization to increase "forward progress" (therefore performance).

Solving these challenges is non-trivial because dynamically tuning cache configuration for energy efficiency, based on predictions of energy and application behavior, can have an undesirable impact on performance. Towards addressing the challenges above, we present and evaluate *Mystique*, which is a dynamic application phase and energy environment prediction-based cache scaling strategy designed explicitly for NVPs. In addition to intelligently scaling the cache based on power availability, our approach also accounts for the application needs of the resources. For example, the cache capacity is downsized even when sufficient energy is available if there is knowledge of a smaller memory footprint for the upcoming execution epoch. Our contributions are as follows:

- We propose a minimum energy utilizing memory-boundedness predictor.
- We propose an energy predictor to predict energy utilization and harvesting for future application phases to proactively re-size the cache based on available power.
- We propose an energy environment and application memory-boundness aware dynamic "cache reconfiguration" scheme to maximize "forward progress" and increase performance in EH NVPs.
- We evaluate the proposed approach using our *cycle-accurate NVP simulator* and a set of 12 application programs targeting different EH scenarios. We obtain ∼11% increase in performance by decreasing the cache energy consumption by ∼96% and reducing power failures by ∼28%.

6.2 Motivation

Many emerging IoT applications rely on multi-modal sensing and typically operate using networks of wireless sensor nodes that collect large-scale data and communicate it to a central gateway capable of high-throughput general-purpose computing for data processing and analytics. Applications like smart farming and remote incident monitoring often rely on many such sensor nodes. These sensor nodes are often limited by a small

form factor and the challenges of ensuring data integrity and reliable communication. So, the robustness of the general-purpose processors (GPPs) at the gateway becomes extremely important. However, deploying a battery-powered general-purpose, power-hungry processor in the gateway, which is often remotely located, might not be practical in many cases because of the frequent maintenance constraints (e.g., for battery replacement). Fortunately, these application domains are often rich with opportunities for **harvesting ambient energy**, like using the vibration of a pump [207] or a grain cleaning machine [49], which can be used to power or augment the capabilities of the processor. Considering the unreliable nature of these harvested energy sources, consistently provisioning energy for a power-hungry, commercial off-the-shelf GPP can be challenging. It also poses an interesting opportunity to consider a "software-hardware co-design" for *maximizing* the utilization of harvested energy and efficiently utilizing available hardware resources while making *useful forward progress.*

Several works [27, 40, 114] have leveraged software- and compiler-based solutions, such as checkpointing, code optimization, and approximate computing, to *intermittently* store and execute the given program in an energy-scavenging scenario. However, their dependence on commercial off-the-shelf hardware, which often loses state on power failures, forces them to perform redundant saves and restores, thereby wasting cycles and energy on unnecessary operations. Therefore, just using software for freezing and retaining the system state by saving the register contents of a typical processor to an NVM is *not* always feasible if that NVM is further in the memory hierarchy [155] and save/restore costs are high in time and energy. Recent works on NVPs [109, 110, 113, 134, 155, 188] use their inherent non-volatility to maintain the state of the processor, eliding the requirement software save and restores. However, these works do not take advantage of the information about the nature of the running application and do not try to save battery power by harmonizing *application-specific* hardware "resource requirement prediction" and "energy prediction" to *dynamically reconfigure* the underlying hardware.

Approximate computing [40] is a solution proposed for more forward progress by using subword pipelining, subword vectorization, and skim points techniques to transform all-or-nothing computing to as-is computing. However, in some applications [107, 159, 175, 178], accuracy is also a critical factor and approximate computing may not be the best option for achieving desired accuracy and forward progress at the same time. Checkpointing the system state by saving it to an NVM is a solution for keeping the state during a power outage [114]. However, even in checkpointing, some system state data may be lost due to the granularity of checkpointing. Besides, using non-volatile hardware

components such as registers, cache, and STT-MRAM allows the system state to be retained within these components whenever a power failure happens, showing that checkpointing is unneeded. Therefore, first inspecting application phase behavior, then using that information with the knowledge about energy environment, and combining them to *reconfigure* the hardware can be more beneficial regarding the forward progress of applications. So, the probability of rolling back and restoring decreases as energy is utilized more optimally.

In a typical system, a large cache improves instruction locality, data locality, and performance at the expense of more passive power consumption and high per-access energy. However, an EH system does *not* have a consistent and reliable energy budget, and different cache configurations can lead to power emergencies. Thus, EH systems can reap substantial benefits from intelligent "dynamic reconfiguration" of cache for energy efficiency to avoid power emergencies if this can be done while maintaining the desired performance. The same can be said for memory, but at the expense of frequent reads and writes from the storage to enable such reconfiguration. We see a significant benefit in energy and performance by intelligently reconfiguring the cache associativity. The benefits are more pronounced in an energy scavenging system which often uses an NV cache[1] for data retention in times of (sporadic but sometimes frequent) power failures, mainly because a significant portion of the power that is dissipated goes to static leakage. This work focuses on predicting the cache usage and turning off the non-essential parts of the cache to save energy and improve the instantaneous energy consumption while ensuring forward progress.

We aim to increase forward progress, optimize energy utilization (by decreasing dynamic and static cache energy consumption), and increase performance by reducing power failures. Various applications, especially in the IoT domain, make many data accesses and consume high energy in on-chip caches. In particular, prior works [96, 181, 182, 212] report that 10-30% of the total energy can be spent in on-chip caches. Our experiments showed that applications in the smart farm, incident, and health monitoring settings have different access patterns and memory intensities. Also, there is a large gap between application phase prediction and EH scenarios. The harvested energy is unpredictable, so we need an energy predictor and a phase predictor to predict the memory intensity to reconfigure the cache according to memory and energy needs. Therefore, there is a strong need for a *harmonized system* that can operate best depending on the compute and memory demand from the phase predictor and the future status of the

[1]Hereafter, we use "NV cache" and "cache" interchangeably.

energy environment from the energy predictor to maximize forward progress. This system would remedy the energy limitations of the GPGs because it saves energy and ensures "maximum forward progress" in various time-sensitive IoT applications.

6.3 Background

6.3.1 Non-Volatile Processors (NVPs)

Energy harvesting (EH) [109, 188] offers a promising solution to addressing energy needs in the IoT domain. However, the intermittent nature of harvested energy causes reliability issues and performance degradation due to frequent power emergencies. NVPs [109–111, 188] allow forward progress even with *unstable* incoming energy. Conventional CMOS circuits are volatile and forget everything in power outages and cause power leakage during sleep mode [187]. This behavior causes rollbacks when a reboot happens. NVPs perform backups of the system context into a Non-Volatile Memory (NVM) when there are power failures and restore that system context whenever the power level becomes higher than a certain threshold, achieving more forward progress [187, 188]. So, NVPs use non-volatile technologies and maintain temporary states within non-volatile flip-flops (NVFFs) when the power drops a certain threshold and resumes to compute from the exact state once the power increases above a threshold [188]. Figure 6.1 illustrates an example of NVP. Prior research demonstrated that NVPs could lead to ∼1000× speedup in backup and recovery times over conventional processors [188]. Further, NVPs can be used in many IoT applications because they require them to be self-sufficient in power and maintenance and are limited in size.

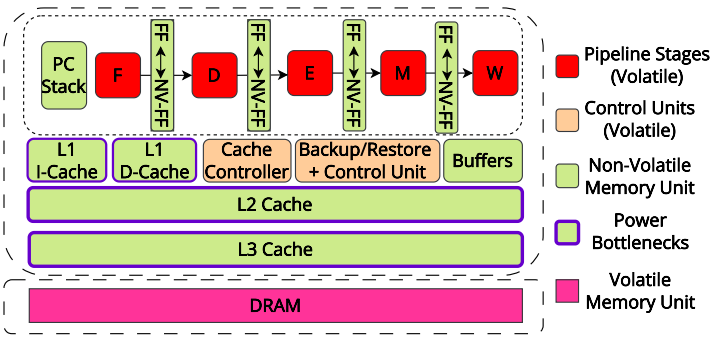


Figure 6.1: A fundamental block diagram of an NVP.

6.3.2 Application Phase Prediction

There has been longstanding interest in workload characterization in the computer architecture domain, leading first to "phase analysis," which tracks the changes in the execution behavior of applications, and later to "phase prediction." In this context, a "phase" is defined as a set of contiguous portions of program execution with similar behavior, irrespective of temporal adjacency [99]. Phase classification partitions a set of intervals of a specific metric into phases with similar behavior and predicts the upcoming interval's phase classification in the program execution [99]. Representative phase prediction works from high-end computing and embedded systems domains include [52, 69, 99, 173, 174].

In this work, we employ a phase prediction strategy similar to the one proposed in [69], which proposes a real-system implementation of a runtime phase predictor that allows on-the-fly dynamic resource management. The critical insight behind the approach in [69] is to use an algorithm similar to previous branch predictor designs to perform phase prediction with a phase history table (PHT). In the proposed approach, the authors use a PHT, and each entry in the PHT is calculated using the $Memory\_transactions/\mu Ops$ ($mem/\mu Op$) metric. The authors divide phase values into ranges to identify the memory boundedness of each phase for phase classification. The approach to predicting the next phase value uses patterns in the PHT, which are stored in the pattern history table. Using this pattern history table, they match the PHT entry patterns with a pattern in the pattern history table and identify the phase. The *least recently used (LRU)* policy is utilized to update the PHT. Their results with this predictor indicate around 90% prediction accuracy for many of the tested benchmarks and a 34% improvement in the energy-delay product when the phase prediction strategy is coupled with dynamic frequency and voltage scaling [69]. Even though we utilize a similar phase prediction strategy to this [69] work, we modified the algorithm to create a low-power phase prediction and reconfiguration scheme by omitting the pattern history table and calculating the weighted average of the PHT entries, which we obtained by using the same metric as [69]. Figure 6.2 shows the high-level view of our predictor implementation.

6.3.3 Energy Behavior Prediction

Understanding the energy environment for upcoming phases of an application is necessary for increasing the effectiveness of decision-making strategies in dynamic hardware reconfiguration for EH NVPs. This is because harvested energy is intermittent, and predicting

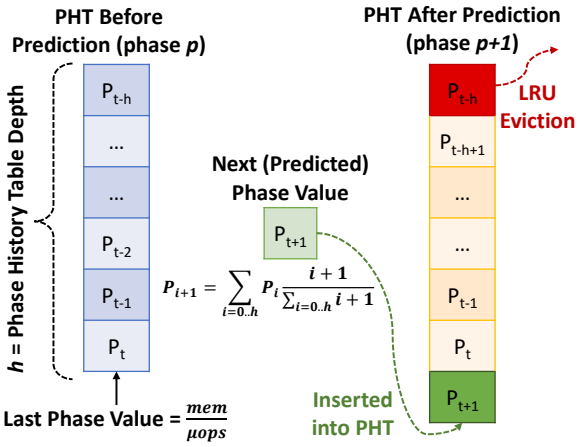


Figure 6.2: High-level view of Phase Prediction module.

the future energy environment will avoid harmful hardware configurations for maximum energy efficiency purposes. Previous work has shown that energy prediction is critical to designing an energy-efficient architecture for energy-harvesting environments [155]. Using an energy predictor, coupled with the application phase predictor, can increase the accuracy of the reconfiguration algorithm, and we adopt this approach in our work.

6.3.4 Dynamic Cache Reconfiguration

Most existing microarchitectures employ a static memory hierarchy configuration suitable for an average (target) application [13]. However, no single memory hierarchy configuration is optimal for all applications. Previously, it has been shown that employing a reconfigurable design can potentially increase energy efficiency in EH scenarios [155]. Moreover, other previous works ([13, 174]) use phase detection algorithms to reconfigure the memory hierarchy in GPP architectures dynamically. Since additional cache resources do not contribute to forward progress if an application does not use them, dynamically reconfiguring the memory hierarchy can significantly reduce the energy consumption of caches in modern processors [13, 174]. We use a similar approach to [174] in *Mystique* to create our dynamic cache reconfiguration strategy and to tackle this inefficiency. In [174], the associativity of the cache is *dynamically* decreased in execution phases where higher associativity is not expected to be beneficial. When a lower cache associativity is used, some programs have performance degradation with a decrease in energy utilization; some programs use more energy due to degradation in performance; and some programs have high energy savings with no degradation in performance. Therefore, "dynamic cache reconfiguration" can be very beneficial for the "forward progress" of EH NVP systems,

where performance degradation is negligible and energy savings exist.

In our work, after predicting the phase value (memory boundedness) and energy behavior of the EH system at every phase interval, we reconfigured the cache according to the phase intervals[2] used in [69]. However, we did *not* reconfigure every cache level/layer with the same associativity; instead, we increased the associativity of the L2 cache at a lower pace than that of the L1 caches. This is because the L2 cache is large, and consequently, the static and dynamic energy consumption would be very high, which would decrease the energy savings, hence the forward progress. That is, having higher cache associativity may be unnecessary in the smaller phase indices (lower memory boundedness). While the idea behind our proposed approach can be applied to other hardware resources as well (in particular main memory), our focus is on dynamic cache reconfiguration due to the high static and dynamic energy consumption of caches, especially when the associativity is high [22, 83, 151, 181, 200, 201, 212, 213].

6.4 System Design

This section presents the system design of the WSN that we intend to incorporate Mystique. More specifically, we explain the details of the hardware used for sensor nodes, the local gateway concept, the main server, and the communication and computation components.

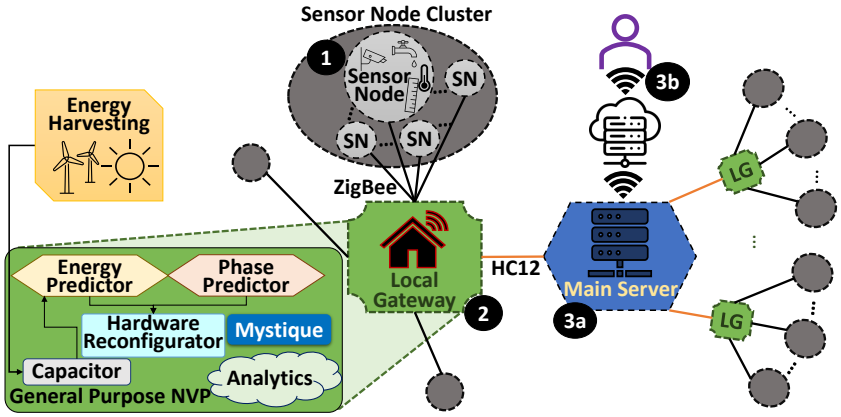


Figure 6.3: WSN System design incorporating Mystique.

As seen in Figure 6.3, ① represents the Sensor Node Clusters (SNC). These clusters consist of multiple sensor nodes specific to the intended use case of the WSN. For example, there can be 3-4 sensor nodes in these clusters for a smart farm, whereas there can

[2]We refer to phase intervals as "phase indices" throughout the text.

be 2-3 for incident monitoring and typically only 1 for health monitoring. Moreover, for a smart farm and incident monitoring WSN, 4-5 SNCs are distributed throughout a large area. For a health monitoring WSN, 2-3 SNCs are distributed throughout an apartment/house/facility in which a patient resides. The types and capabilities of the sensors in these SNCs change depending on the use case. In our system, the sensor nodes have 16KB flash memory and 2KB SRAM. The SNCs communicate with the Local Gateways (described in the subsequent paragraphs) using the ZigBee communication module [120], which uses IEEE 802.15.4 2006 with 127-byte packets and an underlying data rate of 250 kbps. The Zigbee packet format results in a 68-byte payload, and if the payload is above 68 bytes, a packet is *fragmented* into multiple sub-packets. Zigbee is more suitable for short-distance communications and is designed to be energy efficient, consuming between 10-100 mW [120]. Since ZigBee has a low power and data rate, it is particularly suitable for our EH scenario, where significant fluctuations in the incoming energy exist. Hence, it is used to communicate between SNCs and Local Gateways.

Local Gateways (LG) (②) are used in WSNs. LGs are not conventional gateways, as they consist of multiple LGs connected and are gates between sensor nodes and the main server. These gateways are distributed to far away locations and cannot be wired because they are either mobile or are in hard-to-reach places. Hence, they are typically battery-backed. LGs allow little maintenance since they do not always have to be active (only when they need to be computing/receiving data from SNCs). Therefore, the LGs decrease the costs of maintaining WSNs.

Moreover, since LGs are not wired, they can use "energy harvesting" (e.g., when they are located in shoes during health monitoring or smart farm equipment engines, using piezoelectricity); hence, they are inherently environment-friendly. LGs would benefit from having Non-Volatile Processors (NVPs), allowing more "forward progress" in unreliable energy environments and power failures. Furthermore, the sensor nodes cannot directly send the data to a main gateway/server because they are extremely far away and may not have enough energy to communicate data to a far main gateway/server/cloud, which motivates the usage of local gateways. The LG concept is proposed to ensure that communication is performed over shorter distances, which reduces the energy cost for wireless communication.

We focus on "local gateways" and incorporate *Mystique* into these LGs. LGs use the HC-12 wireless communication module, designed for long-range serial communication (up to 1 km), because the distance between local gateways and the main servers could be considerable in some use cases. Moreover, HC-12 is cheaper and more reliable than

other long-range wireless communication modules [35]. Our local gateways use NV SRAM (8T hybrid nvSRAM cell employing FinFETs and ferroelectric FETs) for the NV caches. This technology supports on-chip backup and restore capabilities by enabling energy-efficient and low-latency store/recall operations [211]. Our local gateways use STT-MRAM because it is smaller and requires low power, making it suitable for EH NVP IoT WSN use-cases [12].

The Local Gateways are connected to a Main Server (③ⓐ), a "wired" static server that obtains "end-computation results" from the LGs. The main server is connected to the "cloud" (③ⓑ), using LTE/3G/4G to communicate the end computations. Furthermore, the end-users connect to the cloud via wireless communication or LTE/3G/4G to obtain the end-computation results to complete the desired smart farm, incident, or health monitoring tasks.

6.5 Incorporating Mystique

This section describes the design of Mystique (see Figure 6.4) and how it is incorporated into the standalone NVPs of local gateways of our wireless sensor network model (see Figure 6.5). We provide the details of the predictors and the dynamic cache reconfiguration. First, we explain the NVP state machine design, the *Application Phase Predictor (APP)*, the *Energy Behavior Predictor (EBP)*, and lastly, the *Dynamic Cache Reconfiguration (DCC)*. The APP and EBP results are merged to decide on a cache configuration at the end of each execution phase of an application. We concentrate on a granularity of 10 million instructions for one phase unit. Our experimental analysis shows this as an optimal frequency for capturing complex phase behaviors of multiple workloads from diverse application domains. It is compatible with the energy harvester frequency ($100\mu s$).

6.5.1 NVP State Machine Design

Our local gateways consist of NVPs because they would significantly benefit from higher forward progress even when utilizing the fickle ambient harvested energy. Our EH NVP system has `POWER ON`, `RETENTION (SLEEP)`, and `POWER OFF` states. The harvested energy is stored in a capacitor. We also have two energy thresholds (`thresh_1_to_ret` and `thresh_ret_to_off`), and our processor design has a state machine to execute the state changes systematically. When the *Energy* is below `thresh_1_to_ret`, and the state is

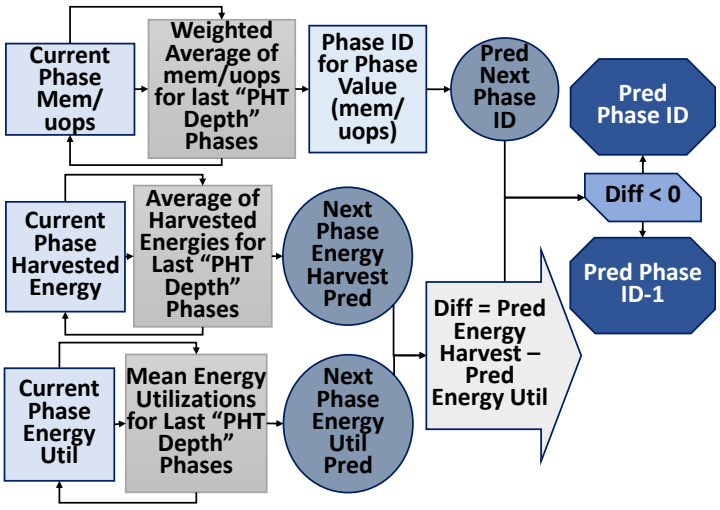


Figure 6.4: System design of Mystique.

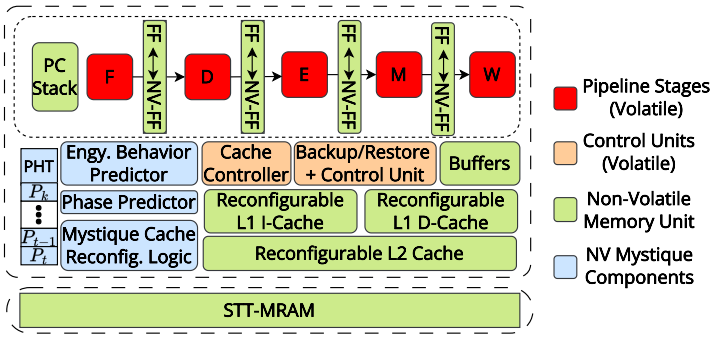


Figure 6.5: Mystique being integrated into a standard NVP

`POWER ON`, the state becomes `RETENTION`. When the `Energy` is above `thresh_1_to_ret` and the state is `RETENTION`, the processor state becomes `POWER ON`. When the *Energy* is below `thresh_ret_to_off` and the current state is `RETENTION`, the state becomes `POWER OFF` state. When the *Energy* is above `thresh_ret_to_off` and the current state is `POWER OFF`, the processor state becomes `RETENTION`. When the *Energy* is below `thresh_ret_to_off` and the current state is `POWER ON`, the processor goes to the `POWER OFF` state. When the *Energy* is above `thresh_1_to_ret` and the state is `POWER OFF`, the processor state is transitioned to `POWER ON`.

6.5.2 Application Phase Predictor (APP)

To understand how much an application will utilize each hardware component in the upcoming phases, we implemented an APP that uses a similar approach to [69]. We opted to employ this algorithm in our system because it is suitable for phase prediction

accuracy and minimal energy usage. We implemented our application phase predictor by using the already available hardware counters. The two counters we used are i) the number of memory bus transactions and ii) the number of micro-operations[3]. We use the $mem/\mu Op$ metrics for each phase of the application because it gives a vital insight into the current phase's "memory boundedness," which allows us to guess the cache requirements for the upcoming phases more accurately. For most applications, application phases are regular (not many anomalies in phase values) throughout the execution. This behavior allows the recent history to capture critical information to predict the upcoming application behavior.

In our phase predictor, using a similar concept to [69], we created a phase history table (PHT) which contains the $mem/\mu Op$ phase values for the last h phases. Using PHT, we predicted the upcoming phase behavior by taking the weighted average of the h values in the history table (see Figure 6.2). The historical averaging allowed us to address the oscillation problem that will be explained in more detail in Section 6.6. We used similar ranges of this phase value as [69] and divided the ranges into six different configuration schemes as seen in Table 6.1. We changed the algorithm in [69] to make it less energy-consuming by eliminating the pattern prediction table and decreasing the two-level phase prediction scheme to one-level.

Phase ID	**Phase Interval**	**L1I and L1D**	**L2**
1	`phaseVal` <0.005	1	1
2	0.005 <= `phaseVal` <0.010	2	1
3	0.010 <= `phaseVal` <0.015	4	2
4	0.015 <= `phaseVal` <0.020	8	4
5	0.020 <= `phaseVal` <0.030	16	8
6	`phaseVal` >= 0.030	16	16

Table 6.1: Phase value ranges used in our cache associativity reconfiguration policies.

Furthermore, the weighted averaging scheme gave more importance/weight to the most recently seen phase values since they likely better represent upcoming phases regarding hardware behavior. After calculating the weighted average of the values in the table, we predicted the memory boundedness of the upcoming phase. Our phase predictor module is developed as an extension of the cache controller. The PHT is stored as a register (6 rows $\times$ 4 bytes per row = 24 bytes), along with the cache controller. We used the First-in, First-Out (FIFO) replacement policy to evade the values in the PHT.

[3]Micro-operations are low-level instructions that make up complex instructions. They are in both ARM and x86 architectures.

6.5.3 Energy Behavior Predictor

In addition to predicting the memory boundedness of the upcoming phase, we predict how much energy would be harvested and utilized for the upcoming phase. For EH prediction, we used an energy harvesting table by recording how much energy was harvested in each of the last h phases. We calculated the average of the last h phases for EH prediction and predicted that that value would be harvested for the upcoming phase. For the evictions of old values, we used the FIFO replacement policy. This scheme is less energy-consuming, and since we are using piezo energy in our EH system, the deviation is minimal, making our predictor more accurate.

Similarly, we predicted the energy utilization for the upcoming phase using the energy utilization table for the last h phases. After obtaining the predicted energy harvesting and utilization numbers for the upcoming phase, we compared them. If the predicted energy harvesting is less than the utilization, we used a more conservative approach to increase the cache associativity. Our energy predictor modules are developed as an extension of the cache controller, similar to our phase predictor module. The energy predictor tables are stored in our NVP as registers (2 tables $\times$ 6 rows $\times$ 4 bytes per row = 48 bytes) along with the cache controller.

6.5.4 Dynamic Cache Reconfiguration

While reconfiguring the cache associativity, we used *both* application phase predictor and energy predictor as a "pipeline" (see Figure 6.4). First, after obtaining the application phase prediction output, we decide on an associativity configuration for all three caches according to Table 6.1. Later, we compare the predicted harvested energy with the predicted utilization value for the upcoming phase. If energy harvested for the upcoming phase is less than the predicted energy utilization for the upcoming phase, and if the phase predictor tries to increase the associativity, we increase it to a configuration level that is one less than the original prediction, as can be seen in the right side of Figure 6.4. This policy is crucial for ensuring a "conservative" strategy for the upcoming phase and reducing the caches' static leakage and dynamic energy consumption. This behavior is because if cache associativity is small, the static leakage and dynamic energy consumption would significantly reduce, thus allowing more energy for the application's forward progress. Before reconfiguring the cache associativity to a lower value, we ensure that all the writes are propagated and the dirty blocks are coherent in the memory and the cache.

6.6 Evaluation

We aim to explore a "multi-dimensional" evaluation space that includes application execution time, cache hits, cache misses, main memory requests, power failures, and energy utilization metrics. Since our main method/knob is to decrease the number of cache-ways whenever they are not needed without hindering the application runtime, we show why decreasing the dynamic and static cache energy usage would benefit the application's *forward progress* in the context of EH NVPs.

6.6.1 Benchmarks

To evaluate Mystique, we chose a set of representative applications crucial for the target smart farm, incident monitoring, and health monitoring WSNs to operate. Most of our applications are from the MiBench [53] benchmark suite. We implemented the `Arraywalk32` and `Arraywalk128` benchmarks. The `K-Means` and `KNN` benchmarks are from two GitHub repositories[4]. Figures 6.6, 6.7, and 6.8 show the system flow diagrams of the use-cases of our application suite, which includes the following programs:

`Fast Fourier Transform (FFT)`: To process raw signals coming from sensors [140].

`String Search, Arraywalk32, and Arraywalk128:` To search strings with queries in the database for the end users of the Smart Farm, Incident Monitoring, and Health Monitoring Apps [95, 107, 206].

`Blowfish Encode` and `SHA`: To encrypt raw data when they have to be sent to the LG for security purposes [95] and encrypt user information.

`Blowfish Decode`: To decode data for computing in the LG.

`Basicmath`: To perform basic mathematical operations for calculating temperature and moisture values of crops in Smart Farm [68, 95]; calculating sound and temperature values [107] to detect incidents for incident monitoring [175]; and calculating and detecting abnormalities in heart rates [178] and breathing patterns in sleep apnea detection [140] for health monitoring.

`Bitcount`: For compressing raw sensor node data.

`Susan Smooth`: To perform image processing to assess crop health [68, 72].

`K-Means`: To perform crop disease detection using clustering in smart farms [202]; human activity recognition (HAR) for health monitoring for assisted living patients [64].

`K-Nearest-Neighbors (KNN)`: To perform ECG arrhythmia detection [178]; activity recognition [159] in health monitoring.

[4] `K-Means`: https://github.com/pramsey/kmeans;
`KNN`: https://github.com/Rabadaz/K-Nearest-Neighbor-C

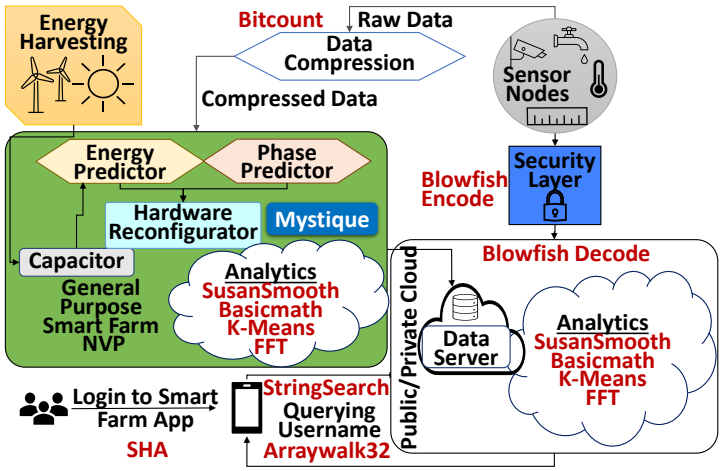


Figure 6.6: The system flow for smart-farm.

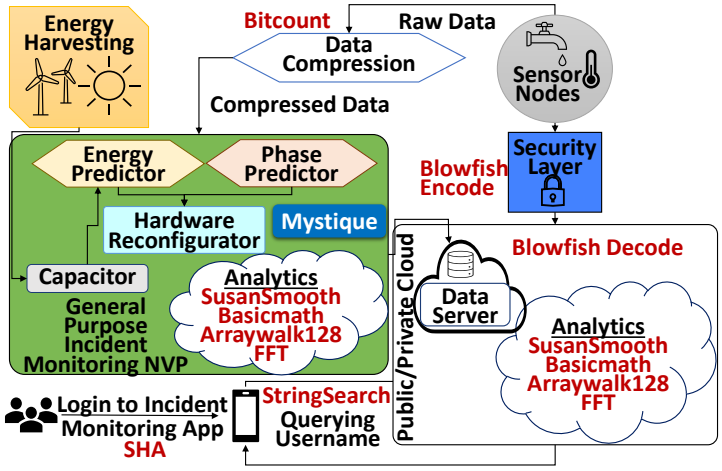


Figure 6.7: The system flow for incident monitoring.

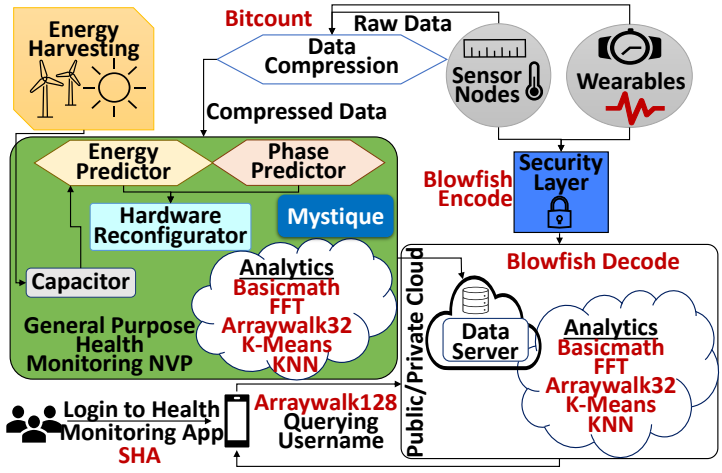


Figure 6.8: The system flow for health monitoring.

Workload	Smart Farm		Health Monitoring		Incident Monitoring	
	Receive	Transmit	Receive	Transmit	Receive	Transmit
arraywalk32	2	1	690	480	0	0
arraywalk128	0	0	2	1	480	240
basicmath	24	24	480	480	240	240
bitcount	192	0	4800	0	1920	0
blowfish_decode	192	0	4800	0	1920	0
blowfish_encode	192	0	4800	0	1920	0
FFT	240	24	4800	480	2400	240
stringsearch	2	1	0	0	2	1
sha	0	0	0	0	0	0
susan_smooth	48	24	0	0	480	240
K-Means	240	24	4800	480	0	0
KNN	0	0	4800	480	0	0

Table 6.2: Daily number of packets sent and received by the LG for the benchmarks in each IoT workflow.

We performed experiments for the execution phases of the applications in our suite and created a "lookup table" for each case (regarding different Oracle policies and predictors). Our experiment script traverses the lookup tables to find the paths for the statistics of every possible scenario to compare the predictor results to the Oracle policies.

6.6.2 Benchmark Communication Modelling

In this section, we present the communication frequencies of the benchmarks for each IoT workflow as seen in Table 6.2. Each packet transmitted is 68 B, due to the communication protocols mentioned in Section 6.4. Table 6.3 shows the daily energy spent in mJ by the LG for the communication portion of the benchmarks in each IoT workflow. We assumed 20 kbps data transfer speed and 50 mW data transfer power for the ZigBee module (average for higher distances, as ZigBee has 10-100 mW power specification) [120].

6.6.3 Implementation

We implemented a "cycle-accurate" non-volatile processor (NVP) simulator that can capture all the dynamic energy consumptions and static leakages of hardware components. Our simulator is built on top of the GEM5 simulator [16]. We implemented our design's phase predictor, energy predictors, and dynamic cache reconfiguration components in this NVP simulator. We used a real-world harvested *piezoelectricity energy trace* [155] to obtain the EH information in our simulations. The trace that we used is plotted in Figure 6.9.

Workload	Smart Farm		Health Monitoring		Incident Monitoring	
	Receive	Transmit	Receive	Transmit	Receive	Transmit
arraywalk32	2.7	1.4	938.4	652.8	0.0	0.0
arraywalk128	0.0	0.0	2.7	1.4	652.8	326.4
basicmath	32.6	32.6	652.8	652.8	326.4	326.4
bitcount	261.1	0.0	6528.0	0.0	2611.2	0.0
blowfish_decode	261.1	0.0	6528.0	0.0	2611.2	0.0
blowfish_encode	261.1	0.0	6528.0	0.0	2611.2	0.0
FFT	326.4	32.6	6528.0	652.8	3264.0	326.4
stringsearch	2.7	1.4	0.0	0.0	2.7	1.4
sha	0.0	0.0	0.0	0.0	0.0	0.0
susan_smooth	65.3	32.6	0.0	0.0	652.8	326.4
K-Means	326.4	32.6	6528.0	652.8	0.0	0.0
KNN	0.0	0.0	6528.0	652.8	0.0	0.0

Table 6.3: Daily energy spent in mJ by the LG for the communication portion of each benchmark in each IoT workflow. We assumed 20 kbps data transfer speed and 50 mW data transfer power for the ZigBee module.

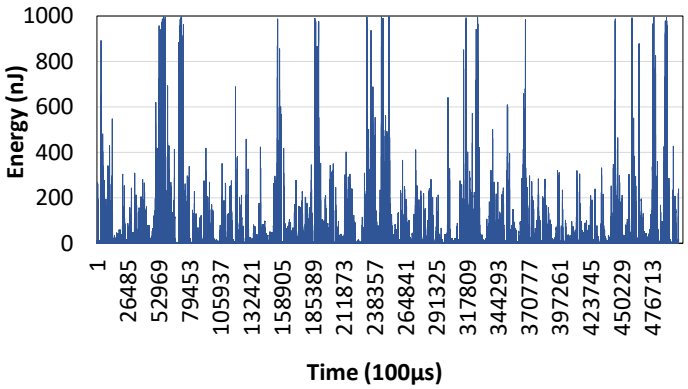


Figure 6.9: Real-world piezoelectricity energy trace used in our simulations.

Our simulator accurately models the latencies and static/dynamic energy consumptions of different cache configurations. We used the CACTI simulator [138] to obtain the energy consumptions and latencies of different cache associativities for different cache levels/layers. We used [12] and DRAMSim [164] for modeling the main memory energy consumptions. We also leveraged [38, 163, 167] to accurately model the cache and capacitor leakages.[5] The proposed system always allows padding for communication energy consumption for receiving/transmitting samples before power failures. The data collection/aggregation from the sensor nodes and data transmission/reporting to the main server are modeled as "communication overheads." The streaming nature of data collection/aggregation does not involve much cache usage dynamism. Therefore, we only

[5]The simulator code is publicly available at https://hackmd.io/@anonymous-gxg/B1d06pHY3.

consider filtering and other compute kernels in our workloads.

6.6.4 Experimental Setup

We adapted the system configuration from a low-cost information monitoring WSN design for smart farming applications [169]. This design uses ZigBee to communicate between sensor nodes and LGs. ZigBee has a low power and data rate, making it particularly suitable for our EH scenario, where significant fluctuations in the incoming energy exist. Similar to [169], our WSN also uses HC-12 to communicate between the LGs and the main server. In our system, the sensor nodes have 16KB flash memory, 2KB SRAM, and an HC-12 module for communication (cf. Section 6.4). Our LGs use NV SRAM for the NV caches [211]. Moreover, our LGs use STT MRAM since it is smaller and low power, making it suitable for the IoT WSN use-cases [12].

Our WSN-based applications require multiple sensor nodes attached to LGs. Therefore, they need high I/O bandwidth for different data types and compute-heavy applications to run in a reasonable execution time, meaning that gateways with GPPs that offer multi-processing, parallel I/O, and addressable memory are highly desirable for the design of the LGs of our system [95, 169]. On the other hand, microcontrollers, used in many IoT settings, are unfit for the application domains mentioned above because these domains require running complex algorithms using large amounts of data gathered from hundreds of sensor nodes distributed over a large area. Microcontrollers lack multi-processing and parallel I/O capabilities, and their instruction sets are usually limited. Since our target applications are diverse in terms of memory, storage, and program type due to multi-modal sensing and different algorithms, and there are a lot of data transmissions requiring high I/O throughput, attaching an NV memory to a microcontroller would *not* be sufficient. Our processor is 800MHz, and all the application programs tested are compiled against ARM architecture. A single-threaded ARM core is simulated.

The capacitor that stores the harvested energy is 70 Milli Farads, and we have a piezoelectricity power trace, which has a frequency of harvesting of 100us. We used the piezoelectric power source in our design because it is readily available in *all* of the application domains that we target. For example, smart farms could have a vibrating motor in harvester machines, and the waste vibrations can be used for harvesting piezoelectricity [207]. Besides, the piezoelectricity can be generated from automated grain cleaning [49]. For incident monitoring, piezoelectricity can be generated from the ambient sound [36]. Lastly, in health monitoring, the piezoelectricity can be generated from either the shoes of the patients that can walk [42] or other bodily movements such

as muscle contractions or breathing [154]. Given the piezoelectric harvesting capabilities, charging and continuously operating a processor at 800MHz is difficult. However, in many IoT WSN cases, deployment is not form factor-limited and can be augmented with multiple energy harvesters. For example, running workloads once a day in smart farms would require 50 piezoelectric harvesters.

The cache sizes and associativities corresponding to phase reconfiguration indices are shown in Table 6.4. The phase reconfiguration indices correspond to different ranges of phase values (calculated with the $mem/\mu Op$ metric) as seen in Table 6.1 and as explained in Section 6.4. We increased the associativity of the L2 cache one index later than the L1 cache because, since the L2 cache is large, the static and dynamic energy consumption would be high. This would decrease energy savings, hence slowing down progress. Consequently, having higher associativity may be unnecessary, especially in the first and second phase indices, where the memory operations are much less than micro-operations. The main memory in our system is 4 MB.

Index	**L1I and L1D Caches**		**L2 Cache**	
	Size (B)	**Associativity**	**Size (B)**	**Associativity**
1	4096	1	32768	1
2	8192	2	32768	1
3	16384	4	65536	2
4	32768	8	131072	4
5	65536	16	262144	8
6	65536	16	524288	16

Table 6.4: L1-Instruction, L1-Data, and L2 sizes, associativities, and corresponding phase reconfiguration indices.

6.6.5 Results and Analysis

To evaluate the benefits of Mystique, we selected several metrics such as *execution time*, *cache energy consumption* (cache dynamic read and write access energy consumptions and leakage consumption), and *number of power failure states.* These metrics give vital insight into the benefits that our system brings in an EH WSN setting.

We used metric graphs to better explain the advantages/ disadvantages of Mystique in terms of the metrics mentioned above. We compared the "best-performing oracle" for *each metric* with our predictor and the baseline. The execution time metric (Figure 6.10), cache energy consumption (Figure 6.11), and power failures graphs (Figure 6.12) were the most crucial metric graphs chosen to study the performance gains achieved by

our proposed system. Figure 6.13, which gives the dynamic and static cache energy breakdowns for Mystique and Baseline, illustrates the reasons for the presented results. **Baseline Selection:** We created several "oracle" experiments targeting the minimum energy, maximum useful energy (utilized_energy / harvested_energy), minimum execution time, minimum cache miss, maximum cache hit, minimum main memory read and write requests configurations. We compared Mystique and the oracle results to a **baseline** configuration, shown in Table 6.5, which uses 16-way L1-Instruction, L1-Data, and L2 caches and keeps static associativity throughout the execution. Since, in the baseline, the caches are not reconfigured to lower associativities, the unneeded cache ways still consume energy.

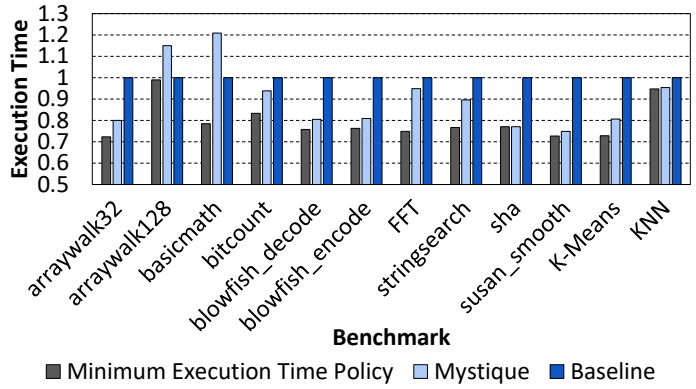


Figure 6.10: Execution times for different benchmarks normalized to the baseline.

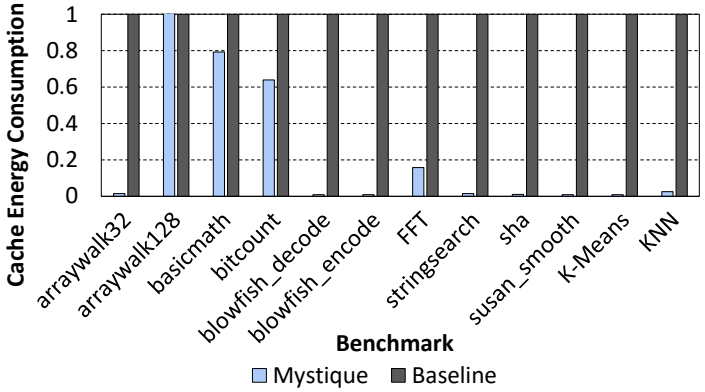


Figure 6.11: Cache energy consumptions for different benchmarks normalized to the baseline.

Figure 6.10 plots the execution time for each workload for the minimum execution time policy oracle, Mystique, and baseline, *normalized*, to the baseline values. As can

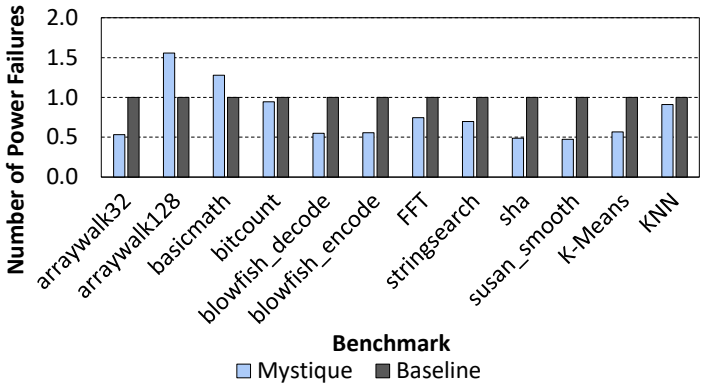


Figure 6.12: Power failures for different benchmarks normalized to the baseline.

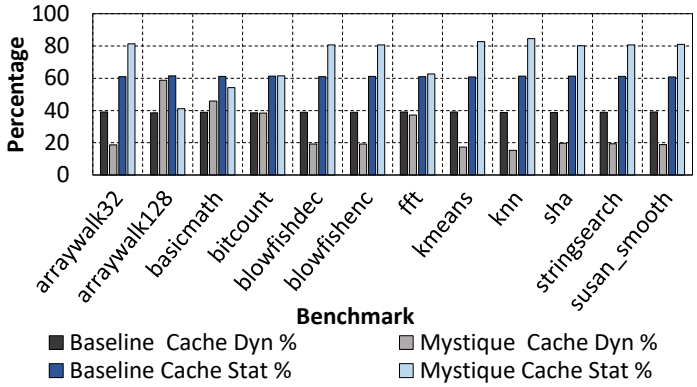


Figure 6.13: Cache dynamic and static energy breakdown for Mystique and the baseline.

be observed from this graph, *Mystique* significantly reduces the execution time of 10 workloads (∼11% on average). The execution times of the remaining two workloads increased negligibly.

Some benchmarks, like `blowfish_decode` and `susan_smooth`, experienced more performance than the others because such benchmarks did not use the cache fully and/or did not have much temporal locality. As a result, when we decreased the cache associativity dynamically to decrease phase-based dynamic and static cache energy consumption, not many additional cache misses were introduced. Therefore, the savings in dynamic and static cache energy (due to less associativity and size) dominated the cache miss energy

L1I & L1D $ Latency	L2 $ Latency	DRAM Latency	Pipeline Stages	$ Data & Tag Lookup
1.55 ns	1.86 ns	14.3 ns	F, D, E, M, W	Parallel

Table 6.5: Baseline configuration.

overheads, ultimately leading to an increase in "forward progress."

Conversely, we have yet to achieve significant performance benefits for benchmarks like KNN and FFT, which utilize the caches fully and/or with high temporal or spatial locality. There was performance degradation in a few of these benchmarks (`arraywalk128`, `basicmath`), as can be seen in Figure 6.10. This degradation in performance happened because when Mystique reconfigured the cache dynamically to smaller associativities (than the baseline configuration), the cache misses significantly increased, causing more costly main memory accesses and dominating any potential dynamic and static energy improvement stemming from reconfiguration.

As can be observed from Table 6.6, Mystique reduced the cache energy consumption by an average of 96.02% for Smart Farm, 93.25% for Incident Monitoring, and 93.68% for Health Monitoring use cases. On average, across all benchmarks, the cache energy usage reduction is 95.93%, as seen in Figures 6.11 and 6.14. This significant reduction in cache energy usage, in turn, reduced the power failures of Smart Farm by 31.85%, Incident Monitoring by 25.65%, and Health Monitoring by 24.83% (see Table 6.7). On average, for all the benchmarks tested, the reduction in power failures was 28.10% (Figure 6.12). This reduction in the number of power failures was reflected as speedup, by 13.52% in Smart Farm, 9.25% in Incident Monitoring, and 9.17% in Health Monitoring use-cases, as can be observed from Table 6.8. The average speedup improvement when considering all benchmarks was 10.71%, as seen in Figure 6.10. This increase in performance is due to more "forward progress," which reflects less dynamic cache energy consumption, as seen in Figure 6.13. Even though dynamically downsizing the cache introduces extra cache misses and accesses to the main memory, the end-to-end benchmark speedups dominated the latencies caused by these additional cache misses.

Workflow	**Cache Energy Reductions (%)**
Smart Farm	96.02
Incident Monitoring	93.25
Health Monitoring	93.68

Table 6.6: Average of cache energy reduction for different IoT workflows.

Workflow	**Power Failure Reductions (%)**
SmartFarm	31.85
Incident Monitoring	25.65
Health Monitoring	24.83

Table 6.7: Average of power failure reductions for different IoT workflows.

Workflow	Speedups (%)
SmartFarm	13.52
Incident Monitoring	9.25
Health Monitoring	9.17

Table 6.8: Average of speedups for different IoT workflows.

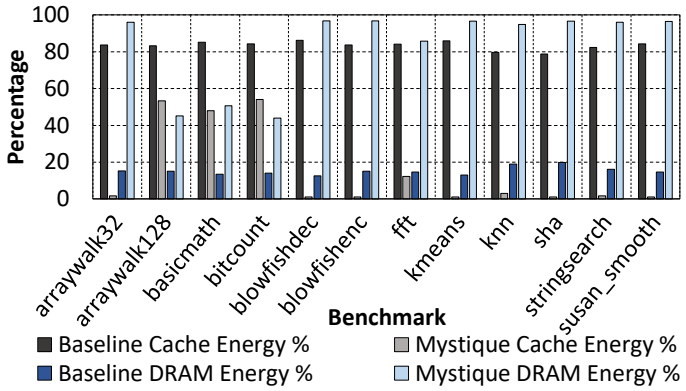


Figure 6.14: Cache and DRAM energy consumption of overall processor energy.

End-to-End Energy Consumption: The overall end-to-end energy consumption remained the same because the increase in cache misses and energy-hungry accesses to the main memory after the cache reconfiguration was balanced out with the significant reduction of the cache energy consumption. As seen in Figure 6.14, even though the cache energy consumption has decreased significantly using Mystique, the DRAM energy consumption has increased, showing this balance. End-to-end energy consumption is not critical as a design metric for Mystique as we are not constrained by a limited energy storage device such as a battery as the only power source for our processor. However, a trickle of intermittent energy must be consumed for intermittent and practical forward progress instead of losing efficiency to energy storage and leakage. Furthermore, due to the significant reduction in instantaneous cache dynamic and static energy consumption, causing a significant reduction in instantaneous energy consumption in different energy-critical phases, the number of power failures decreased, allowing more "forward progress."
Accuracy Evaluation: We evaluated the accuracy of our phase predictor, energy utilization predictor, and energy harvesting predictor, and the results are presented in Figure 6.15 and Table 6.9. To evaluate the accuracy of our phase predictor, we compared the actual phase values with the predicted ones for each benchmark. We marked a prediction as "accurate" if a predicted value is *within* one-phase index of the actual phase value. We used the "one-phase index" as our margin because reconfiguring the cache one

index below or above the actual phase value reflects negligibly on the performance and energy savings compared to 2 or 3-index distances. To evaluate our energy utilization and harvesting predictors, we checked whether each predicted value was within one standard deviation of the actual energy value at that index.

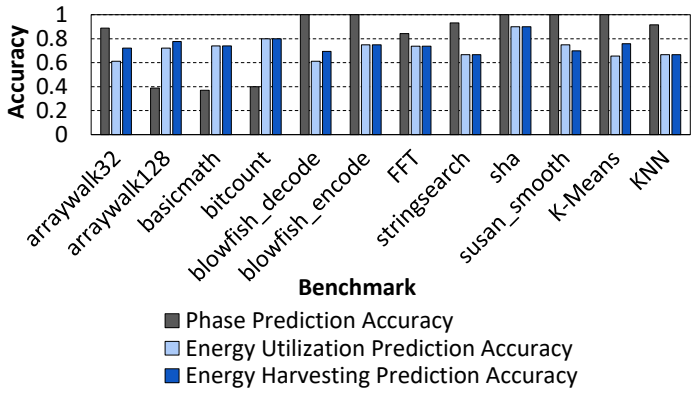


Figure 6.15: Accuracies of predictors for each benchmark.

Predictor	**Cumulative Accuracy (%)**
Phase Prediction	81.17
Energy Utilization Prediction	71.75
Energy Harvesting Prediction	74.28

Table 6.9: Cumulative phase and energy predictor accuracies for all benchmarks.

As can be observed from Figure 6.15, benchmarks `arraywalk32`, `blowfish_decode`, `blowfish_encode`, `FFT`, `stringsearch`, `sha`, `susan_smooth`, `K-Means`, and `KNN` had high phase prediction accuracies (above ∼80%). However, for other benchmarks like `arraywalk128`, `basicmath`, and `bitcount`, the phase prediction accuracies were lower because the phase values fluctuated dramatically for them, which contributed to an unpredictable phase behavior. The low accuracy of the energy predictors in some benchmarks (`blowfish_decode`, `stringsearch`, and `KNN`) is because the energy utilization and/or the harvested energy have fluctuated significantly at each phase. Our predictors could not capture complicated fluctuation behavior, as they are "low-power" predictors. If our predictors were to capture this behavior better, e.g., by employing pattern-based phase prediction algorithms [69], the energy savings that contribute to the forward progress would be significantly less, because they use an extra level of a lookup table for the patterns that will significantly increase the phase prediction energy overhead. For phase prediction, the cumulative accuracy among all of the benchmarks was 81.17%

(see Table 6.9); for energy utilization prediction, it was 71.75%; and for energy harvesting prediction, it was 74.28%.

To further examine the accuracy of our system visually, we created "cache reconfiguration index graphs." The overlaps of the cache configuration indices between our predictor and Oracle executions for the `stringsearch` benchmark are shown in Figure 6.16. In our benchmarks, Mystique's cache reconfiguration indices mainly overlap with the *Minimum Execution Time* and *Maximum ICache Hits* Oracle policies. We only show the String Search benchmark result as a representative workload.

Mystique does not favor any stable system state. The predictor is conservative and amenable to sudden changes because our weighting approach is linear, not exponential. As seen from Table 6.10, Mystique outputs, there is a wide range of cache associativities for most workloads.

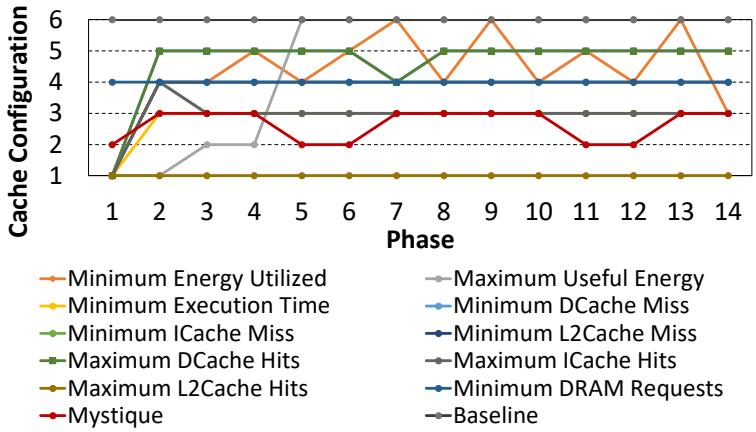


Figure 6.16: Phase reconfiguration indices for different policies of the String Search benchmark.

Oscillation Challenge: During the development of Mystique, when we observed the cache reconfiguration index graphs, we noted an oscillation problem between higher and lower cache configurations. This oscillation created significant energy and performance penalties compared to our baseline due to excessive cache misses introduced when the cache associativity was decreased radically and then increased immediately. This behavior is because our system dynamically reconfigured the caches using application phase prediction that solely used the **last** seen phase value ($mem/\mu Op$). Therefore, there were cache misses due to caches being too small when the cache was reconfigured to smaller associativity and spatial cache locality reduction due to the cache being too large when the last seen phase value was on the opposite end of our index categorizations. We *improved* our predictor and solved the oscillation issue by inputting the energy

Workload	Min L1I&L1D	Min L2	Max L1I&L1D	Max L2
`arraywalk32`	2	1	4	2
`arraywalk128`	2	1	16	16
`basicmath`	2	1	16	16
`bitcount`	2	1	16	16
`blowfishdec`	2	1	2	1
`blowfishenc`	2	1	2	1
`fft`	2	1	16	16
`kmeans`	2	1	2	1
`knn`	2	1	8	4
`sha`	2	1	2	1
`stringsearch`	2	1	4	2
`susan_smooth`	2	1	2	1

Table 6.10: Minimum and maximum cache associativities Mystique outputs for each workload.

prediction information and the weighted average of last seen h (GPHT depth) phase values (historical prediction), as mentioned in Section 6.4. This improvement allowed a smoother and optimal cache reconfiguration strategy, solving both problems associated with cache misses.

6.7 Related Work

To our knowledge, *none* of the prior works have focused on the "dynamic reconfiguration of hardware" in EH NVP-based environments. However, prior works have been in related areas such as phase prediction, dynamic hardware reconfiguration, NVPs, smart farming, incident monitoring, and health monitoring.

Krintz et al. [95] propose a design based on a hybrid cloud approach to smart farming and an on-farm analytics appliance. To operate fire departments properly, Lloret et al. [107] use WSNs to detect and verify fires in forests and rural areas. Their scalable multi-sensor network can sense fire by infrared radiation and smoke and send an alarm along with real-time images [107]. Smeaton et al. [175] demonstrate that audio data can significantly aid video data analysis for incident monitoring.

AlarmNet [206] is a WSN for analyzing circadian activity rhythms (CARs) in assisted living and residential monitoring settings. Moreover, since annotating sensor datasets is quite challenging, unsupervised activity recognition models have recently been proposed [64]. We want to emphasize that the WSN designs above have significant room for improvement due to their continuous "forward progress" requirements.

There exist several prior phase prediction strategies [31]. In Basic Block Vectors (BBV), a phase change happens when the Manhattan Distance between two BBVs exceeds a threshold. In Working Set Signatures (WSS) [31], a program phase is defined by thresholding the relative distance between WSSs. In the Conditional Branch Counter predictor, a *phase change* is defined as the "difference" in branch counts of any consecutive intervals beyond a threshold [31]. In the Global Phase History Table Predictor work [69], an on-the-fly phase prediction technique is proposed, where the phase prediction is performed by using the $Memory\_transactions/\mu Op\_Granularity\,(mem/\mu Op)$ metric [69]. A Global Phase History Table (GPHT) is used to predict the phase, which observes the patterns from previous sample phase values to deduce the next phase behavior [69]. Phase prediction has also been used to *reconfigure* processors [52].

In [13], the authors use a phase detection algorithm to dynamically reconfigure the memory hierarchy in processors. In Spendthrift [113], a predictor-driven scheme for dynamically allocating future power budgets between frequency and resource scaling is developed. In [93], a Coarse-Grained Reconfigurable Architecture is introduced, which leverages the variable usage of functional units according to application phases. However, CGRAs must be more generic to accommodate the variable application domains of IOT WSNs and multiprocessing that a multi-core NVP could accommodate. However, portions of this work can be applied to CGRAs by, for example, reconfiguring the CGRA memory dynamically.

Even though there have been many works on dynamic processor reconfiguration using phase prediction, these works only focus on "energy minimization" and do not consider the possibility of an energy outage, contrary to Mystique, which operates under varying/unreliable energy. So, instead of minimizing energy utilization, Mystique proposes "adaptation" to "energy availability." Also, there has been a significant amount of work related to dynamic cache reconfiguration [22, 201, 213]; however, these works need to consider energy environment information while performing the proposed reconfigurations. Similarly, there are works on VLIW processor resource reconfiguration [8, 61], but they do not apply to our single-core, single-issue system.

6.8 Summary and Future Work

In this chapter, we harmonize application-specific memory-boundedness and energy environment predictions to reconfigure the cache associativities on the fly. This dynamic cache reconfiguration increases the performance of several IoT benchmarks by $\sim$11%,

decreasing the cache energy consumption by ∼96%, and decreasing the power failure states by ∼28%. This harmonized system is a solution to the general-purpose gateways because it increases energy efficiency, leading to more application forward progress.

This work can be extended to incorporate main memory reconfiguration, which can be explored in multiple dimensions. A straightforward way is to allow using a standard DRAM with reconfiguration potentials. These reconfigurations can span from changing the refresh rate, clock speed, or memory capacity to utilizing different DRAM technologies, like LPDDR or EDRAM, to augment energy savings. Moreover, we can utilize a "hybrid version" using STT-MRAM (or any other persistent memory technology) with a standard DRAM (with the optimizations above) to obtain better performance-energy tradeoffs. Moreover, we will extend this work to leverage low-power ML kernels to perform phase and energy behavior predictions to increase accuracy. Lastly, we use simultaneous tag-data lookup in this work, and investigating the correlation between energy harvesting patterns and different tag-data lookup options is an exciting extension.

Chapter 7
Conclusions

7.1 Summary of Contributions

The demand for computational power has significantly grown in the last several decades. However, not many advancements target solving this problem from different angles. This dissertation fills this gap and proposes adaptive optimization techniques to optimize performance and resource management for HPC applications from application characterization, dynamic phase characterization and architecture reconfiguration, intelligent job scheduling, compiler-guided strategy, and dataset augmentation perspectives.

We first propose a Machine Learning Aided Architectural Bottleneck Analysis Model, MLAM, aimed at robustly inferring bottlenecks from profiling data and identifying mathematical relationships between architectural bottlenecks and performance metrics for application characterization. Our model achieves above 70% accuracy in inferring bottlenecks for single architecture and multi-architecture (knowledge transfer) cases.

In the second part, we present a cache reconfiguration scheme that dynamically uses the microarchitectural behavior of application phases to optimize forward progress and instantaneous energy utilization of the phase. We obtain a ∼69% reduction in dynamic cache energy utilization, which reduces the power failures in EH NVPs by ∼47%, reflecting a ∼6% increase in performance.

In the third part, we propose an intelligent GPU job-scheduling policy, GYAN, for multi-GPU infrastructures to ensure efficient resource allocation. We specifically target the Galaxy platform since many GPU-supported tools have been developed to be executed within this platform. GYAN uses instantaneous GPU memory usage information for optimal performance and leads to ∼2× improvement in performance for the Racon tool and ∼50× improvement for the Bonito base calling tool.

In the fourth part, we propose a compiler-guided strategy for recomputation optimization in multi-threaded applications and a compiler-guided cache replacement strategy to

optimize performance. Our recomputation across threads strategy improves performance by ∼13.25% over the conventional optimizations that do not use recomputation and 7.68% over a single-thread centric recomputation. The corresponding improvements when employing recomputation-conscious caching are 19.12% and 11.63%, respectively.

7.2 Future Research Directions

As architectures evolve and the need for computational power increases with the popularity of compute-intensive applications, analyzing and characterizing popular applications has been of paramount interest. Since companies are already improving their products, especially for higher compute capabilities and more extensive data, using these improvements efficiently depends on improved resource allocation and performance optimizations. For this, (1) having a bottleneck analysis tool that is trained on recent and popular AI workloads (such as LLMs), (2) that is also trained on profiling data of applications that are executed on state-of-the-art architectures such as newer GPUs, (3) using lower power machine learning kernels to predict phase behavior, and (4) increasing job scheduling intelligence, especially for hosts that have GPU support is essential.

7.2.1 Training MLAM with Popular AI workloads

In Chapter 3, we proposed MLAM that aids in predicting bottlenecks for SPEC CPU, ML, and DL benchmarks. However, the benchmarks that MLAM is trained on do not include state-of-the-art popular AI workloads like LLMs. Training MLAM for newer workloads to improve user experience by examining their architectural behavior and finding resource optimization areas would be crucial.

7.2.2 Adapting MLAM to GPU Architectures

MLAM is only trained on CPU architectures. Specifically, it is trained in Intel Skylake, Ice Lake, Cascadelake, and Broadwell architectures. To quickly transfer bottleneck knowledge to other architectures, MLAM should also be trained on emerging architectures, GPUs, and newer CPUs. This approach will improve user experience, as the bottleneck value and ordering prediction accuracy would be much higher.

7.2.3 Increasing the Intelligence of GYAN

In Chapter 4, we propose using instantaneous memory consumption of GPU processes to schedule upcoming jobs to less busy GPUs. We could also use historical job information and AI to better predict GPU SM utilization. Moreover, we could use time series estimations for more optimal scheduling.

7.2.4 Using Low-Power ML Kernels and Compiler Analysis to Predict Phase Behavior for NVPs

The current version of Mystique proposed in Chapter 6 uses a basic phase predictor that employs weighted historical averaging for the last seen **h** phases. The accuracy can be improved significantly if the older history of applications is also taken into account, code analysis is performed to find regions of interest for memory intensity, and this information is input to an ML model to make more informative predictions. One thing to emphasize here is that compiler analysis and ML kernel should be chosen intelligently to reduce power consumption so that they do not hinder forward progress.

7.2.5 Inspired by this Dissertation: Performance Optimization for Surrogate Models

Since HPC applications are becoming more compute-heavy, some (e.g. simulations), are replaced with surrogate models [7, 54, 189]. Surrogate models deal with vast amounts of data, decreasing performance. We target the "performance and resource management optimization of HPC applications" goal from another angle by proposing input dataset modification for HPC applications, specifically for atomistic system simulation surrogate deep neural networks (DNNs), to optimize performance and accuracy.

Data pruning is a prevalent method to improve the performance of ML/DL models [6, 37, 118, 148, 171, 176, 177, 190, 195]. It is crucial in deploying large DL models on resource-constrained devices or real-time applications where execution time is vital. We propose creating OOD dataset instances to prune unneeded data points and increase the performance of atomistic simulation surrogate DNNs. Our method improves performance up to 2× when the original test dataset and varying training datasets are used and 3× when the training dataset is fixed, and the test dataset is varied, with negligible accuracy degradation. This proposal is inspired by this dissertation, as it continues the full-stack optimization from a software angle, similar to Chapter 3.

Bibliography

[1] Advanced Micro Devices, Inc. Advance Data Center AI with Servers Powered by AMD EPYC™ Processors. `https://www.amd.com/en/products/processors/server/epyc/ai.html`. Accessed: 2024-09-17.

[2] Enis Afgan, Dannon Baker, Nate Coraor, Brad Chapman, Anton Nekrutenko, and James Taylor. Galaxy CloudMan: Delivering cloud compute clusters. *BMC Bioinformatics*, 11:S4, 6 pages, 2010.

[3] Ismail Akturk and Ulya R. Karpuzcu. AMNESIAC: Amnesic Automatic Computer. *ACM SIGPLAN Notices*, 52(4):811–824, April 2017.

[4] Ismail Akturk and Ulya R. Karpuzcu. Trading Computation for Communication: A Taxonomy of Data Recomputation Techniques. *IEEE Transactions on Emerging Topics in Computing*, 9(1):496–506, 2021.

[5] André Altmann, Laura Tolosi, Oliver Sander, and Thomas Lengauer. Permutation Importance: A Corrected Feature Importance Measure. *Bioinformatics*, 26(10):1340–1347, 2010.

[6] Anelia Nedelcheva Angelova. Data pruning. Master's thesis, California Institute of Technology, 2004.

[7] Claudio Angione, Eric Silverman, and Elisabeth Yaneske. Using machine learning as a surrogate model for agent-based simulations. *Plos One*, 17(2):e0263150, 2022.

[8] Fakhar Anjam, Stephan Wong, Luigi Carro, Gabriel L. Nazar, and Mateus B. Rutzig. Simultaneous reconfiguration of issue-width and instruction cache for a VLIW processor. In *Proceedings of the International Conference on Embedded Computer Systems (IC-SAMOS)*, pages 183–192, Samos, Greece, 2012. IEEE.

[9] Apple, Inc. Apple introduces M4 chip. `https://www.apple.com/newsroom/2024/05/apple-introduces-m4-chip/`. Accessed: 2024-09-17.

[10] Anton Arkhipov, Jana Hüve, Martin Kahms, Reiner Peters, and Klaus Schulten. Continuous fluorescence microphotolysis and correlation spectroscopy using 4Pi microscopy. *Biophysical Journal*, 93(11):4006–4017, 2007.

[11] Armbian Contributors. Armbian: Linux for arm development boards. `https://www.armbian.com/`. Accessed: 2024-09-17.

[12] Kazi Asifuzzaman, Rommel Sánchez Verdejo, and Petar Radojković. Performance and power estimation of STT-MRAM main memory with reliable system-level simulation. *ACM Transactions on Embedded Computing Systems*, 21(1), January 2022.

[13] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO-33, pages 245–257, New York, NY, USA, 2000. Association for Computing Machinery.

[14] Saambhavi Baskaran, Mahmut Taylan Kandemir, and Jack Sampson. An architecture interface and offload model for low-overhead, near-data, distributed accelerators. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture*, MICRO-55, pages 1160–1177, Chicago, Illinois, 2022. IEEE.

[15] Martin Bauer, Sebastian Eibl, Christian Godenschwager, Nils Kohl, Michael Kuron, Christoph Rettinger, Florian Schornbaum, Christoph Schwarzmeier, Dominik Thönnes, Harald Köstler, and Ulrich Rüde. waLBerla: A block-structured high-performance framework for multiphysics simulations. *Computers & Mathematics with Applications*, 81:478–501, 2021. Development and Application of Open-source Software for Problems with Numerical PDEs.

[16] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, August 2011.

[17] Avishek Biswas and Anantha P. Chandrakasan. Conv-RAM: An energy-efficient SRAM with embedded convolution computation for low-power CNN-based machine learning applications. In *Proceedings of the IEEE International Solid-State Circuits Conference - (ISSCC)*, pages 488–490, 2018.

[18] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[19] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '92, pages 311–321, New York, NY, USA, 1992. Association for Computing Machinery.

[20] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques

Grobler, Robert Layton, J. Vanderplas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. *ArXiv*, abs/1309.0238:1–15, 2013. `http://arxiv.org/abs/1309.0238`, Accessed: 2024-09-26.

[21] Cerebras Systems, Inc. Wafer-Scale Engine 3: The Largest Chip Ever Built. `https://8968533.fs1.hubspotusercontent-na1.net/hubfs/8968533/Datasheets/WSE-3%20Datasheet.pdf`. Accessed: 2024-09-17.

[22] Shounak Chakraborty and Magnus Själander. WaFFLe: Gated Cache-Ways with Per-Core FIne-Grained DVFS for Reduced On-Chip Temperature and LeAkage Consumption. *ACM Transactions on Architecture and Code Optimization*, 18(4), September 2021.

[23] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.

[24] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A Machine-Learning Supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '14, pages 609–622. IEEE, 2014.

[25] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014. `https://arxiv.org/abs/1410.0759`, Accessed: 2024-09-26.

[26] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. EMNIST: Extending MNIST to handwritten letters. In *Proceedings of the International Joint Conference on Neural Networks*, IJCNN '17, pages 2921–2926, Anchorage, AK, USA, 2017. IEEE.

[27] Alexei Colin, Emily Ruppel, and Brandon Lucia. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 767–781, New York, NY, USA, 2018. Association for Computing Machinery.

[28] William J. Dally, James Balfour, David Black-Shaffer, James Chen, R. Curtis Harting, Vishal Parikh, Jongsoo Park, and David Sheffield. Efficient embedded computing. *Computer*, 41(7):27–32, July 2008.

[29] Ercan M Dede, J Lee, and T Nomura. *Multiphysics Simulation: Electromechanical System Applications and Optimization (Simulation Foundations, Methods and Applications)*. Springer, London, UK, 2014.

[30] Willem de Koning, Milad Miladi, Saskia Hiltemann, Astrid Heikema, John P Hays, Stephan Flemming, Marius van den Beek, Dana A Mustafa, Rolf Backofen, Björn Grüning, and Andrew P Stubbs. NanoGalaxy: Nanopore long-read sequencing data analysis in Galaxy. *GigaScience*, 9(10), 2020. Article no. giaa105, 7 pages.

[31] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-36, pages 217–227, San Diego, CA, USA, 2003. IEEE.

[32] Jack Dongarra and Alexey L Lastovetsky. *High Performance Heterogeneous Computing*. John Wiley & Sons, Hoboken, NJ, USA, 2009.

[33] Jack Doweck, Wen-Fu Kao, Allen Kuan yu Lu, Julius Mandelblat, Anirudha D. Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro*, 37:52–62, 2017.

[34] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '16, pages 523–535, 2016.

[35] Elecrow. HC-12 Wireless Serial Port Communication Module User Manual, Version 1.18. `https://www.elecrow.com/download/HC-12.pdf`, 2012. Accessed: 2024-09-17.

[36] Liew Hui Fang, Syed Idris Syed Hassan, Rosemizi Abd Rahim, Muzamir Isa, and Baharuddin bin Ismail. Exploring piezoelectric for sound wave as energy harvester. *Energy Procedia*, 105:459–466, 2017. 8th International Conference on Applied Energy, ICAE2016, 8-11 October 2016, Beijing, China.

[37] Vitaly Feldman and Chiyuan Zhang. What neural networks memorize and why: Discovering the long tail via influence estimation. In *Advances in Neural Information Processing Systems*, volume 33 of *NIPS '20*, pages 2881–2891, 2020.

[38] K. Flautner, Nam Sung Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 148–157, Anchorage, AK, USA, 2002. IEEE.

[39] Hironobu Fujiyoshi, Tsubasa Hirakawa, and Takayoshi Yamashita. Deep Learning-based Image Recognition for Autonomous Driving. *IATSS Research*, 43(4):244–252, 2019.

[40] Karthik Ganesan, Joshua San Miguel, and Natalie Enright Jerger. The what's next intermittent computing architecture. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pages 211–223, Washington, DC, USA, 2019. IEEE.

[41] Eva García-Martín, Crefeda Faviola Rodrigues, Graham Riley, and Håkan Grahn. Estimation of energy consumption in machine learning. *Journal of Parallel and Distributed Computing*, 134:75–88, 2019.

[42] N. Gershenfeld, J. Kymissis, C. Kendall, and J. Paradiso. Parasitic power harvesting in shoes. In *Proceedings of the 16th International Symposium on Wearable Computers*, pages 132–139, Los Alamitos, CA, USA, October 1998. IEEE Computer Society.

[43] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, New York, NY, USA, 1998. Association for Computing Machinery.

[44] M. Giles, E. Lazlo, I. Reguly, J. Appleyard, and J. Demouth. GPU implementation of finite difference solvers. In *Proceedings of the 7th Workshop on High Performance Computational Finance*, WHPCF '14, pages 1–8, 2014.

[45] Mark Giles. An Introduction to GPU Programming. `https://people.maths.ox.ac.uk/gilesm/old/pp10/lec2_2x2.pdf`, 2010. Accessed: 2024-09-17.

[46] Jeremy Goecks, Anton Nekrutenko, and James Taylor. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11, 2010. Article no. R86, 13 pages.

[47] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: The terasys massively parallel pim array. *Computer*, 28(4):23–31, April 1995.

[48] Google Cloud. Introducing Google Axion Processors, our new Arm-based CPUs. `https://cloud.google.com/blog/products/compute/introducing-googles-new-arm-based-cpu`. Accessed: 2024-09-17.

[49] LLC Grain Cleaning. Fan powered grain cleaner. `https://gcsgraincleaner.com/product_categories/fan-powered-grain-cleaner/`, 2023. Accessed: 2024-09-17.

[50] Björn Grüning, Ryan Dale, Andreas Sjödin, Brad A. Chapman, Jillian Rowe, Christopher H. Tomkins-Tinch, Renan Valieris, ..., Daniela Beisser, ..., and Johannes Köster. Bioconda: sustainable and comprehensive software distribution for the life sciences. *Nature Methods*, 15(7):475–476, July 2018.

[51] Gulsum Gudukbay. gulsumgudukbay/racon_dockerfile. https://hub.docker.com/r/gulsumgudukbay/racon_dockerfile. Accessed: 2024-09-17.

[52] Qi Guo, Anderson Sartor, Anthony Brandon, Antonio C. S. Beck, Xuehai Zhou, and Stephan Wong. Run-time phase prediction for a reconfigurable VLIW processor. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition*, DATE '16, pages 1634–1639, Dresden, Germany, 2016. IEEE.

[53] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth IEEE International Workshop on Workload Characterization, WWC-4*, WWC '01, pages 3–14, USA, 2001. IEEE Computer Society.

[54] Ehsan Haghighat, Maziar Raissi, Adrian Moure, Hector Gomez, and Ruben Juanes. A physics-informed deep learning framework for inversion and surrogate modeling in solid mechanics. *Computer Methods in Applied Mechanics and Engineering*, 379:Article no. 113741, 22 pages, 2021.

[55] David J Hardy, John E Stone, and Klaus Schulten. Multilevel summation of electrostatic potentials using graphics processing units. *Parallel Computing*, 35(3):164–177, 2009.

[56] Dion Harris. Amped Up: HPC Centers Ride A100 GPUs to Accelerate Science. https://blogs.nvidia.com/blog/2020/05/15/hpc-supercomputers-a100-gpus/, 2020. Accessed: 2024-09-17.

[57] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '16, pages 770–778, 2016.

[58] J.L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, 33(7):28–35, 2000.

[59] Torsten Hoefler, William Gropp, William Kramer, and Marc Snir. Performance Modeling for Systematic Performance Tuning. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 1–12, Seattle, WA, USA, 2011. IEEE.

[60] Changwan Hong, Wenlei Bao, Albert Cohen, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. Effective Padding of Multidimensional Arrays to Avoid Cache Conflict Misses. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 129–144, New York, NY, USA, 2016. Association for Computing Machinery.

[61] Sensen Hu and Jing Haung. Exploring adaptive cache for reconfigurable vliw processor. *IEEE Access*, 7:72634–72646, 2019.

[62] C. Hung and C. Y. Tang. Bioinformatics tools with deep learning based on GPU. In *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine*, BIBM'17, pages 1906–1908, 2017.

[63] Tim Hwang. Computational Power and the Social Impact of Artificial Intelligence. *CoRR*, abs/1803.08971, 2018. `http://arxiv.org/abs/1803.08971`, Accessed: 2024-09-26.

[64] Ayokunle Olalekan Ige and Mohd Halim Mohd Noor. A survey on unsupervised learning for wearable sensor-based activity recognition. *Applied Soft Computing*, 127, 2022. Article no. 109363, 22 pages.

[65] Intel Corp. Intel® Gaudi® AI Accelerators. `https://www.intel.com/content/www/us/en/products/details/processors/ai-accelerators/gaudi-overview.html`. Accessed: 2024-09-17.

[66] Intel Corp. Intel Performance Monitoring Units (PMUs). `https://perfmon-events.intel.com/`, 2022. Accessed: 2024-09-17.

[67] Intel Corp. Intel VTune Profiler. `https://software.intel.com/en-us/vtune`, 2022. Accessed: 2024-09-17.

[68] IoT Solutions World Congress. IoT transforming the future of agriculture. `https://www.iotsworldcongress.com/iot-transforming-the-future-of-agriculture/`, 2022. Accessed: 2024-09-17.

[69] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-39, pages 359–370, Orlando, FL, USA, 2006. IEEE.

[70] Aaron D Jaggard, Swara Kopparty, Vijay Ramachandran, and Rebecca N Wright. The design space of probing algorithms for network-performance measurement. *Sigmetrics*, 41(1):105–116, 2013.

[71] Kevin Jamieson and Ameet Talwalkar. Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, AISTAT '16, pages 240–248, 2016.

[72] Prem Prakash Jayaraman, Ali Yavari, Dimitrios Georgakopoulos, Ahsan Morshed, and Arkady Zaslavsky. Internet of things platform for smart farming: Experiences and lessons learnt. *Sensors*, 16(11), 2016. Article no. 1884, 17 pages.

[73] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. Association for Computing Machinery.

[74] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[75] Kaggle. Home credit default risk. `https://www.kaggle.com/c/home-credit-default-risk`, 2019. Accessed: 2024-09-17.

[76] Kaggle. New York City Taxi Trip Duration. `https://www.kaggle.com/c/nyc-taxi-trip-duration/data`, 2018. Accessed: 2024-09-17.

[77] Kaggle. GTZAN Dataset - Music Genre Classification. `https://www.kaggle.com/andradaolteanu/gtzan-dataset-music-genre-classification`, 2020. Accessed: 2024-09-17.

[78] Kaggle. Toxic Comment Classification Challenge. `https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge/data`, 2018. Accessed: 2024-09-17.

[79] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramenujarn. A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):115–135, 1999.

[80] Mahmut Taylan Kandemir, Jihyun Ryoo, Xulong Tang, and Mustafa Karakoy. Compiler support for near data computing. In *Proceedings of the 26th ACM*

SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 90–104, New York, NY, USA, 2021. Association for Computing Machinery.

[81] Sergey Karayev, Matthew Trentacoste, Helen Han, Aseem Agarwala, Trevor Darrell, Aaron Hertzmann, and Holger Winnemoeller. Recognizing image style. In *Proceedings of the British Machine Vision Conference*, pages 1–11, Nottingham, England, 2014. BMVA Press.

[82] Anna Katharina Hildebrandt, Daniel Stockel, Nina Fischer, Luis de la Garza, Jens Kruger, Stefan Nickels, Marc Rottig, Charlotta Scharfe, Marcel Schumann, Philipp Thiel, Hans-Peter Lenhof, Oliver Kohlbacher, and Andreas Hildebrandt. Ballaxy: Web services for structural bioinformatics. *Bioinformatics*, 31(1):121–122, 2015.

[83] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 240–251, New York, NY, USA, 2001. Association for Computing Machinery.

[84] Onur Kayıran, Adwait Jog, Mahmut T. Kandemir, and Chita R. Das. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 157–166, Edinburgh, Scotland, UK, 2013. IEEE Computer Society Press.

[85] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 17th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, NAACL-HLT 2019, pages 4171–4186, 2019.

[86] Shwet Ketu and Pramod Kumar Mishra. Cloud, fog and mist computing in IoT: an indication of emerging opportunities. *IETE Technical Review*, 39(3):713–724, 2022.

[87] David E Keyes, Lois C McInnes, Carol Woodward, William Gropp, Eric Myra, Michael Pernice, John Bell, Jed Brown, Alain Clo, Jeffrey Connors, Emil Constantinescu, Don Estep, Kate Evans, Charbel Farhat, Ammar Hakim, Glenn Hammond, Glen Hansen, Judith Hill, Tobin Isaac, Xiangmin Jiao, Kirk Jordan, Dinesh Kaushik, Efthimios Kaxiras, Alice Koniges, Kihwan Lee, Aaron Lott, Qiming Lu, John Magerlein, Reed Maxwell, Michael McCourt, Miriam Mehl, Roger Pawlowski, Amanda P Randles, Daniel Reynolds, Beatrice Rivière, Ulrich Rüde, Tim Scheibe, John Shadid, Brendan Sheehan, Mark Shephard, Andrew Siegel, Barry Smith, Xianzhu Tang, Cian Wilson, and Barbara Wohlmuth. Multiphysics simulations: Challenges and opportunities. *The International Journal of High Performance Computing Applications*, 27(1):4–83, 2013.

[88] Soheil Khadirsharbiyani, Jagadish Kotra, Karthik Rao, and Mahmut Kandemir. Data Convection: A GPU-Driven Case Study for Thermal-Aware Data Placement in 3D DRAMs. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1), February 2022. Article no. 7, 25 pages.

[89] Huesung Kim, A. K. Somani, and A. Tyagi. A Reconfigurable Multifunction Computing Cache Architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):509–523, August 2001.

[90] Orhan Kislal, Jagadish Kotra, Xulong Tang, Mahmut Taylan Kandemir, and Myoungsoo Jung. Enhancing Computation-to-Core Assignment with Physical Location Information. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 312–327, New York, NY, USA, 2018. Association for Computing Machinery.

[91] H. Koc, M. Kandemir, E. Ercanli, and O. Ozturk. Reducing Off-Chip Memory Access Costs Using Data Recomputation in Embedded Chip Multi-Processors. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 224–229, New York, NY, USA, 2007. Association for Computing Machinery.

[92] H. Koc, O. Ozturk, M. Kandemir, S. H. K. Narayanan, and E. Ercanli. Minimizing Energy Consumption of Banked Memories Using Data Recomputation. In *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED '06, pages 358–362, New York, NY, USA, 2006. Association for Computing Machinery.

[93] G. Korol, M. Jordan, R. S. Silva, M. M. Pereira, M. Brandalero, M. B. Rutzig, and A. C. S. Beck. A runtime power-aware phase predictor for CGRAs. In *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, Cancun, Mexico, 2019. IEEE.

[94] J. B. Kotra, H. Zhang, A. R. Alameldeen, C. Wilkerson, and M. T. Kandemir. CHAMELEON: A Dynamically Reconfigurable Heterogeneous Memory System. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 533–545, October 2018.

[95] C. Krintz, R. Wolski, Nevena Golubovic, Benji Lampel, V. Kulkarni, Barbara Balaji Sethuramasamyraja, and Bruce Roberts. SmartFarm: Improving agriculture sustainability using modern information technology. In *Proceedings of the 22nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1–5, New York, NY, USA, 2016. Association for Computing Machinery.

[96] Chang-Jung Ku, Ching-Wen Chen, An Hsia, and Chun-Lin Chen. Linked instruction caches for enhancing power efficiency of embedded systems. *Microprocessors and Microsystems*, 38(3):197–207, 2014.

[97] A. Kumar. Phase-based Reconfiguration of Level One Cache for Single-core Processors without Affecting Second Level Cache. In *Proceedings of the Communication, Control and Intelligent Systems*, CCIS '15, pages 421–426, November 2015.

[98] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, pages 75–86, USA, 2004. IEEE Computer Society Press.

[99] J. Lau, S. Schoenmackers, and B. Calder. Transition Phase Classification and Prediction. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 278–289, 2005.

[100] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. `http://yann.lecun.com/exdb/mnist/`, 2010. Accessed: 2024-09-17.

[101] Christian Ledergerber and Christophe Dessimoz. Base-calling for next-generation sequencing platforms. *Briefings in Bioinformatics*, 12(5):489–497, 2011.

[102] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits. *CoRR*, abs/1603.06560, 2016.

[103] Maxwell W. Libbrecht and William Stafford Noble. Machine Learning Applications in Genetics and Genomics. *Nature Reviews Genetics*, 16(6):321–332, 2015.

[104] Jonathan Lifflander and Sriram Krishnamoorthy. Cache locality optimization for recursive programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–16, New York, NY, USA, 2017. Association for Computing Machinery.

[105] Chun-Yi Liu, Jagadish Kotra, Myoungsoo Jung, and Mahmut Taylan Kandemir. Centaur: A Novel Architecture for Reliable, Low-Wear, High-Density 3D NAND Storage. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(2), June 2020. Article no. 28, 25 pages.

[106] Chun-Yi Liu, Yunju Lee, Wonil Choi, Myoungsoo Jung, Mahmut Taylan Kandemir, and Chita Das. GSSA: A Resource Allocation Scheme Customized for 3D NAND SSDs. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, HPCA '21, pages 426–439, Seoul, Korea (South), 2021. IEEE Computer Society Press.

[107] Jaime Lloret, Miguel Garcia-Pineda, Diana Bri, and Sandra Sendra. A wireless sensor network deployment for rural and forest fire detection and verification. *Sensors (Basel, Switzerland)*, 9:8722–47, November 2009.

[108] Qingda Lu, Christophe Alias, Uday Bondhugula, Thomas Henretty, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, P. Sadayappan, Yongjian Chen, Haibo Lin, and Tin-fook Ngai. Data Layout Transformation for Enhancing Data Locality on NUCA Chip Multiprocessors. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 348–357, USA, 2009. IEEE Computer Society Press.

[109] K. Ma, X. Li, S. Li, Y. Liu, J. J. Sampson, Y. Xie, and V. Narayanan. Nonvolatile processor architecture exploration for energy-harvesting applications. *IEEE Micro*, 35(5):32–40, 2015.

[110] K. Ma, X. Li, K. Swaminathan, Y. Zheng, S. Li, Y. Liu, Y. Xie, J. J. Sampson, and V. Narayanan. Nonvolatile processor architectures: Efficient, reliable progress with unstable power. *IEEE Micro*, 36(3):72–83, 2016.

[111] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan. Architecture exploration for ambient energy harvesting nonvolatile processors. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA '15, pages 526–537, Burlingame, CA, USA, 2015. IEEE.

[112] Kaisheng Ma, Xueqing Li, Huichu Liu, Xiao Sheng, Yiqun Wang, Karthik Swaminathan, Yongpan Liu, Yuan Xie, John Sampson, and Vijaykrishnan Narayanan. Dynamic power and energy management for energy harvesting nonvolatile processor systems. *ACM Transactions on Embedded Computing Systems*, 16(4), 2017. Article no. 107, 23 pages.

[113] Kaisheng Ma, Xueqing Li, Srivatsa Rangachar Srinivasa, Yongpan Liu, John Sampson, Yuan Xie, and Vijaykrishnan Narayanan. Spendthrift: Machine learning based resource and frequency scaling for ambient energy harvesting nonvolatile processors. In *Proceedings of the 22nd Asia and South Pacific Design Automation Conference*, ASP-DAC '17, pages 678–683, Chiba, Japan, 2017. IEEE.

[114] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *Proceedings of the13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18), pages 129–144, Carlsbad, CA, October 2018. USENIX Association.

[115] Saaduddin Mahmud. Parallel Programming With CUDA Tutorial (Part-2: Basics). `https://saadmahmud14.medium.com/parallel-programming-with-cuda-tutorial-part-2-96f6eaea2832`, 2018. Accessed: 2024-09-17.

[116] Andrea Manconi, Marco Moscatelli, Matteo Gnocchi, Giuliano Armano, and Luciano Milanesi. A GPU-based high performance computing infrastructure for specialized NGS analyses. *PeerJ Preprints*, 4, 2016. Article no. e2175v1, 3 pages, Accessed: 2024-09-17.

[117] Ariana Manglaviti and Peter Genzer. Brookhaven Lab Named an NVIDIA GPU Research Center. https://www.bnl.gov/newsroom/news.php?a=111821. Accessed: 2024-09-17.

[118] Max Marion, Ahmet Üstün, Luiza Pozzobon, Alex Wang, Marzieh Fadaee, and Sara Hooker. When less is more: Investigating data pruning for pretraining LLMs at scale. *arXiv preprint arXiv:2309.04564*, 2023. https://arxiv.org/abs/2309.04564, Accessed: 2024-09-26.

[119] Richard C Martineau. The MOOSE multiphysics computational framework for nuclear power applications: A special issue of nuclear technology. *Nuclear Technology*, 207(7):iii–viii, 2021.

[120] Atmel Co. AT03663: Power consumption of ZigBee end device. https://ww1.microchip.com/downloads/en/AppNotes/Atmel-42321-Power-Consumption-of-ZigBee-End-Device_ApplicationNote_AT03663.pdf, 2015. Accessed: 2024-09-17.

[121] Docker, Inc. What is a Container? | App Containerization | Docker. https://www.docker.com/resources/what-container. Accessed: 2024-09-17.

[122] Galaxy Project. Docker Integration. https://galaxyproject.org/admin/tools/docker/. Accessed: 2024-09-17.

[123] Galaxy Project. Galaxy 101 - What is Galaxy? https://galaxyproject.org/tutorials/g101/. Accessed: 2024-09-17.

[124] Galaxy Project. Galaxy Project Stats. https://galaxyproject.org/galaxy-project/statistics/. Accessed: 2024-09-17.

[125] Galaxy Project. Galaxy Publication Library. https://galaxyproject.org/publication-library/. Accessed: 2024-09-17.

[126] Galaxy Project. Galaxy Tool XML File. https://docs.galaxyproject.org/en/master/dev/schema.html. Accessed: 2024-09-17.

[127] Illumina, Inc. Next-Generation Sequencing (NGS). https://www.illumina.com/science/technology/next-generation-sequencing.html. Accessed: 2024-09-17.

[128] NVIDIA Corp. High-Performance Computing Products and Solutions. https://www.nvidia.com/en-us/high-performance-computing/. Accessed: 2024-09-17.

[129] NVIDIA Corp. Kepler TM GK110/210: NVIDIA's Next Generation CUDA Compute Architecture. https://www.nvidia.cn/content/dam/en-zz/Solutions/Data-Center/documents/NV-DS-Tesla-KCompute-Arch-May-2012-LR.pdf, 2012. Accessed: 2024-09-17.

[130] NVIDIA Corp. TESLA K80 GPU Accelerator: Board Specification. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/Tesla-K80-BoardSpec-07317-001-v05.pdf, 2015. Accessed: 2024-09-17.

[131] Pacific Biosciences of California, Inc. PacBio Company | About Us. https://www.pacb.com/company/about-us. Accessed: 2024-09-17.

[132] Sylabs.io. Singularity. https://sylabs.io/. Accessed: 2024-09-17.

[133] Andrew Meola. Smart Farming in 2020: How IoT sensors are creating a more efficient precision agriculture industry. https://www.businessinsider.com/smart-farming-iot-agriculture, 2021. Accessed: 2024-09-17.

[134] Cyan Subhra Mishra, Jack Sampson, Mahmut Taylan Kandemir, and Vijaykrishnan Narayanan. Origin: Enabling on-device intelligence for human activity recognition using energy harvesting wireless sensor networks. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, DATE '21, pages 1414–1419, Grenoble, France, 2021. IEEE, IEEE.

[135] Monash University. Raw fast5s. https://bridges.monash.edu/articles/dataset/Raw_fast5s/7676174. Accessed: 2024-09-17.

[136] Ken Montanez. Amazon Access Samples Data Set. https://archive.ics.uci.edu/ml/datasets/Amazon+Access+Samples, 2011. Accessed: 2024-09-17.

[137] Matthias S. Müller, John Baron, William C. Brantley, Huiyu Feng, Daniel Hackenberg, Robert Henschel, Gabriele Jost, Daniel Molka, Chris Parrott, Joe Robichaux, Pavel Shelepugin, Matthijs van Waveren, Brian Whitney, and Kalyan Kumaran. SPEC OMP2012 – an Application Benchmark Suite for Parallel Systems Using OpenMP. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, pages 1–14, Berlin, Heidelberg, 2012. Springer-Verlag.

[138] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-40, pages 3–14, Chicago, IL, USA, 2007. IEEE.

[139] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. Processing data where it makes sense: Enabling in-memory computation. *Microprocessors and Microsystems*, 67:28–41, 2019.

[140] Rajalakshmi Nandakumar, Shyamnath Gollakota, and Nathaniel Watson. Contactless sleep apnea detection on smartphones. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 45–57, New York, NY, USA, 2015. Association for Computing Machinery.

[141] Ansgar Niemöller, Michael Schlottke-Lakemper, Matthias Meinke, and Wolfgang Schröder. Dynamic load balancing for direct-coupled multiphysics simulations. *Computers & Fluids*, 199, 2020. Article no. 104437, 12 pages.

[142] NVIDIA Corp. The Proven Standard for Enterprise AI. `https://www.nvidia.com/en-us/data-center/dgx-platform/`. Accessed: 2024-09-17.

[143] NVIDIA Corp. Nsight Compute. `https://docs.nvidia.com/nsight-compute/NsightCompute/index.html`, 2022. Accessed: 2024-09-17.

[144] NVIDIA Corp. & Aff. Profiler User's Guide. `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`, 2022. Accessed: 2024-09-17.

[145] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5), 2008.

[146] Oxford Nanopore Technologies. `https://nanoporetech.com`. Accessed: 2024-09-17.

[147] PacBio. Alzheimer IsoSeq 2016. `https://downloads.pacbcloud.com/public/dataset/Alzheimer_IsoSeq_2016/`. Accessed: 2024-09-17.

[148] Mansheej Paul, Surya Ganguli, and Gintare Karolina Dziugaite. Deep learning on a data diet: Finding important examples early in training. In *Advances in Neural Information Processing Systems*, volume 34 of *NIPS '21*, pages 20596–20607, 2021.

[149] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011.

[150] Yasset Perez-Riverol. BioContainers. `https://biocontainers-edu.readthedocs.io/en/latest/what_is_biocontainers.html`. Accessed: 2024-09-17.

[151] M.D. Powell, A. Agarwal, T.N. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture*, MICRO-34, pages 54–65, Austin, TX, USA, 2001. IEEE.

[152] Tribuvan Kumar Prakash and Lu Peng. Performance Characterization of SPEC CPU2006 Benchmarks on Intel Core 2 Duo Processor. In *Proceedings of the International Conference on Parallel Processing*, pages 1–6, 2008.

[153] Mukta Punjani. Register rematerialization in GCC. In *Proceedings of the GCC Developers' Summit*, volume 2004, pages 131–139, Ottawa, Ontario, Canada, 2004. GCC Developers' Summit.

[154] Zheng Qiang, Bojing Shi, and Zhong Wang. Recent progress on piezoelectric and triboelectric energy harvesters in biomedical systems. *Advanced Science*, 4, March 2017. Article no. 1700029, 23 pages.

[155] Keni Qiu, Nicholas Jao, Mengying Zhao, Cyan Subhra Mishra, Gulsum Gudukbay, Sethu Jose, Jack Sampson, Mahmut Taylan Kandemir, and Vijaykrishnan Narayanan. ResiRCA: A Resilient Energy Harvesting ReRAM Crossbar-Based Accelerator for Intelligent Embedded Processors. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, HPCA '20, pages 315–327, San Diego, CA, USA, 2020. IEEE.

[156] Gajendra P.S. Raghava. OSDDlinux: A Customized Operating System for Drug Discovery. `http://osddlinux.osdd.net`. Accessed: 2024-09-17.

[157] Raghunathan Ramakrishnan, Pavlo O. Dral, Matthias Rupp, and O. Anatole von Lilienfeld. Big Data Meets Quantum Chemistry Approximations: The Δ-Machine Learning Approach. *Journal of Chemical Theory and Computation*, 11(5):2087–2096, 2015.

[158] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. Reconfigurable Caches and Their Application to Media Processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 214–224, New York, NY, USA, 2000. ACM.

[159] Nishkam Ravi, Nikhil Dandekar, Preetham Mysore, and Michael Littman. Activity recognition from accelerometer data. In *Proceedings of the Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*, volume 3, pages 1541–1546, January 2005.

[160] Austin Reese. Used cars dataset. `https://www.kaggle.com/austinreese/craigslist-carstrucks-data`, 2021. Accessed: 2024-09-17.

[161] Christopher I Rodrigues, David J Hardy, John E Stone, Klaus Schulten, and Wen-Mei W Hwu. GPU acceleration of cutoff pair potentials for molecular modeling applications. In *Proceedings of the 5th Conference on Computing Frontiers*, CF'08, pages 273–282, New York, NY, 2008. ACM.

[162] A. Rogers and K. Pingali. Process Decomposition through Locality of Reference. *ACM SIGPLAN Notices*, 24(7):69–80, June 1989.

[163] Nezam Rohbani, Tapas Kumar Maiti, Dondee Navarro, Mitiko Miura-Mattausch, Hans Jürgen Mattausch, and Hirotaka Takatsuka. NVDL-Cache: Narrow-Width Value Aware Variable Delay Low-Power Data Cache. In *Proceedings of the IEEE 37th International Conference on Computer Design*, ICCD '19, pages 264–272, Abu Dhabi, United Arab Emirates, 2019. IEEE.

[164] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[165] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[166] Jihyun Ryoo, Mengran Fan, Xulong Tang, Huaipan Jiang, Meena Arunachalam, Sharada Naveen, and Mahmut T. Kandemir. Architecture-Centric Bottleneck Analysis for Deep Neural Network Applications. In *Proceedings of the International Conference on High Performance Computing, Data, and Analytics*, HiPC '19, pages 205–214, 2019.

[167] Pankaj Saha, Satadru Dey, and Munmun Khanra. Modeling and state-of-charge estimation of supercapacitor considering leakage effect. *IEEE Transactions on Industrial Electronics*, 67(1):350–357, 2020.

[168] Christos Sakalis, Zamshed I. Chowdhury, Shayne Wadle, Ismail Akturk, Alberto Ros, Magnus Sjalander, Stefanos Kaxiras, and Ulya R. Karpuzcu. Do Not Predict - Recompute! How Value Recomputation Can Truly Boost the Performance of Invisible Speculation. In *Proceedings of the International Symposium on Secure and Private Execution Environment Design*, SEED 2021, pages 89–100, United States, 2021. Institute of Electrical and Electronics Engineers.

[169] Muhammad Saqib, Tarik Adnan Almohamad, and Raja Majid Mehmood. A low-cost information monitoring system for smart farming applications. *Sensors*, 20(8), 2020. Article no. 2367, 18 pages.

[170] K. T. Schütt, P. Kessel, M. Gastegger, K. A. Nicoli, A. Tkatchenko, and K.-R. Müller. SchNetPack: A deep learning toolbox for atomistic systems. *Journal of Chemical Theory and Computation*, 15(1):448–455, 2019.

[171] Ozan Sener and Silvio Savarese. Active learning for convolutional neural networks: A core-set approach. *arXiv preprint arXiv:1708.00489*, 2017. `https://arxiv.org/abs/1708.00489`, Accessed: 2024-09-26.

[172] Chris Seymour. nanoporetech/bonito: Bonito - A PyTorch Basecaller for Oxford Nanopore Reads. `https://github.com/nanoporetech/bonito`. Accessed: 2024-09-17.

[173] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 165–176, New York, NY, USA, 2004. Association for Computing Machinery.

[174] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. *SIGARCH Computer Architecture News*, 31(2):336–349, May 2003.

[175] Alan F. Smeaton and Michael McHugh. Event detection in an audio-based sensor network. *Multimedia Systems*, 12(3):179–194, December 2006.

[176] Ben Sorscher, Robert Geirhos, Shashank Shekhar, Surya Ganguli, and Ari Morcos. Beyond neural scaling laws: beating power law scaling via data pruning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 19523–19536. Curran Associates, Inc., 2022.

[177] Ben Sorscher, Robert Geirhos, Shashank Shekhar, Surya Ganguli, and Ari Morcos. Beyond neural scaling laws: beating power law scaling via data pruning. In *Advances in Neural Information Processing Systems*, volume 35 of *NIPS '22*, pages 19523–19536, 2022.

[178] Mohamed Sraitih, Younes Jabrane, and Amir Hajjam El Hassani. An automated system for ECG arrhythmia detection using machine learning techniques. *Journal of Clinical Medicine*, 10(22), 2021.

[179] S. Srikantaiah, E. Kultursay, T. Zhang, M. Kandemir, M. J. Irwin, and Y. Xie. MorphCache: A Reconfigurable Adaptive Multi-level Cache Hierarchy. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 231–242, February 2011.

[180] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 207–216, New York, NY, USA, 2015. Association for Computing Machinery.

[181] Michael Stokes, Ryan Baird, Zhaoxiang Jin, David Whalley, and Soner Onder. Improving energy efficiency by memoizing data access information. In *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design*, ISLPED '19, pages 1–6, Lausanne, Switzerland, 2019. IEEE.

[182] Michael Stokes, David Whalley, and Soner Onder. Decreasing the miss rate and eliminating the performance penalty of a data filter cache. *ACM Transactions on Architecture and Code Optimization*, 18(3), May 2021.

[183] Harold S. Stone. A logic-in-memory computer. *IEEE Transactions on Computers*, C-19(1):73–78, 1970.

[184] Mervyn Stone. Cross-Validatory Choice and Assessment of Statistical Predictions. *Royal Statistical Society: Series B*, 36(2):111–133, 1974.

[185] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. TOP500. https://top500.org/lists/top500/, 2024. Accessed: 2024-09-26.

[186] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for modern deep learning research. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 13693–13696, 2020.

[187] Fang Su, Yongpan Liu, Yiqun Wang, and Huazhong Yang. A ferroelectric nonvolatile processor with 46 μ s system-level wake-up time and 14 μ s sleep time for energy harvesting applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(3):596–607, 2017.

[188] Fang Su, Kaisheng Ma, Xueqing Li, Tongda Wu, Yongpan Liu, and Vijaykrishnan Narayanan. Nonvolatile processors: Why is it trending? In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '17, pages 966–971, Leuven, BEL, 2017. European Design and Automation Association.

[189] Luning Sun, Han Gao, Shaowu Pan, and Jian-Xun Wang. Surrogate modeling for fluid flows based on physics-constrained deep learning without simulation data. *Computer Methods in Applied Mechanics and Engineering*, 361, 2020. Article no. 112732, 25 pages.

[190] Haoru Tan, Sitong Wu, Fei Du, Yukang Chen, Zhibin Wang, Fan Wang, and Xiaojuan Qi. Data pruning via moving-one-sample-out. In *Advances in Neural Information Processing Systems*, volume 36 of *NIPS '23*, pages 1–12, 2024.

[191] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning*, ICML '19, pages 6105–6114, Long Beach, California, USA, 2019. PMLR.

[192] Shunpu Tang, Lunyuan Chen, Ke He, Junjuan Xia, Lisheng Fan, and Arumugam Nallanathan. Computational intelligence and deep learning for next-generation edge-enabled industrial IoT. *IEEE Transactions on Network Science and Engineering*, 10(5):2881–2893, 2023.

[193] Xulong Tang, Mahmut Taylan Kandemir, Hui Zhao, Myoungsoo Jung, and Mustafa Karakoy. Computing with Near Data. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(3), December 2018. Article no. 42, 30 pages.

[194] Neil C Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F Manso. The computational limits of deep learning. *arXiv preprint arXiv:2007.05558*, 10, 2020. https://arxiv.org/abs/2007.05558, Accessed: 2024-09-26.

[195] Mariya Toneva, Alessandro Sordoni, Remi Tachet des Combes, Adam Trischler, Yoshua Bengio, and Geoffrey J Gordon. An empirical study of example forgetting during deep neural network learning. *arXiv preprint arXiv:1812.05159*, 2018. https://arxiv.org/abs/1812.05159, Accessed: 2024-09-26.

[196] Robert Vaser, Ivan Sović, Niranjan Nagarajan, and Mile Šikić. Fast and accurate de novo genome assembly from long uncorrected reads. *Genome Research*, 27(5):737–746, 2017.

[197] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 1–11, Long Beach, CA, USA, 2017. Curran Associates, Inc.

[198] André Viebke and Sabri Pllana. The Potential of the Intel (R) Xeon Phi for Supervised Deep Learning. In *Proceedings of the IEEE 17th International Conference on High Performance Computing and Communications, IEEE 7th International Symposium on Cyberspace Safety and Security, and IEEE 12th International Conference on Embedded Software and Systems*, HPCC '15, CSS '15, ICESS '15, pages 758–765, 2015.

[199] Nagampalli VijayKrishna, Jayadev Joshi, Nate Coraor, Jennifer Hillman-Jackson, Dave Bouvier, Marius van den Beek, Ignacio Eguinoa, Frederik Coppens, Sergey Golitsynskiy, Michał Stolarczyk, Nathan C. Sheffield, Simon Gladman, Gianmauro Cuccuru, Björn Grüning, Nicola Soranzo, Helena Rasche, Bradley W. Langhorst, Matthias Bernt, Dan Fornika, David Anderson de Lima Morais, Michel Barrette, Peter van Heusden, Mauro Petrillo, Antonio Puertas-Gallardo, Alex Patak, Hans-Rudolf Hotz, and Daniel Blankenberg. Expanding the Galaxy's reference data. *bioRxiv*, 2020. `https://doi.org/10.1101/2020.10.09.327114`, Accessed: 2024-09-17.

[200] L. Villa, M. Zhang, and K. Asanovic. Dynamic zero compression for cache energy reduction. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-33, pages 214–220, Monterey, CA, USA, 2000. IEEE.

[201] Weixun Wang, Prabhat Mishra, and Ann Gordon-Ross. Dynamic cache reconfiguration for soft real-time systems. *ACM Trans. Embed. Comput. Syst.*, 11(2), July 2012.

[202] Zhibin Wang, Kaiyi Wang, Shouhui Pan, and Hanyun Han. Segmentation of Crop Disease Images with an Improved K-means Clustering Algorithm. *Applied Engineering in Agriculture*, 34:277–289, January 2018.

[203] Sven Warris, N. Roshan N. Timal, Marcel Kempenaar, Arne M. Poortinga, Henri van de Geest, Ana L. Varbanescu, and Jan-Peter Nap. pyPaSWAS: Python-based multi-core CPU and GPU sequence alignment. *PLOS One*, 13(1):Article no. e0190279, 9 pages, 2018.

[204] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[205] W. Wolf and M. Kandemir. Memory system optimization of embedded software. *Proceedings of the IEEE*, 91(1):165–182, 2003.

[206] Anthony D. Wood, John A. Stankovic, Gilles Virone, Leo Selavo, Zhimin He, Qiuhua Cao, Thao Doan, Yafeng Wu, Lei Fang, and Radu Stoleru. Context-aware wireless sensor networks for assisted living and residential monitoring. *IEEE Network*, 22(4):26–33, 2008.

[207] Nitin Yadav and Rajesh Kumar. Harvesting electric energy from waste vibrations of an electric motor using the piezoelectric principle. In Sendhil Kumar Natarajan, Rajiv Prakash, and K. Sankaranarayanasamy, editors, *Recent Advances in Manufacturing, Automation, Design and Energy Technologies*, pages 955–964, Singapore, 2022. Springer Singapore.

[208] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer.

[209] Wucherl Yoo, Kevin Larson, Lee Baugh, Sangkyum Kim, and Roy H Campbell. ADP: Automated diagnosis of performance pathologies using hardware events. *Sigmetrics*, 40(1):283–294, 2012.

[210] Wucherl Yoo, Alex Sim, and Kesheng Wu. Machine Learning Based Job Status Prediction in Scientific Clusters. In *Proceedings of the SAI Computing Conference*, SAI '16, pages 44–53, London, UK, 2016. IEEE.

[211] Wei-Xiang You, Pin Su, and Chenming Hu. A new 8T hybrid nonvolatile SRAM with ferroelectric FET. *IEEE Journal of the Electron Devices Society*, 8:171–175, 2020.

[212] Chun-Chang Yu, Yu Hen Hu, Yi-Chang Lu, and Charlie Chung-Ping Chen. Power reduction of a set-associative instruction cache using a dynamic early tag lookup. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1799–1802, Grenoble, France, 2021. IEEE.

[213] Chuanjun Zhang, Frank Vahid, Jun Yang, and Walid Najjar. A way-halting cache for low-energy high-performance systems. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, ISLPED '04, pages 126–131, New York, NY, USA, 2004. Association for Computing Machinery.

[214] Jie Zhang, David Donofrio, John Shalf, Mahmut T. Kandemir, and Myoungsoo Jung. NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures. In *Proceedings of the International Conference on Parallel Architecture and Compilation*, PACT '15, pages 13–24, Chicago, Illinois, USA, 2015. IEEE Computer Society Press.

[215] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. Benchmarking and Analyzing Deep Neural Network Training. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '18, pages 88–100, 2018.

Vita

Gulsum Gudukbay Akbulut

Gulsum Gudukbay Akbulut was born in Philadelphia, Pennsylvania, USA, in 1996. She holds a bachelor's degree in Computer Engineering from Bilkent University, Turkey. After she found computer architecture topics interesting during her undergraduate education, Gulsum decided to pursue her Ph.D. in computer architecture and joined Penn State University as a Ph.D. student in 2018 under Dr. Mahmut T. Kandemir's supervision as a member of the Microsystems Design Lab. Her dissertation is mainly focused on adaptive optimization techniques for high-performance computing. She has been a technical reviewer for journals such as IEEE Transactions on Computers. She was a teaching assistant for the C++ for Engineers course during her first couple of years. She also worked at Intel Corporation as a Machine Learning Engineering Intern and at AMD as an AI/ML Data Center Research Intern.