Self-aware, Adaptive General-purpose Computing Systems

by

Mihailo Isakov

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved October 2022 by the
Graduate Supervisory Committee:

Michel A. Kinsy, Chair
Aviral Shrivastava
Kevin W. Rudd
Vijay N. Gadepally

ARIZONA STATE UNIVERSITY

December 2022

ABSTRACT

With the breakdown of Dennard scaling, computer architects can no longer rely on integrated circuit energy efficiency to scale with transistor density, and must underclock or power-gate parts of their designs in order to fit within given power budgets. Hardware accelerators may improve energy efficiency of some compute-intensive tasks, but as more tasks are accelerated, the general-purpose portions of workloads account for a larger share of execution time while also leaving less instruction, data, or task-level parallelism to exploit.

Adaptive computing systems have potential to address these challenges by modifying their behavior at runtime. Adaptation requires runtime decision-making, which can be performed both in hardware and software. While software-based decision-making is more flexible and can execute higher complexity operations compared to hardware, it also incurs a significant latency and power overhead. Hardware designs are more limited in the space of decisions they can make, but have direct access to their own internal microarchitectural states and can make faster decisions, allowing for better-informed adaptation and extracting previously unobtainable performance and security benefits.

In this dissertation I study (i) the viability and trade-offs of general-purpose adaptive systems, (ii) the difficulty and complexity of making adaptation decisions, and (iii) how time spent in the observation-analysis-adaptation cycle affects adaptation benefits. I introduce techniques for (a) modeling and understanding high performance computing systems and microarchitecture, (b) enabling hardware learning and decision-making through low-latency networks, and (c) on securing hardware designs using runtime decision-making. I propose an always-awake and active learning 'hardware nervous system' pervasive throughout the chip that can reason about the individual

hardware module performance, energy usage, and security. I present the design and implementation of (1) a reference architecture and (2) a microarchitecture-aware static binary instrumentation tool. Finally, I provide results showing (1) that runtime adaptation is a necessary to continue improving performance on general-purpose tasks, (2) that significant performance loss and performance variation happens under the ISA-level, and is unobservable without hardware support, and (3) that hardware must possess decision-making and 'self-awareness' capabilities at the microarchitecture level in order to efficiently use its own faculties.

ACKNOWLEDGMENTS

Completing this dissertation brings to a close a six year adventure across four different institutions and states. It has truly been a life-changing experience which has allowed me to meet some incredible people, participate in several different scientific disciplines, and develop both as a researcher, a student, a mentor, and a person. None of that would have been remotely possible without the sacrifices of many amazing people to which I will humbly attempt to give due credit.

I will start by acknowledging my parents **Mirjana** and **Svetislav Isakov**, who have given me the love, support, and security to persevere throughout these six years, and without whose sacrifices I would not be here. I would also like to thank my aunt and uncle Dr. **Gordana** and Dr. **Branko Milosavljević**, who were instrumental in helping me achieve my ambitions throughout my life, and without whom this PhD would not have been possible.

My PhD would not have been remotely as fruitful if I was not mentored by Professor **Michel A. Kinsy**. Professor Kinsy has truly put his heart and soul into training me to become a self-sufficient researcher as well as a person, and has made great sacrifices over the years so that I might grow. He has enabled countless opportunities for me, has introduced me to a variety of scientific fields (much more than I signed up for), has taught me everything from hardware design, to research taste, to scientific methodology, to how to build and run a laboratory, and has strived to expose me to the decision-making I will have to perform as an independent researcher. It has been a wonderful ride, and I would have changed none of it.

Besides Professor Kinsy, I was lucky to collaborate with Dr. **Vijay Gadepally**, who put faith in me while I was just entering the field of hardware security and cryptography. Dr. Gadepally has always been in my corner, even as we moved across

Boston University, Texas A&M and finally Arizona State University. I am grateful for all the long hours spent in conversation with Dr. **Kevin Rudd**, the tremendous effort he has provided in guiding me to complete this dissertation, as well as for helping me reason about a whether there is life after PhD. I would like to thank **Andrew (Andy) Glew** for all the branching and diverging discussions we had, as well as his incredible excitement and deep technical insight he is always willing to share. Dr. **Bryant York** has played a large role in my education, and I thank him for steering me into more theoretical areas of computer science and engineering, but even more for showing me an alternative path through academia and how one can preserve the joy of doing research. I would also like to thank Dr. **Aviral Shrivastava**, who I met late into my degree and who has graciously accepted to help me cross the finishing line, as well as Dr. **Seth Abraham** for the time he dedicated to help clarify my dissertation.

During my graduate education I was lucky to collaborate with some brilliant researchers across several institutions and continents. The work I have done in collaboration the folks at Argonne National Laboratory forms the essence of this dissertation. I would like to thank Dr. **Prasanna Balaprakash**, Dr. **Sandeep Madireddy**, Dr. **Philip Carns**, and Dr. **Robert B. Ross** for all their help and time over the several years we have worked together. In my second collaboration with Argonne, I was lucky to be invited by Dr. **Valerie Taylor** to spend time in Chicago and be exposed to some of the most challenging applications of computer engineering I have experienced. I am also grateful to Dr. **Xingfu Wu** for his mentorship during that research visit, and for all time spent with Dr. **Wilkie Olin-Ammentorp**. Finally, I am grateful for the collaboration I had with Dr. **Ahmad-Reza Sadeghi**, Dr. **Ghada Dessouky** and Dr. **Emmanuel Stapf** at TU Darmstadt.

I would like to thank Dr. **Alan Ehret** with whom I shared the highs and lows of

starting, persevering through, and finishing a PhD across Boston, Texas and finally Arizona. Our lab has grown and shrunk several times during the years, and I miss many of my fellow lab mates: **Novak Boškov**, who has been a great friend as well as a collaborator, challenging me to think more broadly about computer science, **Miguel Mark**, who has been a joy to work with and share a lab as well, **Lake Bu**, for all his kindness and mentoring during my early graduate career, **Mikaela Currier**, for all her friendship, as well as enthusiasm and energy she put into our collaboration, **Colin Jaye**, **Adam Awale**, **Daniel Pietz** and **Jacob Abraham** for their friendship during my last year at ASU, and everyone else that helped make our lab(s) a great place to be in: **Sahan Bandara**, **Rashmi Agrawal**, **Eliakin del Rosario**, **Shreeya Khadka**, **Nikola Hardi**, **Donato Kava**, **Micah Secrest**, **Kuntal Nayak**, **Sarthak Pradhan**, **Alexandras Birutis**, **Trey Manuszak**, **Raj Kane**, **Sandesh Ghanta**, and many others.

During my time at Boston University I was lucky to know Dr. **Annie Rabi Bernard**, who has shown me incredible kindness, has been there for me through some of the toughest times during my PhD, and who has greatly helped me develop as a person. I am grateful for my BU friends **Assel Aliyeva**, **Rasoul Jahanshahi**, **Nataša Trkulja**, **Nithin Sivadas**, **Kidane Kebede**, **Onur Zungur**, **Prashant Vaidyanathan**, **Stefan Gvozdenović**, **William Blair** and many other good folks. I would also like thank many of my US-based friends outside of academia: **Ida Boškov**, **Sean Johnston**, **Alex Walker**, **Madison Van Doren**, **Evan DeaKyne**, **Paulina Milligan**, as well as my Serbian friends to which I would do an injustice to by enumerating them.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Chapter 1

INTRODUCTION

1.1 General-Purpose Computing Systems

General-purpose computing systems are a type of computer architecture tasked with executing all programs that do not have more efficient domain-specific or application-specific accelerators. This class of systems includes microcontrollers, microprocessors, mainframes, superscalar processors, out-of-order processors, Very Large Instruction Word (VLIW) processors, multicore and manycore systems, etc. All of these systems are Turing-complete, i.e., can execute any algorithm given enough memory and storage. Because of their generality, adoption, and ease of use, they are typically the first (and often the last) computing platform programmers and users target. As such, improving the performance, energy efficiency or security of general-purpose computing systems can have an impact on a very large number of users.

1.1.1 General-Purpose vs. Application-Specific Computing

General-purpose computing platforms are rarely the optimal solution for any given problem. The cost of their generality is that they do not leverage potential domain-specific properties of a given workload, which would help to better satisfy user goals. By building hardware features optimized to help a broad ranges of workloads, many features provisioned in general-purpose systems go unused on different workloads. For example, systems may possess large caches, translation lookahead buffers, branch

predictors, and hardware prefetchers, and all of these features may help some subset of all workloads, but rarely do all of these features contribute at the same time. In other words, if a user has a specific application they are targeting, a 'leaner' system may offer e.g., better energy efficiency while maintaining the same performance.

1.1.2 The Need for Accelerating General-Purpose Computing

With the increasing difficulties in miniaturizing semiconductors, along with stalled core frequencies and slowing improvements in single-threaded performance, computer architects are seeking alternative ways to accelerate general-purpose programs. Accelerators may help some subset of these programs, but the majority of codes cannot or will not be ported to accelerators, either due to a poor mapping to the accelerator compute substrate, or due to lack of investment. Similarly, multicore and manycore systems offer increased task-level parallelism, but many general-purpose tasks are difficult to parallelize and cannot make good use of such systems. One solution to improving performance and energy efficiency of general-purpose workloads is to build general-purpose architectures that 'adapt' to workloads, e.g., by powering off underused hardware features or reconfiguring their microarchitecture to better fit workload behavior. Instead of 'optimizing for the average case', adaptive general-purpose computing systems can potentially mimic as architectures designed for the best case.

1.2 The Obstacles in Improving General-Purpose Computing

The pace of performance improvement in general purpose computers has significantly decreased since around 2005, as Figure 1.1. The source of this impact is multi-causal:

1. The breakdown of Dennard scaling means that growing portions of chips must be turned off in order to fit within a power budget, forcing majority of processors to be clocked in the low Gigahertz range.
2. Many workloads cannot benefit from wider pipelines due to low instruction-level parallelism (ILP), preventing designers from further scaling out-of-order superscalar cores.
3. Low ILP has forced architectures to use deeper pipelines with aggressive speculation in order to improve performance, but existing designs are pushing the limits of what can be predicted, and may be limited by natural randomness or lack of knowledge about incoming workloads.

In this section I will cover these three effects in detail.

1.2.1 End of Dennard's Scaling and the Dark Silicon Problem

Exponentially increasing transistor counts as predicted by Moore's law and the accompanying reduction in power per area known as Dennard scaling (Dennard et al. 1974) have provided half a century of exponentially growing single-threaded performance. However, since around 2006 Dennard scaling has broken down (witnessed by the capped frequency of high-end CPUs) and the slope of transistors per area predicted by Moore's law is tapering off. After the breakdown, integrated circuits

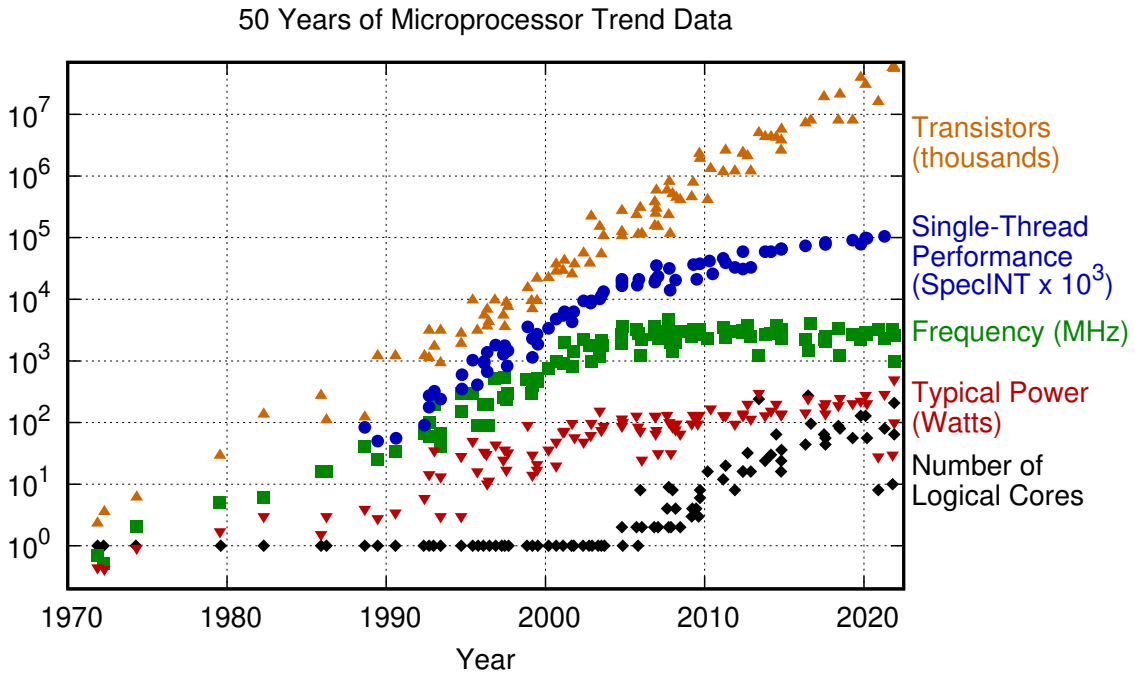


Figure 1.1: A collection of different general-purpose processor transistor counts, frequencies, average power usage, SPECINT benchmark performance results, and number of cores, with respect to year of release.

Source: Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. New plot and data collected for 2010-2021 by K. Rupp.

(ICs) built in every new process technology require more power per area to function, but due to inadequate cooling, a growing percentage of the IC must be turned off to fit within the power budget.

1.2.2 The Impact of Modern Computing Workloads

Compute workloads have significantly changed in the past two decades with the introduction of smart phones and increased reliance on cloud computing. Significant amounts of compute power have moved from desktop computers to the edge and the

cloud that supports it. Processors are typically designed and tuned to achieve good performance on representative workloads such as the SPEC and PARSEC benchmarks, but these workloads do not necessarily follow real-world computing trends.

Recent benchmarks such as CloudSuite (Ferdman et al. 2012) show that modern applications such as web servers, media streaming, data analytics and web browsing exhibit significantly different behaviors than that of conventional, compute-heavy benchmarks found in SPEC and PARSEC. These novel applications often have low instruction-level parallelism, high instruction cache and TLB miss rates, are unresponsive to increased L3 cache sizes, and generally have low utilization of modern superscalar and out-of-order processor features.

They are also difficult to accelerate due to both their generality, and due to the nature of their performance bottlenecks. Workloads with low ILP can be accelerated through more 'aggressive' speculation, but if control flow is unpredictable, significant amount of energy will be wasted speculating on branches that are not taken. Similarly, instruction cache and TLB miss rates may be decreased through larger caches and more aggressive prefetching, but such approaches tend to increase power usage linearly, and increase performance asymptotically with scaling. Since processors are increasingly energy bound, scaling microarchitecture to larger, wider pipelines and caches must be accompanied with increased energy efficiency.

1.2.3 Limited Architectural Design-Time Opportunities

Modern processors are achieving diminishing returns from aggressive processor features such as wide pipelines, branch prediction, speculative prefetchers, large caches, data value prediction, and macro-op fusion. Many existing workloads rely on these

hardware features and disabling them would significantly decrease their Quality of Service (QoS). However, workloads that rely on *all* of these features are rare, i.e., only a portion of these hardware features contributes to their performance. Without a method to specify which features are useful, all of these hardware features use power. For example, graph processing workloads may require a large amount of pointer chasing and can be extremely unpredictable in terms of control flow and data movement. While branch prediction and smarter data prefetching can accelerate these workloads, wider pipelines and large caches may be underutilized. Since caches often consume the majority of power of a CPU, graph processing workloads may be wasting more than half of the energy spent, and have great potential to improve their energy efficiency. This inefficiency is partly the result of computer architects having limited visibility into the future workloads. Even if designers had the capability to accurately predict the landscape of workloads a general-purpose processor will execute, the sheer diversity of tasks may make any application-specific circuitry impractical.

1.3 Venues for Continued Performance Improvements

Several venues have been proposed in the past decades to supplement the decreased growth in single-threaded performance, from exploiting application parallelism to using schedule-time or run-time knowledge about an application in order to adapt the system running the application.

1.3.1 Exploiting Weak Parallelism

As Figure 1.1 shows, modern processors have exponentially rising numbers of cores. Large manycore systems sacrifice single-threaded performance by limiting core frequency and by being conservative with hardware features in order to save on die area. In return, these cores on aggregate gain a larger amount of compute power, assuming the existence of workloads that can fully utilize them. Manycores are well suited to tasks that exhibit weak scaling — a property of workloads where the workload may not be directly parallelizable, but growing the workload increases the workload's parallel portion. Examples of this are databases and web servers, which cannot utilize more cores to faster respond to individual queries or requests, but support requests in parallel given more independent cores, and therefore have better throughput.

1.3.2 Application Specific Architectures

Certain compute-heavy tasks such as graphics rendering, cryptography, machine learning, media compression, data analytics, and cryptocurrency mining can be made faster and more efficient through Application Specific Integrated Circuits (ASICs). These application 'accelerators' have become common in the post-Dennard world, with Graphics Processing Units (GPUs) and neural network accelerators becoming ubiquitous in high-end System-on-Chip (SoC) devices. These accelerators trade generality for efficiency, and are ill-fit for accelerating general-purpose tasks that this work targets.

1.3.3 Profile-Guided Optimization, Dynamic Binary Optimization and Just-In-Time Compilation

Profile-Guided Optimization (PGO) is a method for improving the compilation process by providing the compiler with representative program inputs so that it may understand the statistical distribution of memory accesses, loop iterations, control flow decisions, cache misses, etc. Similarly, Dynamic Binary Optimization (DBO) attempts to augment a program at schedule-time or run-time with newly available knowledge about the workload or system, e.g., by vectorizing loops with vector instructions available on that specific processors, or inlining certain loops once program inputs are known. Interpreted languages such as Python and JavaScript, as well as bytecode languages such as Java and WebAssembly have great portability but require an OS-specific interpreter or a runtime in order to be executed. There exists a significant performance penalty for interpreting these languages, and all of the top-performing interpreters and runtimes currently use Just-In-Time (JIT) compilation. JIT compilers compile the inputted code in small increments while simultaneously executing the generated binaries, typically at the basic block level. The next time an already executed section of code is ran, it leverages the cached compiled code, achieving a significant performance improvement. JITs may perform additional optimization passes once code hot-spots are detected, further improving performance. JITs also allow leverage runtime information that is typically not available to the compiler, potentially achieving better results than those of compiled languages, though these benefits are rarely seen in modern processors that perform similar steps in hardware.

Together, these three types of methods can be viewed as applicable at compile-

time (PGO), scheduling-time (DBO), and run-time (JIT), as they leverage whatever information about the workload and system is available to them at the time.

1.3.4 Hardware (Re)Configuration and Provisioning

Reconfigurable architectures such as Field Programmable Gate Arrays (FPGAs) and Coarse Grained Reconfigurable Architectures (CGRAs) have been proposed as a method to accelerate workloads by reconfiguring themselves into application specific circuitry *at run-time*. While extremely flexible, when emulating general purpose processors, these architectures achieve significantly lower clock speeds and energy efficiency compared to ASIC implementations. In order to make up for this, reconfigurable architectures must exploit some underutilized property of the workloads to gain a competitive edge over ASICs, but such exploits are by nature application-specific, hence reconfigurable architectures have not yet found broad use in general-purpose computing.

1.3.5 System Run-time Adaptation and Introspective Hardware

Adaptive systems change their behavior at run-time to better fit some task or system state. To adapt, these systems need three components: sensors, actuators, and decision-making capability. Sensors supply the decision-making system with information. The decision-making system decides which actuators should be activated and how. Actuators act on decisions and change system behavior to fit tasks and goals. The goal of adaptation may be to improve performance, latency, energy efficiency, security, or some other, possibly user-specified goal.

Adaptation is similar to dynamic binary optimization and just-in-time compilation in that it utilizes runtime information unavailable to programmers and the compiler at compile time. However, hardware adaptation potentially has two advantages above other approaches. First, hardware adaptation has a greater amount of insight into the program due to its ability to access both ISA-level and microarchitectural information. Second, the space of actions JIT or DBO can take is more limited than what hardware can do, as hardware may adapt its microarchitecture to power down unused hardware features or provision more shared hardware to processes that may make use of them. These approaches are complementary though, since the capability of hardware to perform high-level analysis of running binaries is limited.

1.3.6 Directions for Improving General-Purpose System Performance

The above approaches I have listed above can be separated into two categories: (1) application-oriented (ASICs / accelerators, weak scaling, JITs, etc.), and (2) system-oriented (run-time adaptive systems, hardware reconfiguration, provisioning). In this dissertation I will focus on the second approach, since system-oriented methods for increasing computing capabilities are more applicable to general-purpose workloads.

1.4 Adaptive, Introspective and Intelligent Systems

In general, computing systems may adapt their behavior with respect to both running workloads, and internal system states. Examples of the first class are e.g., JITs, which change how a workload is executed once a workload is known. Examples

of the second class are e.g., CPU controls that change core frequency depending on current compute load.

I will use the term 'adaptive' to refer to systems that may change their behavior over time and have *actionability* over how workloads are executed. I will use the term 'introspective' to refer to systems that have *visibility* over both workloads and their internal microarchitecture in order to inform any adaptation decision-making system about beneficial adaptations. I will use the term 'intelligent' to refer to systems that have *agency* over runtime adaptation decisions and can learn and change decisions given more information about workloads or system states. While these three terms are connected, fundamentally adaptation allows systems to take actions, introspection allows systems to observe themselves, and intelligence allows systems to make new decisions which have not been defined at design time.

1.4.1 Adaptive Systems

Static systems are systems that given some workload and initial system state, always execute the workload using the same subset of the microarchitecture and exhibit the same behaviors in treating the workload. While small divergences due to timing, shared resource contention, or initial state may affect the specific states system microarchitecture will go through to execute the workload, the static system has no agency and does not significantly change its behavior over repeated runs of the same workload.

Unlike static systems, adaptive systems *are* able to change their behavior at runtime. The adaptive system has a number of actions it can take, with each action defined through some system actuator or 'knob'. Examples of such actuators are

(i) reconfigurable caches which can exchange number of ways for larger number of cache sets, (ii) tunable CPU pipeline width, allowing processors to load and execute less instructions in parallel while saving power, (iii) Dynamic Voltage Frequency Scaling (DVFS), allowing systems to trade-off frequency for energy-efficiency, etc. In this dissertation I assume the view that many existing adaptation actuators in modern systems should be considered adaptive. However, these mechanisms are typically not controlled by learning entities, i.e., entities that observe workload and system state, decide what changes would benefit the system most, and store memory between observations in order to improve future decisions. Instead, actuator behavior is often hard-coded and does not adapt to changing workloads or unexpected system states.

1.4.2 Introspective Systems

Introspective of 'self-aware' systems adapt not only to the tasks being executed, but also have insight into their own internal (microarchitectural) system state. This state may be accessible to the operating system and potentially user space applications (e.g., CPU utilization or latency of network traffic), or it may be private (e.g., internal microarchitecture of a chip).

In the processor design domain, logic is typically split between three layers or 'fabrics': the compute, memory, and networking fabrics. Introspective adaptive systems may gain insight into all three of these fabrics in order to have greater visibility and control into the system and make better decisions. Therefore, these systems have a fourth, 'introspective' fabric which taps and controls the previous three layers, as Figure 1.2 shows.

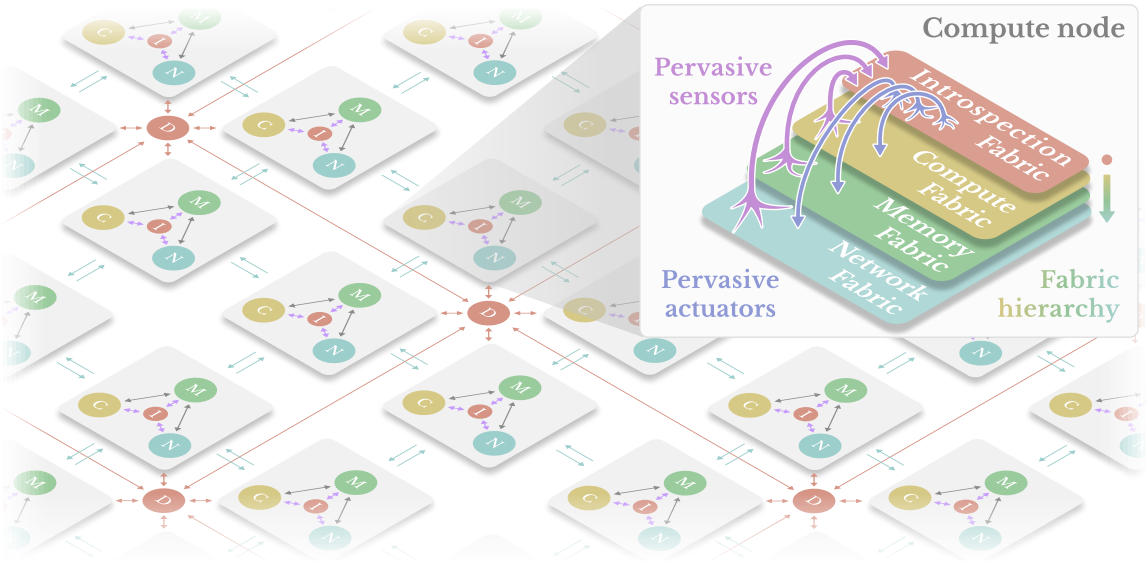


Figure 1.2: The three typical computing stack layers (network, memory, and compute), with a fourth, pervasive layer: the introspection fabric.

1.4.3 Intelligent Systems

In this work I consider computing systems 'intelligent' if it possesses the ability to learn: if the system is repeatedly reset to a known fixed initial state and is tasked with executing the same workload, on subsequent runs the system should change its behavior in order to improve goal satisfaction. Conversely, a system is 'unintelligent' if it is unable to leverage information about previous workloads to improve future execution. Note that this known initial state into which a system is reset between workloads applies to system microarchitecture but not to some intelligence subsystem, since the information about learned lessons must be stored somewhere. For example, an intelligent processing core might reset its internal state by flushing a whole pipeline, but a flush should not reset any learned branch prediction patterns.

The intelligence of systems is also a property of the time scale they are observed within. For example, a given system may be unable to learn on the microarchitectural

level, and repeating the same program phase may not improve system execution. However, the system may learn at scheduling time, e.g., by detecting and isolating misbehaving workloads, so that subsequent workload runs may improve during execution. Taking a broader view, while a specific core or node may not exhibit intelligent behavior, a datacenter or HPC system may over larger time scales. For example, as workloads in a datacenter change, system operators may observe workloads and install updates or optimize software to better control the system, or may provision hardware components more appropriate to the workload distribution. With human operators viewed as a part of the system, the system may exhibit behavior that adapts to changing workloads.

1.4.4 Design-time, Scheduling-time and Run-time Adaptation

In this work we will observe adaptation at three different time scales: (i) Design-time adaptation: adaptation to workloads or system states during the design process of the system, based on data collected in simulation or in the field. The design process may be repeated and upgraded versions of the system may replace existing systems. (ii) Scheduling-time adaptation: adaptation to workloads in-between repeated executions of the same or similar workloads, based on data collected after workload execution has completed. Adaptation may statically configure software or hardware components to best fit future runs. (iii) Run-time adaptation: adaptation to workloads during workload execution, based on both information available at scheduling time, and on runtime information available through ISA-level instrumentation and microarchitectural probing.

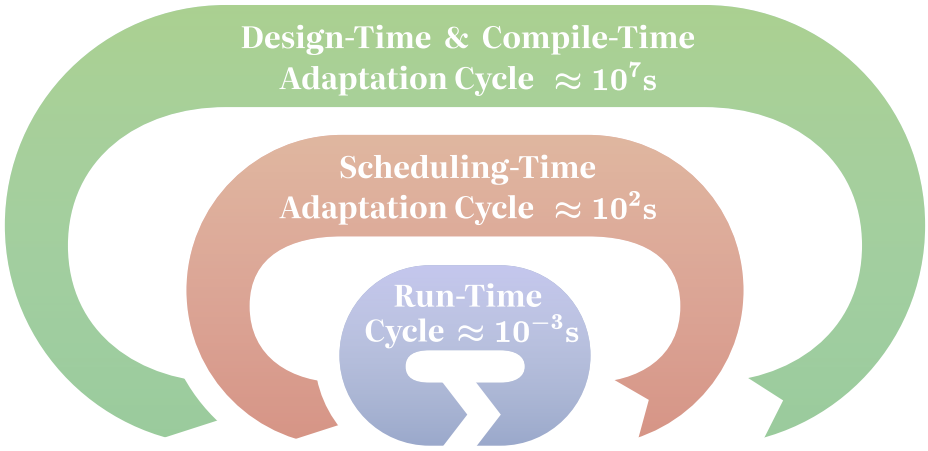


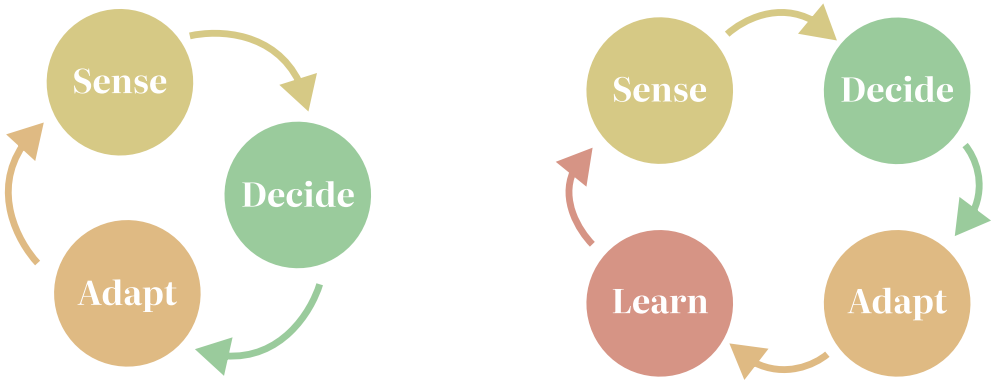
Figure 1.3: Three phases at which adaptation can be performed: at design-time, scheduling-time, or runtime.

1.4.5 Adaptive System Phases

As mentioned in subsection 1.3.5, adaptive systems have three components: sensors, actuators, and decision-making mechanisms. Without loss of generality, an adaptive system can be viewed as operating in a loop shown in Figure 1.4. In practice, a system will likely both perform measurements, make decisions, and act on those decisions in parallel, but this framework will be help analyze the theoretical limitations of adaptive systems in Chapter 7.

1.4.6 Benefits of Adaptive Systems

Adaptive systems can be contrasted with other general-purpose and application-specific (ASIC) architectures on metrics such as performance, energy efficiency, and programmability. Figure 1.5 offers a qualitative diagram of how adaptive systems may compare to other classes of architecture. In terms of performance, while application-



(a) The three adaptation stages.

(b) The four adaptation stages.

Figure 1.4: Three and four stage adaptation cycles.

specific architectures such as GPUs and ASICs lead in performance, adaptive architectures offer equal or superior performance to that of superscalars and many-core processors, since adaptive architectures are a superset of general-purpose processors. In terms of energy efficiency, adaptive architectures can use their run-time insight to lower power usage compared to other processors. In terms of programmability, similarly to modern out-of-order superscalar processors, adaptive architectures may offer good performance on untuned code, simplifying development.

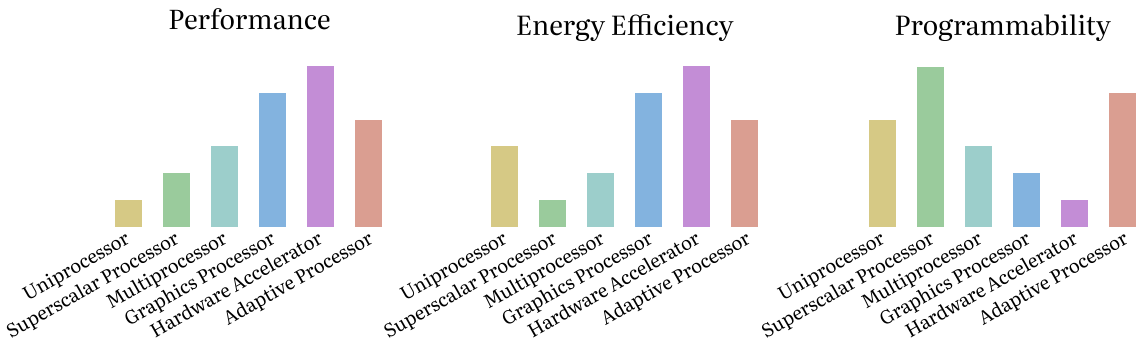


Figure 1.5: A qualitative diagram of the different features of common architectures.

1.5 Aims, Objectives and Research Questions

In this section I discuss the hypothesis of this dissertation, the research questions it will tackle, and my objectives and aims for the dissertation.

1.5.1 Dissertation Hypothesis

The hypothesis of this dissertation is that hardware is better suited than software to make adaptation decisions due hardware's fast response times and exclusive access to internal system state, and can therefore exploit previously unobtainable adaptation benefits, for longer periods of time.

In other words, extracting the full benefits out of general-purpose computing system adaptation may fundamentally require that decision-making is performed at the same time and space granularity as useful computation. If so, adaptation decision-making cannot be performed on the same substrate as general-purpose computation, and requires specialized architectures and subsystems that can keep pace with the general-purpose system.

1.5.2 Research Questions

Currently, adaptive systems as defined in this work have seen relatively little commercial success, and there exist many open questions around their theoretical limits, design methodology, and practical implementations.

The questions I aim to answer in this dissertation are: assuming oracular decision-making, what are the limits of system adaptation with respect to adaptation frequency?

How do adaptation benefits change when adaptation is performed at design-time, workload scheduling time, and run-time? To make these decisions, decision-making algorithms must possess a model of the system being controlled. Here I ask: can systems models describe real system implementations without fully emulating said systems, and explore their structure, granularity, and complexity? Can systems models generalize to new workloads, and can they be static or must they change over time? Can such models be analyzed to better understand the system they are trained on, and can confidence about their predictions be quantified? Finally, I ask: what level of system and workload insight is necessary to approach oracular decision-making? Is architectural (ISA-level) information sufficient to make these decisions, or must systems be re-architected to expose the system internals? Can these internals be automatically selected, or must system designers participate in this process?

1.6 Dissertation Contributions

1.6.1 Contributions to System and Workload Modeling

I show that workload behavior clustering is a useful method for labeling and analyzing workloads, and provide case studies on using hierarchical clustering to optimize HPC jobs. I show that accurate workload and system models can be built from post-execution logs, and that these models can be interrogated in order to understand why workloads or systems behave as they do. I apply HPC clustering insights to microarchitectural adaptation by clustering and analyzing program execution phases.

1.6.2 Contributions to Modeling Error Quantification Methodologies

I show that there exist significant roadblocks to the deployment of systems models. I present a methodology for quantifying why ML models fail when applied to real-world systems, give a taxonomy of modeling errors, provide litmus tests that quantify different error components, and provide methods for reducing each noise source. I apply these insights to design better schedulers, which can analyze why jobs underperform by querying the error prediction model and determining whether jobs are sensitive to contention, are novel workloads, etc.

1.6.3 Contributions to Software Instrumentation

I present the Trireme RISC-V Analysis and Instrumentation Library (TRAIL), a RISC-V-based static binary analysis library that rewrites binaries prior to execution, adding logging functionality and modifying binary behavior. I show how this library can be used to gather ISA-level information about workloads. Next, I propose utilizing the open-source nature of the RISC-V instruction set architecture to (1) enable high-speed extraction of microarchitectural data which can help train better adaptation mechanisms, and (2) present a design that minimizes the amount of microarchitectural pollution that stems from binary instrumentation and microarchitectural probing.

1.7 Outline of the Thesis

In Chapter 2, I provide the motivation for why adaptive and introspective systems are the necessary next step in general-purpose computing. In Chapter 3, I present

a formalization of static systems, statically-adaptive systems, dynamically-adaptive systems, and online-learning system. I introduce 'systems models' and conditions under which systems can be modeled using machine learning algorithms. In Chapter 4, I introduce an experimental setup for modeling systems and workloads based on historical data collected from leadership-class supercomputers.

In Chapters 5, 6, and 7 I explore design-time, scheduling-time, and run-time system adaptation. In Chapter 5, I focus on system design exploration across a multiple generations of a system, project or product, and present `hoppi`: a tool for finding Pareto-optimal system configurations based on both analytical systems models and simulation data. In Chapter 6, I focus on modeling workloads and develop a set of machine learning methods that accurately predict how workloads behave on a specific system. I introduce Gauge, a tool for visualization and exploration of high-performance computing (HPC) workloads, and present how Gauge can be used to interpret black-box models of large systems, helping users discover system bottlenecks and underperforming workloads. In Chapter 7, I focus on applying models from Chapter 6, and present results from applying ML models of workloads and system in production. I showcase the difficulties in deploying systems models, and introduce a taxonomy of systems modeling errors. Finally in 8, I present results and conclude this dissertation.

Chapter 2

MOTIVATION

2.1 Existing Venues for Increasing General-Purpose Performance

Methods for increasing system performance on a set of workloads typically fall into one of the following categories: (i) increasing processor frequency, (ii) increasing the number of instructions per cycle a core can execute, e.g., in the case of superscalar and out-of-order processors (instruction-level parallelism or ILP), (iii) operating on larger amounts of data in parallel through e.g., in the case of vector units and graphics processors and (data-level parallelism or DLP), (iv) increasing the width of or combining independent instructions, e.g., in the case of Very Large Instruction Word (VLIW) computers (bit-level parallelism or BLP) (v) increasing the number of parallel tasks a system can execute, e.g., in the case of multicore and manycore processors (task-level parallelism or TLP)

Many types of general-purpose workloads exhibit low parallelism and can benefit from higher single thread performance, but not from higher processor core counts. For decades, computer architecture has seen exponential growth in single-threaded performance (see blue line in Figure 1.1), owing to several venues of research and development.

Improvements in semiconductor manufacturing have allowed processors to run at higher frequencies, directly improving single-threaded performance (see green line in Figure 1.1). Semiconductor miniaturization increased the transistor count budget available to hardware designs, providing a larger budget for wider pipelines,

larger caches, and more compute units. However, frequencies have stalled due to the breakdown of Dennard scaling and the inability of cooling systems to follow CPU total power draw (see red line in Figure 1.1).

Single-threaded performance has benefited not just from manufacturing, but also from architectural improvements. By exploiting instruction-level parallelism, superscalar processors attempt to executed multiple independent instructions at the same time. To do so, superscalar processors spend their transistor budget to widen instruction and data cache bandwidths, and increase the number of execution lanes. Since independent instructions may be several instructions apart, out-of-order processors spend significant amounts of logic to track instruction dependencies, and execute them out of order without sacrificing correctness. Increasing pipeline size and having more instructions in flight can directly improve IPC on a subset of workloads. However, as pipelines get larger and carry more instructions, control flow penalties grow proportionally. Since branch instructions are resolved relatively late in a pipeline, processors typically predict whether a branch is taken or not taken in order to schedule instructions following a branch. A branch predictor is a processor subsystem that learns branch behavior and attempts to predict future branch decisions and targets. When successful, a processor can fill its pipeline and achieve high hardware utilization. When unsuccessful, the processor must discard instructions behind a mispredicted branch. Better branch predictors decrease the percentage of in-flight instructions that must be discarded, directly improving single-threaded performance and allowing designers to build deeper and wider processor pipelines. Similarly, processors may be bottlenecked by the 'memory wall', where data-intensive programs spend large amounts of time waiting for memory to arrive from caches or DRAM. By building hardware prefetchers that predict which addresses will be accessed in the future, designers can again improve

single-threaded performance. While both of these approaches are provide a significant performance benefit in modern processors, the leftover mispredicted branches and cache misses are more random and harder to predict, causing further investment in terms of hardware area of power yields diminishing returns.

Difficulties in further improving single-threaded performance have incentivized the development of multicore and manycore systems (see black line in Figure 1.1). These systems achieve close-to-linear scaling in terms of performance at the cost of increased programmability. Since many general-purpose tasks are limited by Amdahl's law, and most workloads require additional effort to parallelize, task-level parallelism is less generally applicable compared to other approaches.

Vector and VLIW processing suffers from similar problems: vectorizing applications can provide significant improvements on a subset of workloads, but is application-specific and requires programmer effort or auto-vectorizing compilers. While some benefits can be immediately gained, increasing vector processor widths or very large instruction word widths offers diminishing returns.

2.2 Necessity of System Adaptation

With semiconductor manufacturing process technology is approaching the fundamental limits of physics and microarchitectural approaches are extracting most of the benefits of instruction-level parallelism, the number of venues that can offer performance improvements for general-purpose tasks is decreasing.

To illustrate this problem, let's assume that some future non-adaptive processor design with (1) a fixed power budget and (2) an unlimited hardware area budget is mathematically proven to provide optimal performance on a wide set of workloads.

Since the design is optimal, there exists no improvement in e.g., ALU, pipeline, cache, or branch predictor architecture that can on average improve performance on a given set of workloads. While system optimality cannot be disproven, removing some of the design constraints can invalidate its conclusions. One of those constraints is the immutable nature of microarchitecture, which specifies that the same microarchitecture is used for all workloads. If the hardware design can change its design and behavior for each specific workload ahead of workload execution (at scheduling time), it should be able to find equal or better-performing configuration for every workload. If the hardware can change its design and behavior *during* execution, it may be able to achieve equal or better performance on every program phase as well. Note that this argument applies not only to performance, but to other metrics such as security as well.

The crux of the argument is that optimizing a single system design for a broad range of workloads leads to suboptimal performance on all of them. In Figure 2.1 I illustrate an example where a system with two parameters is iteratively optimized to satisfy workloads from three different applications. A workload is represented by the configuration of the two parameters which would provide it with the best performance, energy-efficiency or any other metric being optimized. A single optimal but static system arrives at equilibrium at the end of the optimization procedure once the weighted sum of all gradients imposed on the parameters is cancelled out. The design is Pareto-optimal, in that improving goals of any single application leads to sacrificing goals of another application.

The promise of system adaptation is that systems do not have to exist as a single configuration or 'point' in the parameter space, but can instead be defined as a *space* of possible configurations. A system may be able to choose at scheduling time or

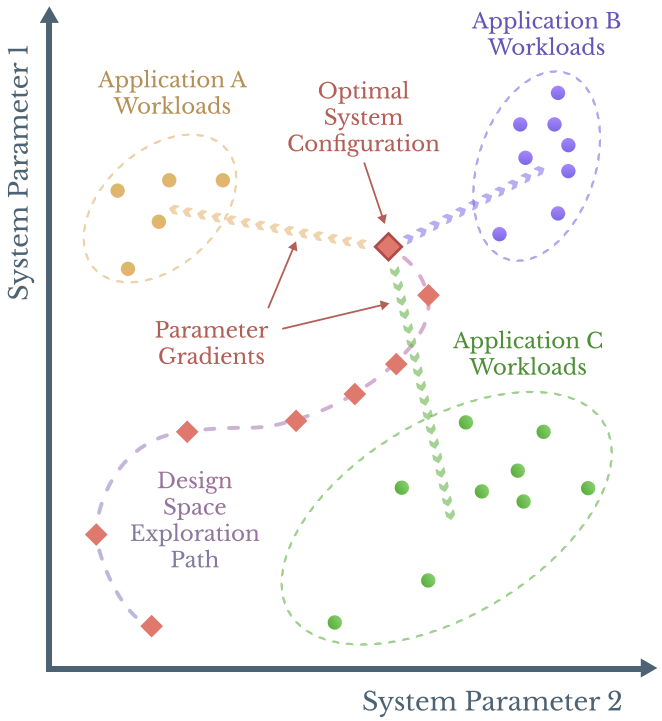


Figure 2.1: An illustration of a design space with two parameters shown on X and Y axes. Best-performing design configurations targeting individual workloads from applications A, B and C are shown in yellow, purple and green. A general-purpose system (red) is iteratively optimized (violet line) until no improvement can be found.

runtime which point in the space to adapt into, approaching optimal configuration for given workload once that workload is known.

2.3 Motivating Examples

To illustrate why general-purpose computing architectures need to be able to adapt at run-time, I will present three different scenarios:

- A situation where more insight into the system can improve quality of service,
- A situation where run-time decision-making and online learning is needed to stop security attacks,

- A situation where simple actuators can have a great impact on energy efficiency of a system.

2.3.1 The Need for Better System Behavior Monitors: Improving QoS for Contention Sensitive Jobs

Certain workloads or behaviors may be very sensitive to memory, network or I/O contention. Take network latency sensitive jobs as an example: network latency sensitive jobs have network communication on their critical path, i.e., their runtime directly corresponds to the time between network request and response start times. A typical behavior of both computer networks (Ethernet / Infiniband / etc.) and Network on Chip (NoC) systems is that as network bandwidth utilization approaches 100%, packet latency approaches infinity. This effect is caused by the fact that when network sources produce more traffic than network sinks can process, network queues grow and packets take longer and longer to get processed. These two effects cause network latency-bound jobs to be particularly sensitive to network utilization, and may have significantly varying performance on similar systems that only differ in the amount of network traffic. The effects may be difficult to diagnose, and conventional general-purpose systems do not provide any support for diagnosing when a job is network latency sensitive, nor do they typically support any provisions for such jobs.

That is not to say that there are not simple fixes to the problem - take the following adaptive system sketch as an example. By collecting execution logs containing job runtime and average network utilization during job execution, the system may infer that for a certain subset of jobs, there exists a positive correlation between the two. This sensing is simple to implement: process runtime is already collected on modern

operating systems, and average network utilization can be collected through software or hardware performance counters in the Network Interface Card (NIC) or the virtual channel buffer in NoCs. Once certain jobs are classified as network latency-sensitive, this knowledge can be used to adapt the system during their execution. Since by definition, network latency-sensitive jobs execute faster when network latency is low, while latency-insensitive jobs do not, network latency-sensitive jobs will benefit from having priority in the network buffer, while other jobs will not sense a significant performance penalty. Implementing packet prioritization may possibly be performed in software, but likely requires hardware support on most high-performance systems. This software or hardware support would be an 'adaptation knob': a run-time controllable parameter that can provide better goal satisfaction given that the controller *has access to the right information about the workload and the system*, and makes the right decisions. This example illustrates that the adaptive system requires that hardware sensors and architectural monitors are placed at the right locations in the system in order to inform the decision-making system and extract full adaptation benefits.

2.3.2 The Need for Better Hardware Decision Making: Real-Time Microarchitectural Attack Detection

When controlling a system that exists within with a possibly random / chaotic but stationary (time-insensitive) environment, there are typically fundamental limits in how complex the controller must become to achieve an ideal outcome. For example, many classical control theory problems have analytically-derived optimal solutions that are simple to describe and to compute, and work even in the presence of large amounts of noise. When dealing with a thinking adversary on the other hand, the controller

complexity can be unbounded. This is apparent from chess engines to antivirus detectors: when the state space is large, it is difficult to find a perfect strategy to beat a powerful adversary (powerful in this case means that the adversary can exert significant control over state transitions, and can perform complex decision-making).

Let us take microarchitectural attacks as an example. Microarchitectural attacks such as cache timing side-channels, as well as branch prediction and prefetcher side-channels, abuse the fundamental mechanisms of how modern general-purpose processors achieve single-threaded speedups. The goal of these attacks is to modify system behavior and extract private information about the hardware, the operating system, or other victim processes running on the system. For example, these attacks may 'prime' a branch predictor or prefetcher into behaving in a way beneficial to the attacker, or may observe victim process behavior through an unintended side-channel such as the last level cache (LLC). These attacks are particularly insidious because they do not have easily recognizable payloads typical of other types of malware, but instead rely only on basic user-space computation to work. Nonetheless, these attacks may have recognizable signatures that a defender may learn to detect. By collecting microarchitectural data on these attacks, the defender may build run-time classifiers which can detect these attacks based on workload behaviors or hardware states, and either isolate them or kill their processes.

Here the difference between interacting with static environments vs. adversarial agents becomes apparent: if the environment was causing a side-channel, the system may be able to model it, prepare for it, and work around it, all ahead of time. However, a reasoning agent can change its behavior after the workaround, and continuously adapt as new defenses are put in place (typically referred to as 'a game of cat and mouse'). If a system is complex enough, the only viable strategy a defender has to completely stop

the attacker is to formally prove that **no** strategy can e.g., cause microarchitectural information leakage. This however is simply not practical as formal verification costs typically grow exponentially with system complexity. An alternative, less secure but more plausible strategy is to attempt to detect microarchitectural attacks, accept that attackers may find workarounds, but build the defenses flexibly enough so that the system may be either updated at run-time with new attack detection logic, or even *learn* on its own from detected attacks. This example illustrates that even with unlimited system insight and actuators, *without powerful and flexible decision-making, the adaptation benefits may not be useful or fully-utilized.*

2.3.3 The Need for Better System Adaptation Actuators: Better Energy Efficiency Through Cache Reconfiguration

Actuators are mechanisms available to the adaptive system to control its behavior or the environment. They may differ by their location in the system, type of control the exert, their temporal granularity, etc. While actuators may control their environment (e.g., changing the supply voltage of a CPU), this dissertation will focus mainly on microarchitectural actuators, i.e., the control mechanisms built from the same logic cells as the adaptive system, and controlling the same system. The following example will focus on last level caches (LLCs), since these caches typically have five or more classes of actuators, controlling cache size, number of sets, ways, line sizes, write policies, replacement policies, etc. The large number of actuators and the significant impact that each actuator can have on system behavior provides the system with the power and flexibility to adapt to workloads.

Modern caches commonly occupy more than 50% of processor die area, and use a

similar percentage of processor power. Many applications such as big data analytics, media streaming, or graph processing however have little benefit from large last level caches (LLC), so power spent on LLCs is wasted. Cache power utilization leads to lower energy efficiency, creating waste heat and preventing the core from achieving higher clock rates for longer periods of time. While poor cache utilization by these workloads is the problem, circumventing caches is not a winning strategy: though large data movements in these applications do thrash the cache and see large miss rates, control flow and latency-critical portions of code do benefit from smaller last level caches.

Reconfigurable caches have been proposed as a solution for helping these types of workloads achieve better energy efficiency. This type of caches adapt their microarchitecture at runtime to (1) increase hit rates through better workload - cache mapping, e.g., by changing their associativity when conflict miss rates are high, or (2) lower waste energy by power-gating cache sets or ways when miss rates when compulsory or capacity misses are high. Typically reconfigurable caches offer 'microarchitectural knobs' in the form of adjustable number of sets, ways, line sizes, replacement and write back/through policies, etc. Reconfiguring caches may require flushing dirty sets, ways, or even the whole cache, which incurs a significant latency and quality of service penalty. Therefore, judicious control over when and how the cache is reconfigured is needed. This example illustrates that actuators are the only mechanism with which the adaptive system controls itself and its environment, and that even with full workload insight and perfect decision-making capabilities, actuators of the right power, flexibility and granularity are required to extract adaptation benefits.

2.3.4 Summary

In summary, adaptive systems require sensing, acting, and decision-making capabilities to extract adaptation benefits. These capabilities must be tailored to each other, i.e., actuators must be able to control behavior that can affect a workload and system in a beneficial way, and sensing must be able to collect the necessary information about the workload and system. Whether decision-making capabilities are specific to the sensor-system-actuators tuple is not as clear.

Chapter 3

FORMAL DEFINITION OF ADAPTIVE SYSTEMS

In this chapter we define and provide a formal foundation for the terms *workloads*, *workload profiles*, *system profiles*, *static systems*, *statically-adaptive systems*, *dynamically-adaptive systems*, and *online-learning systems*. In Section 3.1 we introduce analytical formulations of different types of digital logic systems, and decompose complex and opaque system equations into smaller functions with more manageable numbers of inputs. In Section 3.2 we present several lossless and lossy simplifications and approximations to the systems equations introduced, which allow practical modeling of digital systems using machine learning and bound the modeling error.

3.1 General Systems Equations

The goal of this section is to:

- provide a set of generally-applicable system equations that can describe any digital logic system,
- illustrate the complexity and intractability of working with overly-general system equations, and
- motivate the use of simplified systems models.

Although this dissertation broadly targets general-purpose computing systems, for the sake of this simplifying analysis in this section we restrict systems to just general-purpose processors modeled after the Harvard architecture and described as state

machines. Without loss of generality, all of the conclusions presented in this chapter apply to other types of systems such as e.g., accelerators or HPC systems.

We define a *system* to be a state machine that operates on a number of workloads whose correctness is defined by an Instruction Set Architecture (ISA). The system is modeled as a Harvard machine with four components: a processing element, data memory, instruction memory, and I/O. We treat instruction memory as immutable[1] and a part of the inputs to the state machine, while we treat data memory as part of the state machine's state space. We define a workload to be an immutable (program, input data) tuple that deterministically defines workload execution, i.e., a workload targeting a specific system definition (e.g., ISA) will execute[2] one and only one sequence of system operations. We treat workload inputs and outputs as I/O (and not e.g., as data memory, even if a program statically allocates data), and therefore external to the state machine. A workload is implementation agnostic, and does not depend on e.g., whether a system may speculatively execute or execute out-of-order some operations. We also make two assumptions about all workloads: first, that all workloads terminate, and second, that all workloads are deterministic. Neither of these requirements are necessary for the following formulations, but they simplify the problem of defining systems models. Without the requirement that all programs terminate, the systems model output may be undefined for certain inputs, and without determinism, a single (program, input data) tuple no longer maps to a single output.

All of the systems considered here consist of a state machine that operates on

[1]This assumption is not restrictive, we later describe, the system can be described as operating on workload execution traces which are recorded beforehand and assumed immutable, instead of operating on a mutable array of instruction memory.

[2]A more cautious definition may use the term *commit* instead of execute, but we leave out this discussion for the sake of brevity.

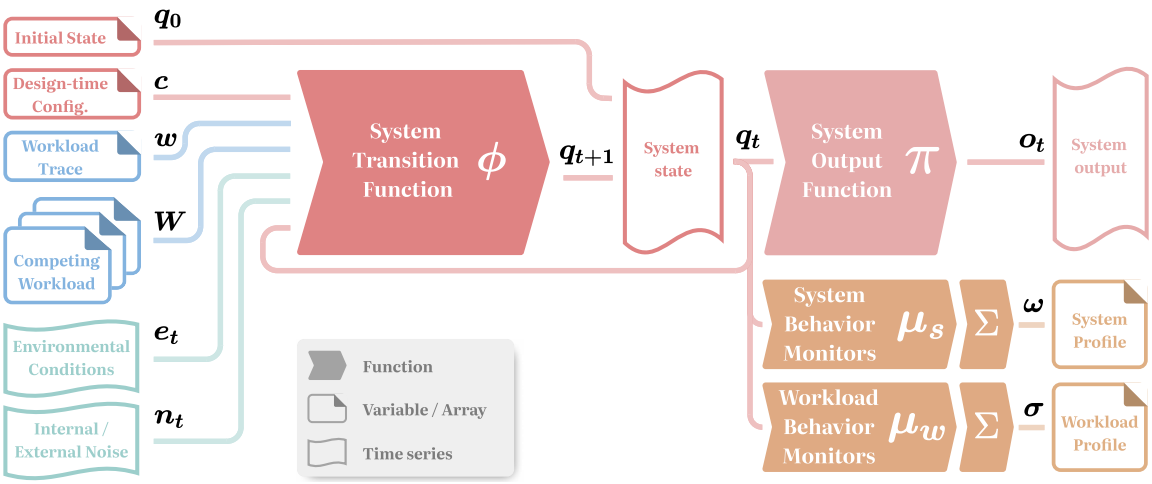


Figure 3.1: Diagram of a static system state machine showing the relationship between system inputs, system state, a state transition function, and system outputs and sensors.

a *workload of interest* specified as a (program, input data) tuple, along with other inputs such as competing workloads, environmental and system conditions, workload noise, and internal and external system noise[3].

3.1.1 Static Systems Equations

A non-adaptive (static) processor represented as the state machine described above can be diagrammed as shown in Figure 3.1. The system has a number of inputs:

- $w \in \mathbb{W}$: a workload of interest w drawn from a set of all workloads $\mathbb{W}$, with w represented as an array of instructions or execution trace, describing behavior of the workload according to some ISA,
- $W \subset \mathbb{W}$: a set of competing workloads described in the same format,

[3]Although a system may possess sources of true randomness, without loss of generality we can simplify and extract those sources outside of the system and make the system deterministic.

- c: a set of design-time decided system parameters,
- q_0: the initial state of all memory inside the system (including registers),
- e_t: a time series of external conditions affecting the system during runtime,
- n_t: a time series of both internal and external random noise affecting the system during runtime,

We choose to represent the system as a Moore machine, where the state of the machine at timestep $t+1$ is determined by the state transition function ϕ, current state q_t, static system inputs c, w and W, and time-dependent system inputs e_t and n_t. In a Moore machine, the system outputs o_t are only a function of the state q_t. Though not present in all digital designs, we add two additional outputs to the system in the form of system sensors or 'monitors' μ_s and μ_w, which observe system and workload behaviors. The monitors produce time series of sensor readouts, but because the stream collected from the sensors may be very large and often cannot be stored due to limited I/O bandwidth, system designers may choose to compress or filter the readings into some fixed-size log or profile (e.g., as in the case of hardware performance counters, workload profilers, etc.). In our formulation, the aggregation function Σ processes the time series from monitors μ_s and μ_o into fixed-size variables σ, the system behavior profile, and ω, the workload execution profile. Note that Σ is not necessarily the summation function, and can represent other aggregation functions such as e.g., maximum or minimum functions, different histogram functions, etc.

With these definitions in place, the static system's next state q_t, next output o_t,

and profiles σ and ω can be expressed as:

$$\begin{aligned} q_{t+1} &= \phi(w, W, c, q_t, e_t, n_t) \\ o_t &= \pi(q_t) \\ \omega &= \sum_{t=0}^{t_{halt}} \mu_w(q_t) \\ \sigma &= \sum_{t=0}^{t_{halt}} \mu_s(q_t) \end{aligned} \tag{3.1}$$

3.1.2 Statically-Adaptive Systems Equations

Statically-adaptive systems may change their configuration after fabrication to better fit workloads or current system state. This reconfiguration may happen at system boot time, when a certain workload has begun execution, or even periodically. The adaptation is static because it does not use any runtime information about the system or workload running on it, but relies only on data available ahead of configuration, e.g., a set of user goals or a workload profile. It is not triggered based on some runtime event during the execution of the workload or by some runtime decision-making system. The statically-adaptive system is unable to proactively respond to situations where a system reconfiguration may be beneficial. We define statically-adaptive systems as those that can change their configuration only at the beginning of execution of a workload of interest. Opposite to these systems, we define dynamically-adaptive systems as those that can change their configuration during the execution workload of interest.

Figure 3.2 extends Figure 3.1 by adding a configuration scheduler function ζ (shown in magenta) which accepts the workload of interest w and user goals g. User

goals g communicate the user's, the operating system's, or any other decision-maker's optimization objectives to the statically-adaptive system. The optimization objectives can be e.g., maximizing energy-efficiency or throughput, minimizing latency, improving security, etc. When ζ is executed prior to workload execution, it predicts which configuration c is best suited to the workload and goal at the time. This configuration c affects system behavior and the system transition function ϕ in some way, and can be treated as an input to the system, as static systems do in Equation 3.1.1. This formulation, however, hides system adaptation mechanisms or *adaptation actuators* inside ϕ, making such formalization moot. Instead, Figure 3.2 represents adaptation actuators by implementing a number of different system transition functions ϕ_1, ϕ_2, ..., ϕ_n, and the configuration c chooses which system transition function should be used to calculate the next state q_{t+1}. Each transition function describes how a system behaves when a different set of actuators is active. Although this formulation obviously leads to a combinatorial explosion in the number of transition functions since every combination of active or inactive actuators must be described, the goal of the formulation is not to describe a practical system, but to segregate the system into static and adaptive components. This formulation will be useful in Section 3.1.5.1 when discussing whether static and adaptive systems are fundamentally different types of machines.

Assuming that the workload of interest starts at time $t = 0$ and ends at time $t = t_{halt}$, and that the scheduler ζ produces a configuration c at the same time, the

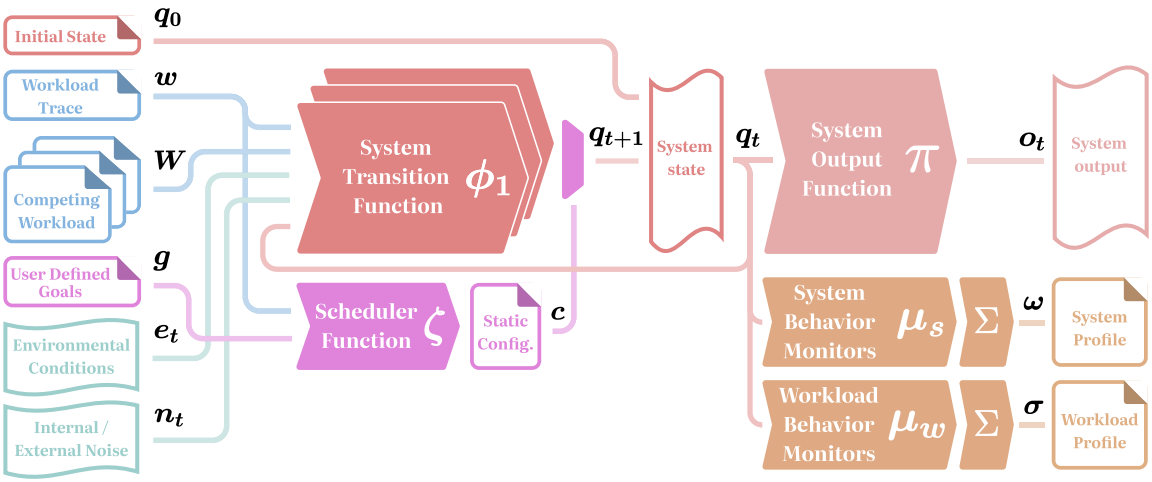


Figure 3.2: Diagram of a statically-adaptive system state machine, with changes necessary to enable static adaptation shown in purple.

statically-adaptive system shown in Figure 3.2 can be written as:

$$
\begin{aligned}
c &= \zeta(w, g) \\
q_{t+1} &= \begin{cases} \phi_1(w, W, q_t, e_t, n_t) & \text{for } \; c = 1 \\ \phi_2(w, W, q_t, e_t, n_t) & \text{for } \; c = 2 \\ \ldots & \\ \phi_n(w, W, q_t, e_t, n_t) & \text{for } \; c = n \end{cases} \\
o_t &= \pi(q_t) \\
\omega &= \sum_{t=0}^{t_{halt}} \mu_w(q_t) \\
\sigma &= \sum_{t=0}^{t_{halt}} \mu_s(q_t)
\end{aligned}
\tag{3.2}
$$

3.1.3 Dynamically-Adaptive Systems Equations

By definition, statically-adaptive systems are incapable of adapting to the changing runtime requirements of the user, the workload and the system. While they may potentially possess agency (the ability to reconfigure the system during the runtime), all of the inputs that the adaptation scheduler ζ receives are constant during the execution of a given workload. Without any visibility into the runtime conditions of the system, the scheduler and the system have to maintain the same configuration made when the workload of interest was initiated.

Dynamically-adaptive systems extend the system with a set of sensors that monitor the runtime state of the workload and the system, and use runtime decision-making to possibly change their decision during workload execution. The actuators available to the dynamically-adaptive system are a superset of those available to statically-adaptive systems. While statically-adaptive systems only decide between configurations at scheduling time, dynamically-adaptive systems can change between configurations at runtime, as well as trigger one-off behaviors implemented by the system (e.g., flush the cache). Dynamically-adaptive systems are therefore a superset of statically-adaptive systems.

Figure 3.3 shows a diagram of a dynamically-adaptive system. The system differs from statically-adaptive systems in two regards. First, the configuration scheduler function, which is invoked only when a workload is first scheduled, is replaced with a runtime decision making function, which is invoked continuously during execution. We will use ζ to represent both functions. Second, the adaptation decision making function receives an additional input δ_t, and has an updated function signature $c_{t+1} = \zeta(w, g_t, \delta_t)$. The new input δ_t is generated by system sensors $\delta_t = \mu_d(q_t)$. This

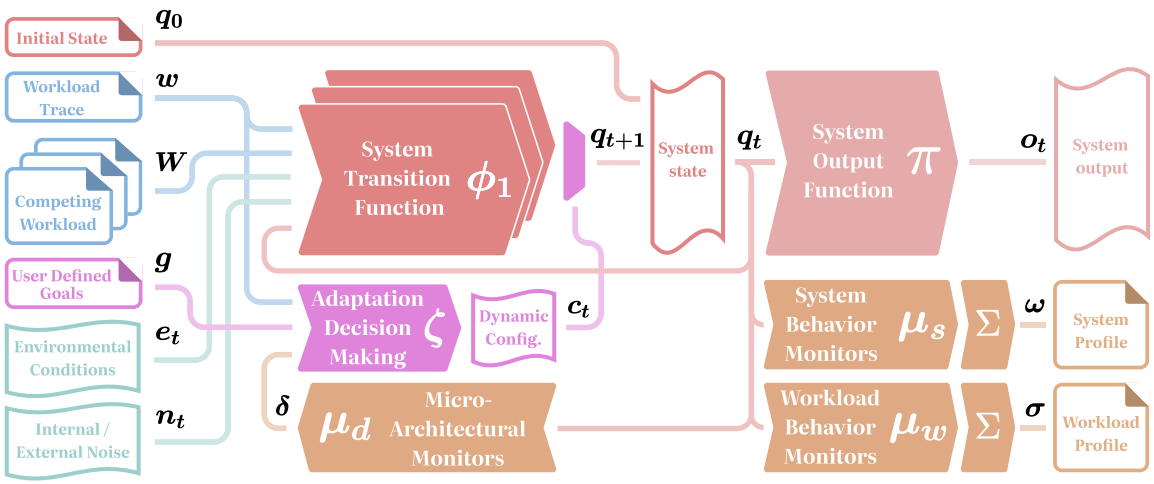


Figure 3.3: Diagram of a dynamically-adaptive system state machine, with highlighted changes necessary to enable static adaptation.

output is a time series and represents system state q_t (microarchitectural) monitor readings during the execution of the workload. Unlike in previous systems, sensor readings are not only compressed into profiles or logs for later analysis, but are also directly consumed by the adaptation decision making function ζ.

The adaptation decision making function ζ produces a time series of configurations c_t that it decides are beneficial for workload execution and *current* g_t goal satisfaction. Similarly as in the case of statically-adaptive systems, the adaptation configuration is used to select between different implementations of the system transition functions ϕ_1, ϕ_2, ..., ϕ_n. However, the dynamically-adaptive computing system can choose between functions during runtime, significantly broadening the space of system transitions, possible execution traces, and workload profiles. Furthermore, the runtime system behavior changes makes the system difficult to model, as transient configurations (which may be difficult to log and profile) may cause long-lasting effects on the system in certain situations.

The dynamically-adaptive system shown in Figure 3.3 can be written as:

$$
\begin{aligned}
\delta_t &= \mu_d(q_t) \\
c_{t+1} &= \zeta(w, g_t, \delta_t) \\
q_{t+1} &= \begin{cases} \phi_1(w, W, q_t, e_t, n_t) & \text{for } \; c_t = 1 \\ \phi_2(w, W, q_t, e_t, n_t) & \text{for } \; c_t = 2 \\ \ldots & \\ \phi_n(w, W, q_t, e_t, n_t) & \text{for } \; c_t = n \end{cases} \\
o_t &= \pi(q_t) \\
\omega &= \sum_{t=0}^{t_{halt}} \mu_w(q_t) \\
\sigma &= \sum_{t=0}^{t_{halt}} \mu_s(q_t)
\end{aligned}
\tag{3.3}
$$

Dynamically-adaptive systems are fundamentally different from statically-adaptive systems in three ways: (i) they have runtime access to system sensors δ_t, (ii) user goals g_t are allowed to change at runtime, and (iii) the decision-making system can update its decisions at runtime.

3.1.4 Online-Learning Dynamically-Adaptive Systems Equations

The behavior space of a system is the set of different behaviors a system can choose to exhibit. System actuators allow systems to change behavior at runtime, and each actuator can be treated as an orthogonal dimension in system behavior space. As non-adaptive systems cannot change behaviors at runtime, their behavior space is zero-dimensional, i.e., a single point. The behavior space of adaptive systems is a

Cartesian product $\mathbb{C} = \prod_i^n C_i$ of behaviors which the n different system actuators can exhibit. For example, an adaptive system that can set its frequency as either 800MHz, 1600MHz, and 3200MHz, set its cache associativity as either 2-way or 4-way, and select between two different branch predictors at runtime, can exhibit 12 different behavior configurations. Understanding the size of the behavior space is useful for bounding the search space of possible adaptation strategies and detecting whether the problem of finding an optimal adaptation strategy is tractable.

Not all points in the system behavior space may be valid, as some system configurations may violate correctness, and because some behaviors do not have viable physical implementations. The *feasible* space of behaviors an adaptive system can exhibit is limited by both the limitations of the system actuators and of the system decision-making logic. The first limitation is also the stronger one, where the boundaries of the system behavior space are defined by actuators capabilities, their runtime flexibility, and any resource constraints that may exist (e.g., limited power or thermal capacity prevents a design from continuously operating in some high-performance mode). These actuator boundaries implicitly define a set of feasible configurations $\mathbb{C}' \in \mathbb{C}$ in which statically- or dynamically-chosen configurations $c \in \mathbb{C}'$ reside in. Given a maximally-flexible decision-making function ζ_{ideal}, the range of $\zeta_{ideal} = \mathbb{C}$. In other words, the ideal decision-making function can order every feasible configuration of adaptation actuators.

Both statically- and dynamically-adaptive systems change their configuration in response to change in goal or workload. Additionally, dynamically-adaptive systems can change configuration in response to system state q_t as well. As q_t is affected by contention, environmental inputs, and noise inputs, dynamically-adaptive systems also respond to changes in those inputs as well. Holding these inputs constant (e.g., by

rerunning the same workloads under the same conditions), adaptive systems as they are presently defined *do not* change their behavior over time, and they are unable to improve their future behavior based on past experiences. This limitation is a product of the constant nature of the ϕ and ζ functions, as they do not change over time. Note that because dynamically-adaptive systems contain a feedback loop:

$$q_t \;\Rightarrow\; \mu_d(q_t) = \delta_t \;\Rightarrow\; \zeta(\delta_t, ...) = c_t \;\Rightarrow\; \phi_{c_t}(...) = q_{t+1} \;\Rightarrow\; ... \tag{3.4}$$

the memoryless function ζ can potentially use state q_t to store memory and affect its future behavior. This means that dynamically-adaptive systems *may* potentially exhibit behavior that changes over time.

Without the ability to purposefully affect future behavior, statically- and dynamically-adaptive systems cannot learn based on previous experience, and their behavior space is implicitly restricted by the capabilities of the existing decision-making function ζ.

We define 'learning systems' as systems that have purposeful (as opposed to incidental) functionality to affect future behavior based on data on previous workloads, system states, environmental effects, etc. Learning is typically classified as either *offline*, where data collection and learning are executed asynchronously prior to the current execution, and *online*, where data collection, learning and execution are performed in parallel. Online learning is a set of methods that train machine learning algorithms at runtime on the same data that the system is evaluated on. The benefit of online learning is that models can improve their predictions on inputs they initially mispredicted on, as well as adapt to any input data distribution shift. For example, online-learning adaptive systems may learn that e.g., previously unseen workloads benefit from some unexpected configuration and preserve this knowledge for future scheduling. Similarly, they may learn any change that happens in the environment

or the system, e.g., a component failing, and again improve future configuration predictions. Note that in computing systems capable of learning, the distinction between offline and online learning is less prevalent, as offline learning is rarely used due to limited communication between the deployed hardware and a hardware manufacturer that can perform offline training (Rubin et al. 1998). Therefore, we will use the term learning and online-learning synonymously. Computing systems that exhibit learning behavior typically do so online, where they actively observe workloads and update future behavior (e.g., in the case of branch prediction). Due to the complexity of interfacing with different operating systems to store learned behaviors, updates to the decision-making logic are typically not persistent and only last until the next system shutdown.

For computing systems to be learning systems as defined in Equations 3.1.1, 3.1.2 and 3.1.3, we require that (1) they can change their behavior function ϕ either ahead of program execution or at runtime, and (2) this change should be a function of past workloads W or past system states $\{q_0, q_1, ..., q_{t-1}\}$. We however restrict ourselves to only observing adaptive learning systems, where learning is confined to the scheduler ζ function. We define an adaptive learning system as one whose choice of behavior $c_t = \zeta(...)$ is not just a function of the present factors such as the current workload w or goal g, but also of past workloads and system behavior. The distinction here is that online-learning systems require that their behavior $\phi_(...)$ changes over time, and not just their inputs. While the system cannot change the space of available actuators and can only exhibit behaviors ϕ_1, ϕ_2, ..., ϕ_n, it can change which behavior is scheduled at which time. This distinction is useful to distinguish between non-learning dynamically-adaptive systems that nonetheless respond differently to repeated workload executions due to change in state q_t (even with all other system parameters held constant), and

online-learning dynamically-adaptive systems that change both their behavior, and their behavior decision-making during execution. Note that learning is independent from adaptation: a system may learn from workloads but lack the capability to act on any gained insight, or vice-versa, a system may have adaptation capabilities that are underutilized due to lack of learning capability.

Accepting that learning systems are distinguished by having a function ζ whose behavior may change over time, the decision-making function ζ in these learning systems can be classified into three categories (fourth one being memoryless, time-insensitive systems which cannot learn):

1. *Containing memory, time-insensitive* ζ: this class of decision-making functions is characterized by the fact that learning is relegated to external memory. For example, non-parametric models such as the K-Nearest Neighbors (KNN) algorithm are an example of such functions, since KNNs do not 'learn' (the algorithm does not change over time). Instead, the 'neighbor' points presented to the KNN algorithm may increase in number with each new KNN inference operation, and future KNN invocations can benefit from these new points.
2. *Memoryless, time-sensitive* ζ: this class of decision-making functions is characterized by the fact that learning modifies the function itself, and has no larger store of memory where data can be stored for later analysis. For example, online-trained neural networks are an example of such functions, since they may change their internal parameters during training, but any data they were trained on is potentially lost.
3. *Containing memory, time-sensitive* ζ: finally, this class of functions is characterized by having both an external store of knowledge which grows with new data, and also a function that changes with new data. Reinforcement learning

agents in e.g., self-driving cars are an example of such functions, since as they acquire new data, they may both update their internal ML models (e.g., neural networks), and also store data for later processing and active lookups by the models.

In this work we focus on memoryless, time-sensitive models, since only this class of models has fixed memory requirements that may be provisioned at design time.

3.1.5 Insights and Conclusions

The design shown in Figure 3.1 is general enough to represent any digital logic system. However, systems equations shown here are incredibly complex when used to describe existing digital systems. The complexity stems both from their very large state space and transition function, and the large amount of iterations (steps, cycles, etc.) needed to complete workload execution. As such, building systems equations is equivalent to building a full system implementation. Although the Moore machine abstraction is helpful in clearly defining the system interface, it neither: (1) improves our ability to reason about the system at a higher level of abstraction, nor (2) allows us to more efficiently execute the system. For that reason, in next section we turn to *analytical models* of computing systems, which attempt to define lossless and lossy analytical transformations that can be applied to models of the system, in order to simplify them and make them practical.

The formulations in Sections 3.1.1, 3.1.2, and 3.1.3 are useful for several reasons. First, they define how workload and system profiles are created by real systems, as these profiles will be used in Chapter 6, where machine learning models will be tasked with predicting system profiles from workload profiles in an attempt to learn

(a portion of the) system's behavior ϕ. Second, in Section 3.1.5.1 they help clarify the differences between static and dynamic systems, and classify existing systems or system components into the two categories. Finally, in Sections 3.1.5.2 and 3.1.5.3, they are useful in explaining the theoretical capabilities of different types of models when faced with novel or changing workloads and systems.

3.1.5.1 Should Existing Computing Systems Be Considered Dynamically Adaptive?

Note that the three equations for static, statically-adaptive, and dynamically-adaptive systems are functionally equivalent, and only differ in additional external inputs. For example, the scheduling function in Section 3.1.2 has an additional input g and a new scheduling function $\zeta(w, g)$. If ζ is subsumed into the system equation ϕ and ϕ receives and additional input g, this new function $\phi'(g, w, W, q_t, e_t, n_t)$ can represent any statically-adaptive system. As the statically-adaptive system behavior function ϕ' only differs from static systems behavior function ϕ by the additional static user goal input g, statically-adaptive systems are a superset of static systems. Furthermore, the two sets of machines are equivalent if g is known at design time, as information about g can be used to choose the static configuration c. A similar case can be made for dynamically-adaptive systems, as many existing systems exhibit behavior that fits within the definition of dynamically-adaptive systems. For example, branch predictors in modern processors (1) sense internal states (e.g., workload control flow, instruction counters, etc.) during runtime, (2) make decisions on what new state or behavior would be beneficial (e.g., update a branch history table), and (3) modify system behavior (e.g., speculate on this branch). In response to this classification, one may choose to distinguish between static and dynamically-adaptive systems by the

fact that the latter can choose between multiple distinct behaviors ϕ_1, ϕ_2, ..., ϕ_n at runtime. However, even though there may exist a practical distinction between these behaviors (e.g., completely separate implementations), one can always construct a unifying static system as shown by the piecewise definitions of q_{t+1} in Equations 3.1.2 and 3.1.3.

The question of whether a system is static or adaptive may appear purely pedantic, but it is important for designers to know the type of system they are designing in order to apply appropriate adaptive system design methodology. It also raises further questions about existing designs, as mechanisms such as branch prediction can be viewed as both a part of the conventional system behavior, or as a dynamically-adaptive feature of a system. In the first view, although the branch predictor may adapt to different workloads, this adaptation always follows the same course and offers no flexibility if user goals or the computing environment changes. This causes the system to appear static (non-adaptive) if viewed from high-enough granularity. In the second view, branch prediction can be shown to have a great impact on certain workloads that typically perform poorly on processors without branch prediction, all owing to the hard-wired observe-decide-act loop integrated in the predictor. By analyzing existing computing system designs within the framework presented in this chapter, new questions about their capability and efficiency can be raised, such as why branch predictors do not possess programmable goals (e.g., when a user is executing a security-sensitive piece of code, they may want to stop hardware speculation), or why hardware prefetchers are not sensitive to cache thrashing by other concurrent processes.

There exist several properties that are connected to, but do not necessarily imply that a system is adaptive:

- the system exhibits workload-dependent behavior,
- the system exhibits goal-dependent behavior,
- the system exhibits external stimuli-dependent behavior,
- the system possesses behavior-affecting programmability and online learning capability.

Systems that exhibit (1) workload-dependent behavior can change their behavior depending on workload properties in ways that do not just conform to the correctness specification, but optimize for other (possibly hard-wired) goals such as performance, energy efficiency, or security. If this property is sufficient to classify a system as adaptive, processors with hardware prefetchers or branch predictors should be classified as dynamically-adaptive. Systems that (2) observe user goals can optimize their behavior to better fit those goals and achieve higher quality of service (which is defined with respect to the current goal) compared to static systems. If this property is sufficient to classify a system as adaptive, then systems that support e.g., BIOS-controllable hardware prefetching or multithreading should be classified as statically-adaptive, and systems that support e.g., runtime-controllable power profiles should be considered dynamically-adaptive. Systems that (3) react to not just the workload of interest or goals, but also to outside stimuli such as resource contention from other workloads W, environmental stimuli e_t or inherent system noise n_t can achieve *graceful degradation* as the computing environment conditions worsen. If this property is sufficient to classify a system as adaptive, systems that e.g., observe cache miss rates and isolate thrashing processes for performance or security reasons can be treated as dynamically-adaptive (Novaković et al. 2013). Systems that (4) continuously observe their adaptation decision quality and learn based on feedback gained during runtime can adapt to unforeseen deployment environments or novel workloads. If this property

is sufficient to classify a system as adaptive, then systems that support e.g., learning schedulers can be classified as online-learning statically-adaptive systems.

While there exists no clear agreement on necessary and sufficient properties that make up a statically- or dynamically-adaptive system, in the opinion of the author, adaptation is beneficial in situations where design-time assumptions fail, and where runtime data offers insights that may be beneficial to the system. All four of the properties above utilize runtime information unavailable at design time to increase system quality of service, and in the rest of this dissertation will treat all of these properties as sufficient to consider a system adaptive. Therefore, systems that contain branch prediction, learning schedulers, and user-controllable power profiles, will all be considered adaptive.

3.1.5.2 Capabilities for Adapting to Dynamically Changing Conditions

With the static, statically-adaptive, dynamically-adaptive, and online-learning system formulations provided above, we can reason about the capabilities of these systems and how they perform in different environments. As Chapter 2 illustrates, static systems witness degraded goal satisfaction in conditions different from what system designers have explicitly provisioned for. Successfully satisfying user goals such as energy efficiency or performance during periods of changing conditions (contention, environments, user goals, or noise) requires modifying behavior to best fit those conditions.

Static adaptation can help adapt the system to slow changing environments where the average time to execute a workload is shorter than the time between changing environmental conditions. Dynamic adaptation is required for fast changing

environments where environmental conditions may change before a workload completes and any learning mechanism can be executed.

3.1.5.3 Capabilities for Adapting to Novel Workloads and Conditions

While systems may adapt to the changing environment (changing goals, contention, etc.), they may also have to adapt to changing workload distributions. This is witnessed when the distribution of workloads at runtime does not match the workload distribution system parameters were tuned for at design time, and is a called concept drift (Madireddy et al. 2019a). Though the behavior space of an adaptive system may consist of hundreds of actuators, this high-dimensional space is dwarfed by the space of possible workloads. Such high-dimensional spaces are difficult to explore and describe, hence finding optimal system configurations for novel workloads is more difficult than finding optimal configurations for novel environments. Some workloads may potentially be pathological and unless the model has previously seen this type of workload, the chosen system configurations are likely to be suboptimal. Furthermore, while the external environment may change slowly, workloads may contain many phases which warrant completely different system configurations, requiring dynamic adaptation.

3.2 Workloads and Systems Modeling Preliminaries

Above listed systems equations are expressive enough to represent any digital logic, but are too complex to practically describe complex systems. In order to understand how these systems work and what adaptation features may improve their

functionality, we seek to simplify the above system equations into more tractable forms. We will simplify the equations by first (1) replacing functions ϕ, ζ, and μ with *analytical models* that operate on compressed forms of the inputs, and later by (2) building practical, data-driven machine learning models in place of these analytical models. In this section we present the necessary methods needed to simplify equations from Section 3.1 and bound their inputs, converting the equations into formats that conventional machine learning algorithms can interface with.

We define a *systems model* to be an equation or algorithm that takes as input either a workload w or workload profile ω and predicts some subset of the system profile σ after the execution of the workload. These system profile subsets can be execution time, energy usage, network bandwidth, I/O throughput, etc. We highlight 'systems' in systems models, since the system modeling domain has several properties distinct from other modeling domains which we will discuss later in this chapter. Systems models differ from systems equations from Section 3.1 since the models are not perfect representations of the system and sacrifice veracity for practicality of construction.

While the systems equations 3.1.1, 3.1.2, and 3.1.3 calculate both the system outputs o_t as well as system and workload profiles σ and ω, the models we are concerned with only attempt to predict the profiles. Though the two profiles are direct results of system state time series q_t, systems models, unlike systems equations, do not necessarily simulate the inner state of the system they model, but instead attempt to shortcut the path between system inputs (e.g., workload w) and the system profile σ.

3.2.1 Domain Assumptions

We will make several assumptions about the workloads and systems modeled in this dissertation:

- All workloads terminate, i.e., always halt. While not strictly necessary, this property forces a systems model to always have a defined function range.
- Workloads and systems are deterministic, and systems may have separate internal and external noise sources that the model treats as inputs.
- Systems are general-purpose computing systems, can be modeled as Harvard machines, and their behavior is defined by an Instruction Set Architecture (ISA) and memory model.
- Systems can execute multiple workloads in parallel but choose to not isolate them from each other.
- Initial system conditions have negligible impact on long-running (hot) systems.
- Workload-system interactions do not exhibit long temporal dependencies, i.e., the system can only be impacted by recent workload behavior.
- Systems and workloads are not actively adversarial and do not purposefully resist modeling by exhibiting unpredictable or pathological behavior.

Before we can present analytical models of the four classes of system, we introduce several modeling simplifications.

3.2.2 Modeling simplifications

Modeling systems and achieving perfect modeling accuracy may require building models of the same order of complexity as the system being modeled[4]. Building such fully-accurate models is not useful however, since the goal of models is to more easily represent the object of modeling, to decrease the time or computation needed to evaluate the object, or to abstract away less important aspects of the object being modeled.

In this section, I will outline three modeling simplifications that help build useful models of computing systems. The shared thread between these simplifications is that they replace variable-sized inputs with fixed-sized inputs. More specifically, while equations commonly accept elements from some sets as inputs, systems models are often tasked with working on unordered sets (e.g., sets of competing workloads), ordered lists (e.g., instruction traces) and time series (e.g., data streams), and unbounded values (e.g., unbounded program inputs).

The simplifications are:

1. Replacing variable-sized function inputs with constant-sized inputs.
2. Replacing an arbitrary number of concurrent workloads with a single, unified proxy workload.
3. Replacing nonstationary functions with stationary alternatives extended with additional timing inputs.

[4]While this may not be immediately clear, we present the following sketch proof. Assume the opposite, i.e., that for any system, a simpler model that identically reproduces the system output can be built. Then, construct a system whose e.g., output, runtime, or some other value is calculated as a cryptographic function, e.g., hash of inputs and some internal state. To accurately describe system behavior, one cannot simulate the system as a black box, but must emulate its internals and have full knowledge of both inputs and internal states. Obviously, this system cannot be simplified without sacrificing accuracy, and is a counterexample to the above assumption.

3.2.2.1 Abstracting Away Variable-Sized Inputs

Analytical and ML models of systems may encounter variable-sized inputs where one or more input dimensions is unknown until the input values are provided. For example, a single workload can be represented either as a (program, input data) tuple, or as a stream of instructions that must be executed. Both the program and the instruction stream have a variable size, though programs are typically bounded in size while instruction streams can potentially contain an infinite number of instructions if the workload never terminates. Here I seek a method to represent generic variable-sized inputs as constant-size inputs in order to enable application of models that only support fixed-sized inputs.

Constant-size representations of variable-sized inputs are by definition lossy approximations, since any input larger than the provisioned constant cannot be represented without information loss. While this property in general reduces model accuracy, it provides a predictable model runtime that enables the use of ML models in production systems (e.g., CPUs, databases, and HPC systems). Before a variable-sized input can be fed into a model, it must first be processed by an aggregation function $\sum$ which produces a constant-sized output. For example, given a variable-length workload, the aggregation function may produce a fixed-length workload profile. System models we study can directly form predictions based on workload profiles, learning to map measured (system configuration, workload profile) tuples to measured system profiles. More formally, given an system defined as:

$$q_{t+1} = \phi_c(w, q_t, ...); \tag{3.5}$$

these system equations can be seen as mapping the configuration c, the system function ϕ_c, the workload w, and other inputs (contention, environment, noise) to a time series

of system states q_t. Next, the system equations map system states to workload and system profiles ω and σ as:

$$\omega = \sum_{t=0}^{t_{halt}} \mu_w(q_t); \qquad \sigma \ = \sum_{t=0}^{t_{halt}} \mu_s(q_t) \tag{3.6}$$

The goal of an ML model f is to learn the system profile σ without learning the time series q_t. The ML model is tasked with mapping the system function ϕ_c, configuration c and workload profile ω to measured system profile σ, and can be written as:

$$\bar{\sigma} = f(\omega, \phi, c) \tag{3.7}$$

The model f attempts to minimize the difference between predicted system profile $\bar{\sigma}$ and measured system profile σ:

$$\min_f ||\sigma - \bar{\sigma}||_2 \tag{3.8}$$

With this simplification, as long as an aggregation function that preserves most important workload features can be found, conventional ML models such as gradient boosting machines or neural networks can be used in place of f.

3.2.2.2 Abstracting Away Workload Plurality

Most general-purpose computing systems will run tens or hundreds of different workloads at a given time. These workloads are contending for shared system resources and negatively impacting each other's ability to satisfy goals such as high throughput and predictable latency. Furthermore, due to the complex nature of the underlying system, inability of the system to predict future workload behavior, and unclear interactions between the workloads on a shared medium, the interfering workloads prevent accurate estimates of the behavior of a specific *workload of interest*.

When modeling the interaction between a system and a workload of interest, it is important to account for how all other workloads affect this target workload. There exist two difficulties analytical models must solve in order to process such inputs: first, since the number of competing workloads is unbounded and changes over time, systems models need to somehow account for the varying number of input workloads. Second, since there is no imposed order to the workloads running on the system, the models need to satisfy input permutation invariance, i.e., changing the order of how workloads are fed to the model should not affect the model outputs. These problems are not restricted to just analytical models, as machine learning models will need to overcome these obstacles too.

The above problem can be more clearly and abstractly defined by specifying that if the systems model is a function f operating on a number of workloads W, the domain of that function is not an element, but a set of elements with unbounded cardinality. Both requirements listed above directly flow from this formulation. Given a set of all workloads $\mathbb{W}$ from which the workload of interest $w_i \in \mathbb{W}$ and a set of competing workloads $W \subseteq \mathbb{W}$ are drawn, and temporarily ignoring the other inputs to the model, a systems model defined as $\bar{\sigma} = f(w_i, W)$ has the domain and range:

$$f : \mathbb{W} \times 2^{\mathbb{W}} \rightarrow \mathbb{R} \tag{3.9}$$

where $2^{\mathbb{W}}$ is the power set of $\mathbb{W}$. Given a workload of interest w_i and a set of n competing workloads $W = \{w_1, w_2, ..., w_n\}$, for the model f to be insensitive to the workload ordering, it must satisfy the property:

$$f(w_i,\ w_1, w_2, ..., w_n) = f(w_i,\ w_{\pi(1)}, w_{\pi(2)}, ..., w_{\pi(n)}) \tag{3.10}$$

for any permutation function $\pi : \{1, 2, ..., n\} \xrightarrow{1\text{-}1} \{1, 2, ..., n\}$.

Due to both its complex signature, as well as the difficulty of forcing ML models to be commutative with respect to inputs, we seek to split model f into functions p and $\bar{f}$ such that:

$$\begin{aligned} p &: 2^{\mathbb{W}} \to \mathbb{W} \\ \bar{f} &: \mathbb{W} \times \mathbb{W} \to \mathbb{R} \end{aligned} \tag{3.11}$$

The functions are chosen to minimize:

$$\min_{p,\, \bar{f}} = \Big|\Big| f(w_i,\ W) - \bar{f}(w_i,\ p(W)) \Big|\Big| \tag{3.12}$$

Here, p is a *proxy contention* function which maps a plurality of workloads $W \subseteq \mathbb{W}$ to a single proxy workload $p(W) \in \mathbb{W}$ which should be approximately equivalent to W in terms of impact on w_i. The model $f(w_i, W)$ is replaced with an updated model $\bar{f}(w_i, p(W)$, which is now a function of only two and not $n+1$ workloads. Note that the reduction from W to $p(W)$ cannot be perfect due to the restricted amount of information that $p(W)$ can carry. Nonetheless, in our analytical models we will assume that good approximations can be found and that competing workloads can be replaced with a single proxy workload.

In terms of a practical implementation, most machine learning models do not accept sets as input arguments, and working on sets has historically remained in the domain of classical algorithms. One of the recent ML approaches that supports variable-sized and permutation-invariant inputs is Deep Sets (Zaheer et al. 2017). Here, authors prove the following theorem, presented with minimal notational changes:

Theorem 1 *A function $f(W)$ operating on a set W having elements from a countable universe, is a valid set function, i.e., invariant to the permutation of instances in W, iff it can be decomposed as $f(W) = \alpha(\sum_{w \in W} \beta(w))$, for suitable functions α and β.*

Through this theorem, we can ensure that the proxy contention function p is commutative and permutation invariant by expressing it using α and β, allowing us to replace a variable number of workloads causing contention on the workload of interest with a single encompassing variable. Additionally, under this definition, all functions have a fixed number of input arguments, simplifying both notation and practical implementation. Referring the previous formulation, if acceptable α and β are found, the contention function p can be formulated as:

$$p(W) = \sum_{w \in W} \beta(w) \tag{3.13}$$

and the updated systems model can: be formulated as

$$f(w_i,\ W) \approx \bar{f}(w_i,\ \alpha(p(W))) \tag{3.14}$$

Since α can be subsumed into $\bar{f}$, we can instead define a third model $\hat{f}$:

$$\bar{f}(w_i,\ \alpha(p(W))) = \hat{f}(w_i,\ p(W)) \tag{3.15}$$

Even though this transformation does not reduce the complexity of practical models (measured by e.g., number of parameters), it allows us to both (1) replace variable-sized inputs with fixed-size inputs, and (2) decompose large 'system' functions into smaller functions that separately learn how workload contention composes, and how the system handles an abstract and unified contention proxy $p(W)$ instead of multiple, 'raw' workloads $w \in W$.

3.2.2.3 Abstracting Away Nonstationary Functions

In statistics, stationary processes are stochastic processes whose probability distribution does not change over time. An example of a stationary process is the Bernoulli

process, a series of (possibly biassed) coin tosses. Since coin tosses are not affected by the time, the Bernoulli process is stationary. On the other hand, non-stationary processes *are* affected by time. For example, a process generating hourly temperature measurements at a certain location will be affected by both the time of day and the current date. Both the time and date have a strong effect on temperature mean and variance. We use the terms stationary and nonstationary to distinguish between functions whose outcomes depend on time, and stationary functions which are independent of the time. In a similar vein to the previous simplification, we seek to replace nonstationary functions with stationary functions that have additional temporal inputs. Such a simplification will useful when modeling adaptive systems, since these systems can alternate between different behaviors at runtime, so the function that describes the system is nonstationary.

Assume that a system being modeled changes its behavior over time, e.g., before some time t_x it is governed by function $\phi_{t<t_x}(I)$, while after t_x it is governed by $\phi_{t\geq t_x}(I)$, where I is some set of inputs. Now, we seek a generalized *stationary* function ϕ that has both I and time t as an input, but does not change over time. Such a function can be expressed as:

$$\phi(t, I) = \begin{cases} \phi_{t<t_x}(I) & \text{if } t < t_x \\ \phi_{t\geq t_x}(I) & \text{if } t \geq t_x \end{cases} \tag{3.16}$$

Obviously, a nonstationary function can be replaced with a stationary function that implements all of the variants the nonstationary function can exhibit over time and chooses between them based on the time input. We will use this property to describe adaptive systems whose behavior can depend on what time the system is run.

3.3 Workloads and Workload Profiles

The problem with system models operating on workloads is that workloads are of arbitrary size. While not a problem for analytical models, it is beneficial for practical models to work with fixed function interfaces. Additionally, it may be computationally infeasible to maintain workload - input pairs and feed these to systems models, since these systems models will have to significant system functionality in order to evaluate workload - system interactions. Historically, large processor manufacturers have replaced workload - input pairs with simpler and more deterministic instruction streams, in order to reduce evaluation time and increase benchmarking reproducibility. On the modeling side, multiple prior works modeling systems and workloads assume that workloads have a fixed-size *workload profile*, which is then fed into models (Madireddy et al. 2018b; Isakov et al. 2020). This profile may be application specific, but generally it consists of a set of parameters that describe application characteristics. The choice of which workload characteristics to include in the limited space provided by the profile is guided by which characteristics help the systems model most in modeling the original workload. While we will discuss how to select these characteristics in Section 6.5.1, for now we assume that such a selection is provided.

Without loss of generality, we will focus on general-purpose processors in order to use existing notions of instructions and bits. The definition can be generalized from processors to systems with only minor notational difficulties. We will assume the existence of a systems model $s(\mathbf{p}, \mathbf{d})$ that given a workload, predicts e.g., the number of instructions the system will commit [5]. The workload passed to the model is defined

[5]We specifically make the system function only a function of the workload and not any system properties, in order to avoid having to introduce additional variables that do not aid the argument.

as a (program $\mathbf{p}$, input data $\mathbf{d}$) tuple. The program $\mathbf{p}$ consists of n instructions $\mathbf{p} = (i_1, i_2, ..., i_n),\ i_j \in \mathbb{I}, n \in \mathbb{N}$, where $\mathbb{I}$ is the *finite* set of all instructions. Borrowing notation, we say that a program is therefore *a string* of arbitrary size formed from alphabet $\mathbb{I}$. We define $\mathbb{P}$ as a language, i.e., the set of all strings over the alphabet $\mathbb{I}$. The input data $\mathbf{d} \in mathbbN$ is unbounded but can always be represented as an positive integer. As we can see, the signature of the system model s is $s : \mathbb{P} \times \mathbb{N} \rightarrow \mathbb{N}$.

The question of how to represent a potentially unbounded workloads in a finite profile remains. The above function signature is problematic both because the string $\mathbf{p} \in \mathbb{P}$ is unbounded in length, and because $\mathbf{d}$ is unbounded in size. First, language $\mathbb{P}$ can be replaced with the set of positive integers $\mathbb{N}_0$ by performing the following substitution: any arbitrary-sized program $\mathbf{p}$, can be represented as positive integer value p as: $p = \sum_{x=1}^{n} i_x 2^{bx}$ [6]. Therefore, the new signature of the systems model is $s : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$.

Next, given a limited size of b_i bits to store the workload profile $\omega \in \mathbb{B}_i,\ \mathbb{B}_i = \{0, 1, ..., 2^{b_i} - 1\}$, we seek to find an approximation $\bar{s} : \mathbb{B}_i \rightarrow \mathbb{B}_o$ of the systems model s, where $\mathbb{B}_o$ is defined similarly to $\mathbb{B}_i$, but with b_o bits. We say that $\bar{s}$ is an approximation due to the fact that $\bar{s}$ has a finite while s has an infinite domain. Given that workload profiles are restricted in size, for each specific workload profile there exist infinitely many programs that map to it.

A deterministic system-independent function (e.g., number of instructions to execute a program) can be written as $e = s(\mathbf{p}, \mathbf{d}),\ p, d, e \in \mathbb{N}$ where p and d are the program and input data, and e is the number of executed instructions. Obviously, ignoring the halting problem for a moment, there exists a function $\bar{s} : \mathbb{N}^2 \rightarrow \mathbb{N}$ that maps p and d to e.

[6] For sake of notation, we add 1 to make the domain $\mathbb{N}$ instead of $\mathbb{N}_0$

The problem with $\bar{s}$ is that it has replaced an arbitrary number of bounded inputs with a constant number of unbounded inputs. Since we are interested in practical and not just analytical modeling, we are forced to work with bounded inputs. Therefore, we seek to approximate $\bar{s}$ with a

an arbitrary-sized program $p \in \mathbb{N}$ is executed on a system or the system is simulated, and a program profile $\omega \in [0, 2^B - 1]$ is collected. This profile represents the behavior of the workload during execution (e.g,. instruction type distribution, average data dependency values, etc.) and its effects on the system (L1/L2/L3 cache miss rates, types of conflicts, page fault rates, etc.). These elements of the program profile are selected to achieve best possible system model accuracy.

Of course, we have no method to bound approximation error, since compressing an arbitrarily-sized program to a fixed size profile is necessarily lossy whenever the program is larger than the profile. For any modeling error bound we could provide, it would be possible to create an adversarial program that has greater error after approximated. For example, knowing e.g., average L3 cache miss rates is not enough information to perfectly predict the runtime of a program, nor is even knowing the full L3 access time distribution.

There are multiple isomorphic ways a workload may be fed into the system:

- As an instruction list - data tuple, where system simulation is needed to determine how the control flow will be resolved,
- As an instruction trace - initial memory state that will be executed, e.g., collected from a running processor

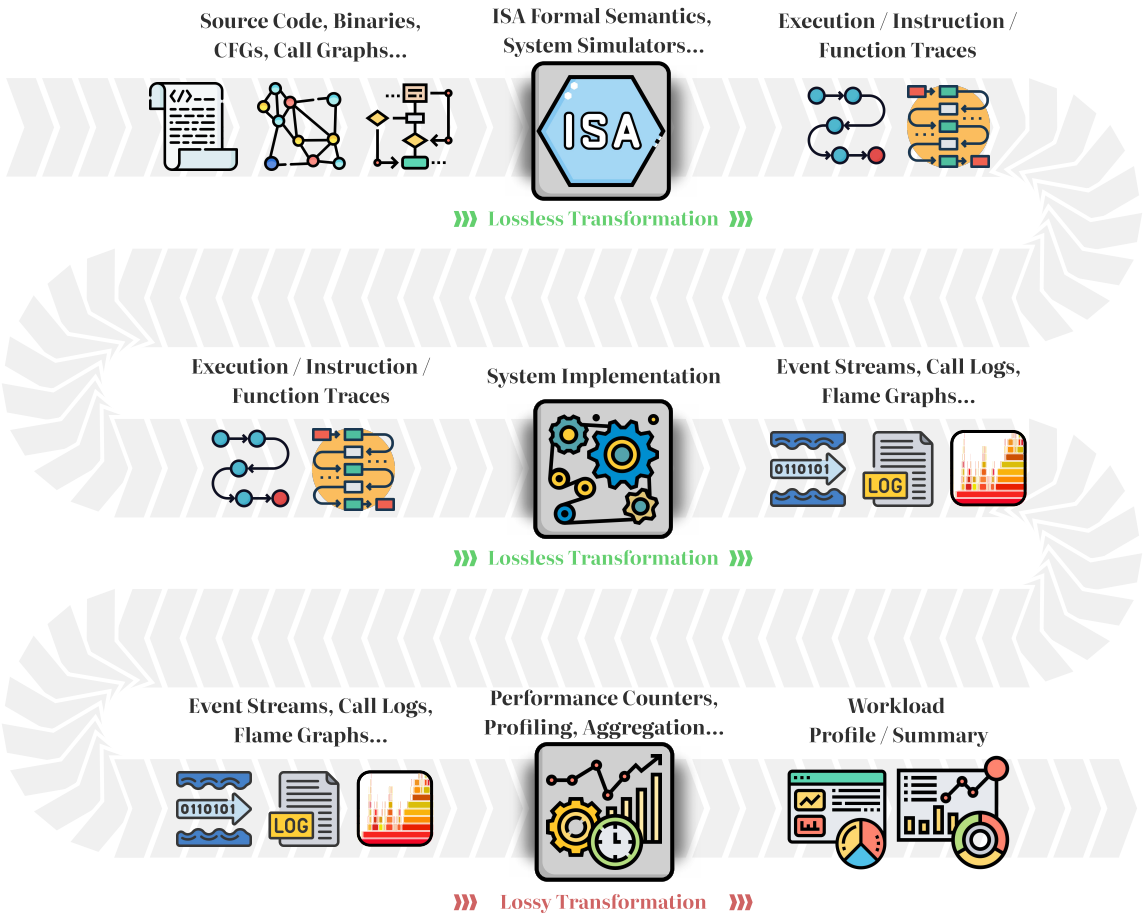


Figure 3.4: Transformation steps and intermediate results of building a workload profile.

Chapter 4

EXPERIMENTAL SETUP

4.1 HPC System and Workload Datasets

Although the end goal of this dissertation is to be develop a methodology applicable to arbitrary computing systems, I first seek an experimental environment that provides enough visibility into a system so that system insight can be developed. Once such knowledge is gained, then the viability of adaptation can be further studied.

There are several requirements for a compute environment that will be the subject of workload and system modeling. First, the set of workloads should be broad enough that the system requires non-trivial functionality and generality. For example, machine learning hardware accelerators are not a good target, despite the existence of thousands of different neural network models whose execution may be studied, since ML workloads do not significantly differ in behavior and they exhibit little performance variance run-over-run compared to general-purpose workloads. Second, the system should exhibit enough complexity to require complex analytical or ML models. For example, simple in-order cores rarely exhibit pathological behavior on certain workloads, and simpler models may describe their performance. Third, the system must provide sufficient insight into its internals out-of-the-box. Hence, commercial processors are not a good target, since no microarchitectural models or RTL are available.

One candidate domain which fulfills all three requirements (diverse workloads, complex systems, visible internals) are High-Performance Computing (HPC) storage systems. The input-output (I/O) subsystems of large supercomputers perform complex

operations with often highly unpredictable responses to different workloads, the system states (e.g., filesystem health) can have a significant impact on throughput, and they are slow enough (requests take milliseconds) to be observed from software without logging becoming an impediment. This chapter focuses on I/O subsystem modeling, as the I/O subsystem is observable from software due to lower request frequencies and limited hardware acceleration, while at the same time I/O bottlenecks are harder to diagnose due to lack of tooling.

4.1.1 High-Performance Computing System Architectures

High Performance Computing (HPC) systems or 'supercomputers' are a good target for the modeling portion of this dissertation, for several reasons. First, modern HPC systems are extremely complex, having thousands of nodes (individual but networked computing systems), and tens or hundreds of thousands of cores on aggregate. They typically have high bandwidth and low latency network fabrics connecting these nodes, and have both node-local storage as well as a separate, parallel distributed filesystems accessed over the network. These filesystems will have separate servers for aggregating requests, hosting file metadata, and actual storage arrays with hard disk drives (HDD), solid state drives (SSD), or magnetic tape. Between these distributed filesystems and the compute nodes may exist additional levels of caching, such as the more recent SSD-based burst buffers. Figure 4.1 illustrates just such a system.

In this chapter I evaluate two Department of Energy (DOE) leadership-class supercomputers: the Argonne National Laboratory Theta system, and the Lawrence Berkeley Cori system.

Theta is a Cray XC40 system with 24 racks and 183 nodes per rack delivering

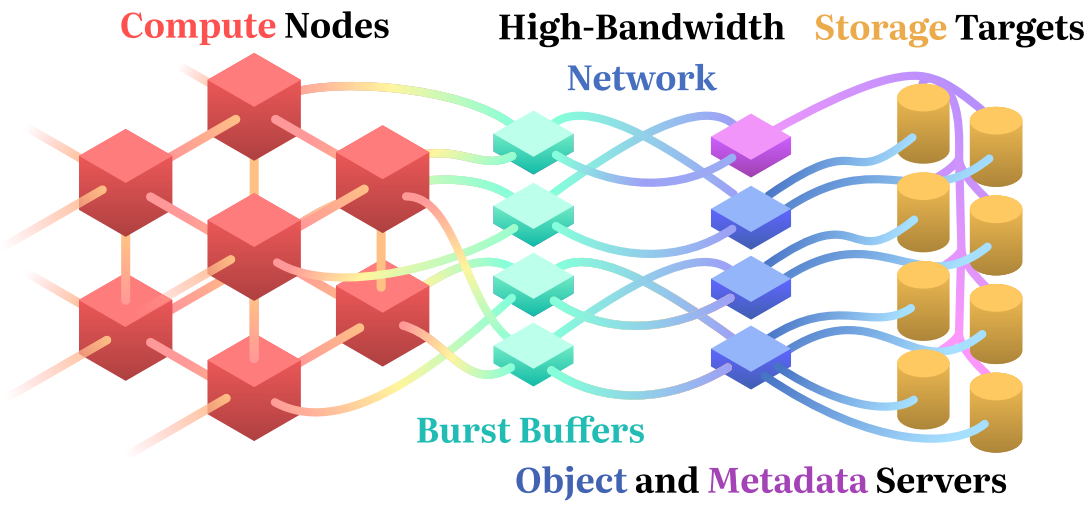


Figure 4.1: The high-level infrastructure of a modern high performance computing system.

11.7 petaflops of computer power. Each compute node has a single-socket Intel 7230 Knight's Landing (KNL) CPU — a 64 core processor with 16 GiB of on-chip DRAM, and also contains 192 GiB of DDR4 DRAM, as well as a 128GiB solid state drive. In aggregate, Theta has 4,392 compute nodes with 281,088 cores, 843,264 GiB DDR4 and 70,272 GiB of on-chip DRAM. Theta is connected to four different distributed filesystems, averaging between 10 and 100 petabytes (PiB) of storage, with I/O throughputs in the hundreds of GiB/s. For example, the majority of projects on Theta in the period between 2017 and 2020 used the `theta-fs0` filesystem, which is a 9.2 PiB Lustre file system delivering 240 GiB/s of I/O throughput.

Similarly to Theta, Cori is also a Cray XC40 system, but with 54 racks housing 9,688 Intel KNL nodes and delivering 29.5 petaflops of performance. Each node has a single 68 core processor, 16 GiB of on-chip and 96 GiB of DDR4 DRAM, with an aggregate of 658,784 cores and 1.06 PiB of DRAM. Cori is connected to the Cori-scratch, a Lustre filesystem with 30 PiB of storage and 700 GiB/s of aggregate

I/O throughput. Cori was also the first system to be equipped with an non-volatile RAM or NVRAM-based burst buffer.

4.1.2 HPC I/O Subsystems

The I/O subsystem of modern HPC systems is a separate entity dedicated to providing low-latency and high-bandwidth networked storage to a large number of HPC jobs. I/O subsystems are often separate from the compute nodes in order to support execution of jobs on multiple different compute clusters without requiring the movement of data between compute clusters, as well as easier maintenance since I/O nodes see higher failure rates than compute nodes due to the amount of mechanical components they possess.

In order to provide high I/O bandwidth, the parallel distributed filesystems such as Lustre (Braam and Schwan 2002) and GPFS (Schmuck and Haskin 2002) running on the I/O subsystem are distributed — files can be stripped across disks, and require that metadata is stored and accessed in order to learn where and how each file is stored. These systems typically inherit the POSIX application programming interface (API), but may also support newer APIs such as MPI-IO. POSIX originally targeted single-node computers, and the limitations of the POSIX interface can have negative effects on performance, hence the usage of newer I/O interfaces is encouraged. A large amount of HPC applications have not yet migrated to the newer technologies, and the I/O subsystem is often underutilized as a result.

4.1.3 I/O Logging Tools

Darshan (Carns et al. 2011) is a scalable HPC I/O characterization tool that collects information about I/O behavior of applications running on a system. It aims to answer both workload questions such as: what applications are running, what are their access patterns, how much I/O are they consuming, and how close are they to the theoretical peak and why, as well as systems questions such as: what is the I/O subsystem utilization and what is the filesystem health. Darshan is run by injecting itself statically at compile time or dynamically at runtime into target applications, which allows Darshan to support both new and legacy workloads. It works by replacing POSIX or MPI-IO calls with Darshan's wrapper implementations which call the original functions but also log each access and collect aggregate or per-access data. Due to the large latency of I/O operations, Darshan has a minimal impact on performance, and is enabled by default on both Theta and Cori. Darshan not only collects data on workload behavior, but also collects system-specific information such as e.g., the runtime of a job, how long I/O operations took, the distribution and variances of these times, etc. Darshan also creates some synthetic features, e.g., job I/O throughput. It has no direct way to measure I/O throughput because many different nodes may be accessing many different drives in the I/O subsystem, and thus each read or write will take a different amount of time to complete. Darshan collects both how much data each request transferred, and how much time the request took, and from these features it offers several estimates of I/O throughput. These estimates are calculated by dividing cumulative I/O volume by either the (1) cumulative time spent performing reads and writes, (2) cumulative time where files were simply held open, (3) cumulative time between a file open operation and the last file access, (4)

cumulative I/O read and write times of the *slowest process* of the job. Interestingly, this fourth I/O throughput estimate has been shown to be the most accurate in practice, as straggler nodes can have a large impact on the whole system whenever synchronization between nodes is required.

4.1.4 Logfile Sanitization and Pre-processing

Darshan collects hundreds of features about jobs, including amount of data read or written, access pattern histograms, percentage of read, write, consecutive, sequential, aligned, etc. accesses, of files that are read-only, write-only, files that are read by a single process, of which user ran the job, which command was used, etc. Not all of this data is useful for modeling the system, and a number of logs lack some features, are corrupted, are collected with an incompatible version of Darshan, etc. Significant log sanitization, feature pruning, and pre-processing is needed before these logs can be fed to any machine learning system.

In (Isakov et al. 2020) we present the following Darshan log sanitization and pre-processing pipeline:

1. **Data sanitization**: In this step, we remove jobs that are not instrumented with POSIX or have invalid values (e.g., negative values). Negative values are typically present when a job was not closed properly or when a hardware fault occurred. Similarly, we remove features that have a large number of missing values. These typically arise when the version of Darshan running on Theta has changed over the years and new features were introduced. In cases where the features were introduced recently and only a small portion of jobs is extended with them, we choose to ignore these new features.

2. **Feature pruning**: We remove common access size features and all time-sensitive features. Common access size features store the most common access sizes in bytes, i.e., describe a histogram of POSIX access sizes. Because the histogram does not have hard-coded bins but instead is calculated dynamically for each job, these values are difficult to convert to an ML-digestible format, and we choose to remove them. Time-sensitive features measure timestamps such as when files are first opened or closed last. The rationale for removing them is that Darshan uses some of these features in calculating job throughput. By leaving them in, we risk that an ML model might pick up Darshan's implementation details, instead of the wanted insight. In Section 6.4.3, we provide evidence that models trained on datasets that have only five features (four time-based features and I/O volume) significantly outperform models trained on datasets with all non-time-based features (Table 4.1). Note that other than time-sensitive features, all other features are a function of the application and input parameters; that is, these values are largely independent of the actual system the application is running on.
3. **Data normalization**: We apply feature engineering to force the values to a more manageable range. Quick investigation shows that the majority of features in our dataset have values in a wide range; for example, the total number of bytes a job has transferred can vary from tens of bytes to multiple petabytes — almost 15 orders of magnitude. This distribution is consistent across features, and without treating it in a special way, it is hard to use ML methods on such wide ranges of values. To tackle this problem, we convert the majority of the features from absolute values to values relative to some other features. We can do so because many of the features represent quantities that are portions of other

quantities. For example, Darshan records both the number of read operations and the number of consecutive and sequential reads. Therefore, because the last two features are at most equal to the number of reads, we can convert them to a percentage of total reads. Arguably, this approach cannot be universally applied; for example, the number of POSIX seek operations cannot be expressed as a ratio of a different feature. To tackle this situation, we replace the values of feature f with $log_{10}(f)$. We use log_{10} instead of ln since it is simpler to interpret and translate back to the original value. As all of the features in the dataset are positive or zero, we increment the feature by a small constant (e.g., 10^{-5}) so that the logarithm is always defined. Doing so forces the values into a more controllable range: a majority of the values lie in (-5, 12).

Data sanitization results in removing individual jobs. On Theta, from the original 661,553 collected jobs, we discard 163 jobs that contain corrupted instrumentation data, 284,464 jobs (43.0%) for which Darshan did not instrument POSIX calls, and 287,082 jobs (43.3%) that have less than 100 MiB of total I/O volume, resulting in 89,844 (13.6%) jobs. As these small (<100 MiB) jobs occupy a fraction of the total traffic (Luu et al. 2015) (small jobs transferred 974 GiB in total, while large jobs transferred 58.9 PiB), we focus on analyzing only large jobs. Data pruning and normalization results in 45 percentage features and 12 logarithmic features, not counting I/O throughput (also logarithmic). In Table 4.1 we give a brief overview of several sets of features, and we refer the reader to our open-source repository with the experiments for this work (Isakov, Currier, and Del Rosario 2022).

The main difficulty that the pipeline solves is feeding unbounded features to machine learning models that can only accept bounded values. For example, I/O volume can vary between KiB and PiB — some twelve orders of magnitude. Although

some types of models (e.g., decision trees) can accept such large variance, models that learn using gradient descent (e.g., neural networks) cannot. These features must first be scaled to acceptable ranges (e.g., $[-1, 1]$), in a way that still retains information. Directly performing min-max scaling, where a linear transformation is applied so that the smallest value before scaling becomes a 0 after, and the largest value becomes a 1 is unacceptable since most values in the range will be scaled close to 0. The pipeline applies a base-10 logarithm to such unbounded values, before applying min-max scaling. Because typically ML model users care about relative predictions (e.g., "predict I/O throughput within 5% of the true value"), instead of absolute predictions (e.g., "predict I/O throughput within 1MiB"), logarithmic scaling preserves the needed information. Table 4.1 provides a list of output features from the pre-processing pipeline.

Note that because we do not have full visibility into the system, we are unable to reconstruct the system's I/O utilization at a given timestamp. Therefore, here our analysis is geared more toward explaining internal reasons for a job's I/O throughput (e.g., by detecting good or bad I/O patterns), and less on external reasons (e.g., I/O contention).

Note that job I/O throughput is missing from Table 4.1. This is because the later sections in this chapter predict I/O throughput values of jobs running on a system based on historical data, hence this feature cannot be fed into the model.

4.2 Sensing and Monitoring Support for RISC-V

4.2.1 Binary Instrumentation

Binary instrumentation (BI) is a technique for extending a target binary with additional functionality in order to observe or modify runtime behavior. BI is typically applied statically by lifting and modifying a binary before any execution inputs are known, or dynamically by rewriting the binary using a Just-In-Time (JIT) compiler. Binary instrumentation is particularly useful for hardware development, since it offers full instruction set architecture (ISA)-level visibility into running executables. Instrumentation allows architects to collect workloads, understand performance bottlenecks, emulate proposed hardware modifications, choose optimal parameter configurations, etc. However, due to the closed nature of most ISAs, evaluating proposed novel features is largely limited to software emulation.

RISC-V is an emerging open-source ISA that offers ISA flexibility allowing users to implement custom instructions, and choice of microarchitecture (µArch) through a variety of open-source RISC-V-based platforms. Binary instrumentation tools targeting RISC-V can both speed up hardware development and verification, as well as find potential synergies through instrumentation-specific hardware extensions. The open nature of RISC-V has fostered diversity in the number and size of platforms, ranging from low-power embedded devices without virtual memory support, security and safety-critical systems, space and rad-hard applications, up to large distributed HPC systems. This platform diversity is enabled by a modular instruction set architecture, but it also introduces a number of challenges: in terms of software and ISA-level support, not all RISC-V systems have the capability to run full-fledged operating

systems (OS) necessary for certain classes of instrumentation tools, or the OS itself may be the target of instrumentation. When instrumenting 'bare-metal' binaries, the instrumentation tools may be unable to rely on virtual memory support and must statically pack both the instrumented binary and the supporting JIT compiler (Engelke, Okwieka, and Schulz 2021). Similarly, systems may not have the capability to flush instruction caches and run JIT-generated code, preventing the use of dynamic binary instrumentation (DBI). In terms of hardware support, BI tools can incur a heavy performance penalty by generating code at runtime and executing large numbers of jumps to relocated code. This penalty can be avoided on systems with deeply-pipelined, out-of-order superscalar processors with branch prediction and hardware prefetchers (Ruiz-Alvarez and Hazelwood 2008), but many RISC-V platforms may not have such features. Instead, the instrumented code may be running on FPGAs with limited cache resources, microarchitectural features, and clock speed, causing penalties to be more pronounced.

Despite the difficulties, working within RISC-V ecosystem introduces new performance and observability opportunities not available to tools targeting proprietary ISAs such as x86 or ARM. With an open ISA, instrumentation tools and hardware can cooperate through ISA extensions to e.g., offer hardware-based instrumentation triggers and lower JIT performance penalty, avoid I-cache thrashing through instrumentation-specific microcode, minimize the microarchitectural pollution from executing instrumentation code, as well as expose certain microarchitectural information to verified tools.

In this work we present the **Trireme RISC-V Instrumentation and Analysis Library (TRAIL)**, a static binary instrumentation tool targeting RISC-V program analysis, rapid hardware prototyping, and microarchitectural data collection on real

hardware. We analyze architectural decisions from existing instrumentation tools and discuss how past design decisions apply to RISC-V. We propose a static binary instrumentation framework and a set of hardware modification that enables efficiently exposing microarchitectural information to computer architects.

4.2.2 Instrumentation in Open HW Ecosystems

Instrumentation tools make implicit assumptions about their operational domain, e.g., whether the user prioritizes precise instruction-level instrumentation or seeks efficient function-level logging, whether the user wants to modify the behavior of the program or just observe execution, whether the target binary is dynamically or statically linked (and whether instrumenting dynamically linked libraries is of interest to the user), whether skipping or incorrectly instrumenting a portion of the binary is acceptable, whether binaries are malicious and are actively attempting to subvert instrumentation, etc.

Developing a binary instrumentation tool targeting RISC-V offers an opportunity to reevaluate these assumptions in the context of an extensible ISA and the availability of open-source RISC-V cores. Limitations of previous tools may not necessarily apply to RISC-V, as these tools have had to work around the constraints of closed ISAs and opaque, proprietary microarchitectures. At the same time, the less entrenched RISC-V software infrastructure and the relative nascence of RISC-V hardware platforms pose both novel challenges and possibilities.

While RISC-V is receiving significant adoption in research areas ranging from IoT to HPC, currently the majority of deployed hardware is in the embedded domain. These devices may not have virtual memory or multithreading, and are running

bare-metal or minimal operating systems. Binary instrumentation tools cannot rely on the operating system loader to load binaries into memory, preventing instrumentation through 'bootstrapping' (Luk et al. 2005). The user base for such tools is also different: while e.g., malware research contributes to a significant part of BI user base for more dominant ISAs, no RISC-V-based malware currently exists. On the other hand, computer architects and embedded programmers form a sizeable cohort, and may need better tooling for analyzing low-level bugs in scenarios where they may not trust hardware correctness as much as e.g., ARM users do. These concerns, along with our need for hardware introspection of RISC-V workloads, lead us to develop the Trireme RISC-V Analysis and Instrumentation Library (TRAIL).

TRAIL aims to enable users to: (1) gather ISA-level insight into application state, allowing users to better optimize code, debug memory and control flow problems, perform concolic execution (Shoshitaishvili et al. 2016), and analyze malware, (2) modify program behavior, allowing users to fuzz binaries, prototype hardware through instruction emulation, and perform taint analysis, and (3) collect privileged microarchitectural data with minimal hardware modification, allowing users to collect µArch traces with minimal microarchitectural pollution, understand design bottlenecks, and explore and optimize hardware configurations.

What separates TRAIL from existing binary instrumentation libraries is a focus on speeding up microarchitectural data collection through execution *on real hardware*, mainly targeting reconfigurable logic. By occupying a middle-ground between ISA-level binary instrumentation and hardware signal taps, TRAIL aims to provide the quick turnaround of software development combined with the full system insight of microarchitectural probes. As software compilation is typically many orders of magnitude faster than hardware synthesis, by building generic µArch exfiltration data

diodes and instrumentation and analysis ISA extensions to configure these diodes, TRAIL aims to remove the need for modifying hardware signal taps where possible, speeding up the hardware design and verification cycle.

4.2.3 TRAIL Static Binary Instrumentation

Dynamic binary instrumentation (DBI) is generally preferred over static instrumentation due to its instrumentation correctness and coverage guarantees, however, dynamically-instrumented binaries have higher hardware requirements to run. DBI may be difficult to run bare-metal and requires an operating system to fork and load the target binary, the system must have enough memory to pack a JIT compiler, must support instruction cache (I-cache) coherence or fences, and must have a large-enough I-cache to prevent cache thrashing by the JIT. Some of these requirements are prohibitive during the hardware development lifecycle, e.g., when targeting area-constrained FPGAs. TRAIL instead is a static binary instrumentation (SBI) tool, building deployable binaries with no new dependencies. TRIAL Instrumentation proceeds in three phases: (i) *control flow graph (CFG) recovery*, (ii) *insertion point detection*, and (iii) *code injection*.

During the CFG recovery phase, TRAIL traverses the binary and identifies basic blocks and functions. In RISC-V, the target of branches (`B**`) and direct jumps (`JAL`) instructions can be statically determined, while for indirect jumps (`JALR`) the target address is stored in a register and is not directly available at compile time. Indirect jumps are commonly used for return addresses, case statements, jump tables, virtual functions, position independent code, etc. Statically resolving the space of possible indirect jump targets to extract the control flow graph is shown to be undecidable

in the general case (Horspool and Marovac 1980). Fortunately, most programs are not obfuscated and actively avoiding detection, and good results can be achieved through symbolic analysis and heuristics. PEBIL (Laurenzano et al. 2010) uses peephole exploration to identify address offsets and correctly disassemble 99.0% of bytes in SPEC CPU2000 integer benchmarks. TRAIL uses the same approach as PEBIL, accepting that a small portion of code will not be instrumented without further interaction from the user. For identifying functions, TRAIL currently requires unstripped binaries and uses symbol tables to determine function starts and ends. During the insertion point detection phase, TRAIL executes a number of instruction-level, basic-block-level and function-level user-defined functions which may potentially inject code. With the symbol table available and the CFI correctly extracted, all three steps are direct. During the code injection phase, TRAIL receives a list of instruction addresses at which specific instrumentation routines need to be invoked. TRAIL replaces the instructions at those addresses with direct jumps to instrumentation routines. Since RISC-V instructions have a fixed length of 4 bytes (TRAIL currently does not support compressed instructions), the replaced instructions can be simply moved to the instrumentation routine without breaking functionality. This feature of RISC-V is important, since it removes the need to relocate whole blocks or functions (which could potentially double code size), and also removes the need to modify relative addressing in the program. If TRAIL skips instrumenting e.g., a basic block connected by an indirect jump, the instrumentation will not run, but original functionality is preserved. Finally, TRAIL populates the instrumentation routines with instrumentation snippets.

Since binary instrumentation tools inject code into a target binary, instruction pointers (IP) may be modified (either by relocating functions or by inserting new

instructions and shifting the rest of the binary) and the correspondence between the original binary and any collected IPs may be lost. Tools such as Intel Pin (Luk et al. 2005) and DynamoRIO (Bruening and Amarasinghe 2004) put significant effort into *address space transparency*, where both the application and the observer see the original IPs. TRAIL achieves transparency by replacing only a single instruction with a jump to an instrumentation routine. While this approach technically does relocate a single instruction, TRAIL takes special precautions in case of program counter-sensitive instructions such as `AUIPC`, `JAL`, and branches, making sure that these instructions observe the original program counter. Additionally, TRAIL identifies free registers to use as the return address so that only one instruction needs to be replaced, since replacing multiple instructions is problematic in cases where e.g., multiple branch instructions are issued one after another. TRAIL uses direct jumps to jump to instrumentation routines, which can be a problem on very large binaries due to `JAL`'s ± 1 MiB range.

4.2.4 HW / SW Co-Design for Accurate and Extensible Microarchitectural Instrumentation

While TRAIL is designed to support RISC-V devices implementing no ISA extensions (bare RV32-I or RV64-I), by leveraging the open nature of RISC-V, there exist significant opportunities to (i) decrease the performance impact of instrumentation, (ii) allow introduction of new hardware performance counters, (iii) expose instrumentation code to possibly precise, microarchitectural events and eliminate the need for more coarse-grained counters, (iv) remove microarchitectural pollution caused by instrumentation, (v) increase ISA and microarchitecture-level data exfiltration

bandwidth, (vi) and prevent software from learning it is instrumented. We will present two proposals for extending RISC-V processors, sorted by the invasiveness of hardware modification:

TRAIL Extensible Nodes: a specification and hardware prototype exposing fine-grained hardware events to software, shown in the top left corner of Figure 4.2.

TRAIL Pair Nodes: a specification and a prototype enabling separation of a target binary from instrumentation code on separate, loosely-coupled nodes, with a 'hero' node executing the binary and sending µArch events to a 'sidekick' node which executes instrumentation code and processes events (bottom row of Figure 4.2).

4.2.4.1 Exposing µArch Events, (Not Counters!), to Software

ISAs typically do not permit software to learn information about the behavior of the underlying microarchitecture. While knowing whether a load caused a cache miss or how often a specific branch mispredicts may be useful both at design time and runtime to optimize hardware or software, and though such information can be gleaned through e.g., timing side-channels, most ISAs do not expose this information. Hardware Performance Counters (HPCs) and Performance Monitoring Units (PMUs) are hardware features that measure microarchitectural events, aggregate them, and possibly take actions on certain conditions (e.g., raise an interrupt).

Unfortunately, HPCs suffer from a number of issues that limit their usefulness: (i) HPCs are often imprecise at the instruction or basic block level, returning incorrect values in the presence of e.g., a large numbers of interrupts (Weaver and McKee 2008); (ii) accurately counting the numbers of events on Out-of-Order (OoO) processors is is difficult to correctly implement, since some types of events (e.g., TLB misses) may be

raised significantly after an instruction is dispatched, and may be attributed to the wrong instructions or sections of code (Weaver, Terpstra, and Moore 2013); (iii) the functionality specific HPCs is often ill-defined. For example, L2 caches may or may not count L1 prefetch misses as L2 accesses (McCalpin 2013); (iv) HPC behavior can also vary between different versions of a microarchitecture; (v) counters may be privileged and require OS interaction to access, reducing performance and polluting data; and (vi) while processors expose hundreds of different events to HPCs, typically only a handful of counters are implemented due to area hardware constraints. When performance analysis tools need to collect more HPCs than a processor possesses, the tools may alternate between collecting disjunct sets of counters and extrapolate total values, further adding to their error.

There exist no fundamental reasons why most HPCs cannot be made trustworthy (Demme and Sethumadhavan 2011; Nowak et al. 2015). While expensive, counters can be made aware of interrupts, and their updates can be committed in order, making HPC read operations sensitive to instruction ordering. Having precise HPCs opens an interesting use case: by reading a counter before and after an instruction or basic block, software can learn whether e.g., a branch was successfully predicted or what was the latency distribution of that operation. Binary instrumentation has additional synergy with precise HPC updates, since BI tools can collect information above the ISA, and HPCs can collect µArch information.

To enable users to precisely observe diverse sets of events, we extend the Trireme Out-of-Order core (Ehret et al. 2022) and Network-on-Chip (NoC) node into a *TRAIL Extensible Node* (top row of Figure 4.2). The extensible node contains an HPC subsystem which taps into a parameterizable number of wires and registers belonging to the node microarchitecture. The subsystem is designed to be extensible, enabling

users to change which and how many signals should connect to the HPCs and allowing users trade-off hardware area for higher frequency. For example, the HPC subsystem in the prototype TRAIL extensible node connects to a set of wires such as the 1-bit branch predictor access and mispredict signals and 64-bit target addresses, 1-bit L2 cache access, load/store, prefetch, and miss signals and 64-bit cache access addresses, etc. While these events directly connect to existing µArch signals, pseudo-events [7] can also be implemented. Pseudo-events may be useful when searching for rare bugs, where a very specific set of conditions needs to be satisfied to start collecting data, or when detecting malware, as combinations of HPCs have been shown to be more discriminative of e.g., side-channel attacks (Mirbagher-Ajorpaz et al. 2020).

The HPC subsystem in the TRAIL extensible node connects these tapped events to a generic and parameterized number of hardware performance counters. The events TRAIL taps into are not necessarily single-bit and rare one-hot events. TRAIL takes a broader view of 'counters' and replaces them with 'aggregators'. Aggregators can count events, sum event values (e.g., sum ROB occupancy to calculate average utilization), or perform simple logic operations at the cost of increased die area for arithmetic operations and storage. A TRAIL HPC consists of four components: a multiplexer, an aggregator, a value register, and a configuration register. The multiplexer selects one of the connected events based on configured event type and process ID[8]. The aggregator is an n-bit ($n \in \{32, 64, 128\}$) ALU implementing addition, minimum and maximum functions, and bitwise AND, OR, and XOR operations. The value register

[7] Pseudo-events are events not present in the microarchitecture directly, but are a product of logic performed on collected events within the HPC subsystem, and only serve to improve µArch observability.

[8] How events are connected to any HPC is left to the implementation in order to allow area-flexibility trade-offs.

has n bits and stores the current HPC value which is hardwired as one of the inputs to the aggregator (the other being the event). The configuration register has 32 bits and controls (i) the event which the multiplexer should select, (ii) the PID of the process being monitored (-1 for the whole system), (iii) whether reading the counter resets the HPC value, (iv) the aggregation strategy (e.g., count, min, max, any, all, odd, etc.)

We choose to memory-map HPCs and not have the counters accessible through the RISC-V CSR registers since the CSR space is limited and TRAIL aims to allow users to create and expose large numbers of new counters. The value and configuration registers are mapped to two separate pages, allowing the operating system to freely map the HPC values to user space without risk that programs will gain unauthorized access to HPCs. During context switches, the OS stores the value and configuration registers, updates the configuration for the new process (e.g., by disabling HPCs), and restores them once a process is resumed.

To illustrate how the TRAIL extensible node is used, in Listing 4.1 we provide example assembly snippet in which a program loops over an array and calls one of three functions depending on array values. The user optimizing the loop may want to know more than the aggregate branch miss count after a number of iterations, seeking to get an accurate branch prediction time series. By either modifying the source files or instrumenting the binary to insert the lines with a highlighted in green, the user can accurately measure the number of branch misses that appeared on lines 18-20 *for every loop iteration.*

While the extensible node can provide instruction and basic block-level microarchitectural insight, the increased amount of collected data can be as detrimental as it is beneficial. In the snippet above, 50% of the code comes from the instrumentation

routines, potentially polluting the µArch by e.g., increasing the distance between branches which may improve the prediction accuracy of some out-of-order cores. Ideally, instrumentation code would have *no* microarchitectural effects, removing pollution and preventing instrumentation detection.

```
1  setup_HPCs: # setup the configuration register
2    lui  a1, <HPC_1 value address>
3    lui  a2, <data exfiltration array address>
4    lui  t1, <HPC_1 configuration address>
5    lui  t2, <branch mispredict ID, DNE on read>
6    sw   t2, 0(t1)  # enable HPC with configuration
7  parse_array: # setup array start and end
8    lui  t1, <array address>
9    lui  t2, <array size>
10 loop: # iterate loop
11   lw   t3, 0(t1)
12   addi t1, t1, 4
13 case_statement: # branch depending on array[t1]
14   sw   zero, 0(a1) # delete any loop branch miss
15   blt  t3, zero, <function_1>
16   beq  t3, zero, <function_2>
17   bgt  t3, zero, <function_3>
18   lw   t4, 0(a1) # read # of branch mispredicts
19   sw   t4, 0(a2) # store for later processing
20   addi a2, a2, 4
21   blt  t1, t2, loop
22 end: ...
```

Listing 4.1: Example RISC-V assembly instrumented to collect and store the number of branch prediction misses *inside the for loop* for each loop iteration.

4.2.4.2 Removing µArch Pollution with Sidekick Cores

Instrumentation tools typically support both observing and modifying a running binary. Limiting instrumentation tools to just observation restricts the direction of the information flow to just one direction, from binary execution to instrumentation routines. The unidirectional information flow can help decouple the observer and the binary in both time and space, e.g., by offloading (a part of) the instrumentation code, program data, and µArch events to separate processing units. By offloading and executing all instrumentation code on a separate node, the node executing the binary can maintain identical microarchitectural state transitions as when executing the original (pre-instrumentation) binary. We will call the node executing the binary the 'hero' node, and the node executing instrumentation code the 'sidekick' node. The goal of the hero node is to execute functionally the same instrumented code that the TRAIL Extensible node executes, but offload the instrumentation instructions to the sidekick node, removing performance penalties and pollution caused by binary instrumentation. For this offloading to work, the sidekick node needs to be executing code that has *the same control flow graph* as the hero node, except that the basic blocks in the sidekick CFG are populated with only the instrumentation code. The hero node must communicate to the sidekick node (i) ISA-level information requested by instrumentation instructions, e.g., register contents and PCs, (ii) a subset of hero node hardware events requested by the instrumentation, and (iii) necessary dynamic control flow information (branch resolution, branch targets, etc.) from the binary running on the hero node, so that the sidekick node may traverse the same path through the CFG. To follow the same path through the CFG, the sidekick node decides which branches to take based on the hero's path, not its own. With these

modifications, by taking the code from Listing 4.1 and extracting the lines highlighted in green as well as any branches and jumps into a separate binary *with the same control flow*, two programs can run in parallel while traversing the same path through a CFG, even if the sidekick is trailing behind the hero.

We implement such a flow with TRAIL pair nodes consisting of a hero and a sidekick node as shown in Figure 4.2. By restricting instrumentation code to purely observation (i.e., no behavior modification), the sidekick node can observe the hero node with some latency and not require tight integration which might limit system frequencies. The hero and sidekick nodes are physically separate and operate in different clock domains and with separate caches, except for the lower portion of the memory hierarchy (L3, NoC, memory controllers, DRAM). All communication between the nodes flows over a number of FIFOs of parameterizable width and depth. The FIFO is paired with a fullness flag which is raised by the hero node when the FIFO is full and the hero node must start dropping hardware events. It is mapped to a number of addresses in memory and popped when the sidekick node reads the memory address, and the sidekick is able to collect HPC events in the same manner as the extensible core would We choose to communicate HPC updates over FIFOs instead of e.g., making updates cache-coherent and visible from other cores due to the potentially very large communication bandwidth HPCs may require, which would affect memory bandwidth and pollute microarchitecture. Additionally, deep FIFOs are preferable since they provide slack and allow for physically separating nodes without the FIFOs being on any critical path.

Table 4.1: Condensed feature set and feature count

Darshan features (present on both Theta and Cori)	Count
log_{10} of the job I/O throughput	1
log_{10} of the total number of {processes, files, accesses, bytes}	4
log_{10} of the number of POSIX {`open`, `seek`, `stat`, `mmap`, `fsync`, `mode`} calls	6
log_{10} of {memory, file} alignment in bytes	2
% of all accesses that are {reads, writes}	2
% of all {reads, writes} that are {consecutive, sequential}	4
% of all accesses that switch between reading and writing	1
% of {read, write} accesses of size in ranges (0B, 100B], (100B, 1KiB], ..., (100MiB, 1GiB], (1GiB+)	20
% of non-aligned {file, memory} accesses	2
% of all bytes that are {read, written}	2
% of {shared, unique, read-only, read-write, write-only} files	5
% of bytes read/written from {shared, unique, read-only, read-write, write-only} files	5
Lustre features (Cori only)	**Count**
log_{10} of file system {byte, inode} fullness {min, mean, max, std}	8
log_{10} of metadata target {closes, getattrs, getxattrs, links, mkdirs, mknods, opens, renames, rmdirs, setattrs, statfss, unlinks} operation mean	12
% of data server {CPU, memory} {min, mean, max, std}	8
% of data target bytes {read, written} {min, mean, max, std}	8
% of metadata server CPU usage {min, mean, max, std}	1
Cobalt features (Theta only)	**Count**
log_{10} of {core, node} count	2
log_{10} of job runtime	1
% of job {start, end} time relative to total system time range	2

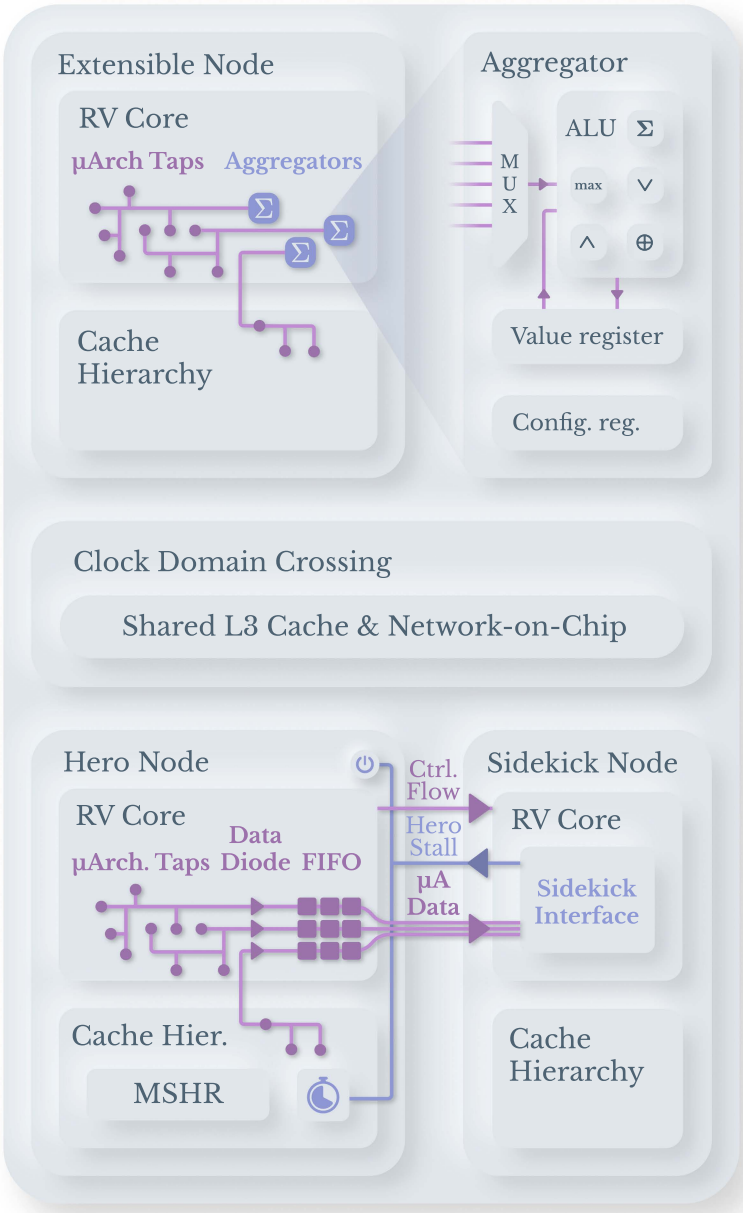


Figure 4.2: A Network-on-Chip (NoC) system with a base RISC-V node, an TRAIL Extensible node, and a Hero-Sidekick pair of nodes. Each node and the NoC sit on a separate clock domain.

Chapter 5

DESIGN-TIME OPTIMIZATION AND ADAPTATION

Adaptive systems may be considered synonymous with systems that modify their own behavior after manufacturing or deployment. However, the necessary mechanisms for a system to be considered as adaptive — sensing, decision-making, and adaptation — can be observed in organizations that build (non necessarily adaptive) systems as well. In this section I will study optimization and adaptation performed at system design-time.

5.1 Design-Time Optimization as Part of the Adaptation Loop

Fundamentally, adaptive systems attempt to solve the mismatch between design-time assumptions about the distribution of workloads or working conditions, and runtime-collected information about specific applications and the system environment. Instead of making system configuration decisions at design-time, scheduling-time and run-time adaptive systems invest resources (additional hardware area, power, and latency) to gain flexibility on which configuration to take.

By taking a broader view of 'systems' and focusing on e.g., a line of products designed by the same organization, adaptation mechanisms present in domains such as computer architecture or high-performance computing can be studied. Here, the 'system' is not be just the physical hardware being designed and deployed, but instead includes the system design team as well. The system designers act as actuators, modifying future generations of the deployed devices, the data collected from end

users acts as sensors, and the decision-making is performed by humans as well as machines.

In this chapter I analyze such human-in-the-loop design-time adaptive systems, introduce a tool for automated design space exploration, and show the limits of design-time adaptation and the necessity of having scheduling-time insight into workloads.

5.1.1 Design-Time Adaptation Goals

General-purpose computing systems may not know end-user requirements, and may need to support a wide range of goals such as throughput, latency, energy-efficiency, security, or cost. Optimizing for multiple goals without a method for unifying these goals (a unifying method may be e.g., by taking a weighted sum of all goals) is a multi-objective optimization problem, and is typically 'solved' not by offering a single point solution, but by presenting a range of solutions and allowing other decision-makers (e.g., end users) to select a solution that fits their needs. A number of parallels can be drawn between (1) scheduling-time or run-time adaptive systems and (2) design teams developing multiple generations of a general-purpose system:

1. **Changing goals**: system designers may not know e.g., area, power, latency, and throughput trade-offs of their designs until late in the system development cycle. They must therefore develop and maintain a parameterized design that works for many different goal configurations.
2. **Unclear workloads**: designers may not precisely know the distribution of workloads their general-purpose system targets, and their systems must remain flexible and reconfigurable when new information is provided.
3. **Provisioning actuators**: adaptive systems may spend some resources such as

area or power to gain runtime flexibility. Similarly, teams may spend development effort building and maintaining parameterized, competing, or optional hardware designs in order to have flexibility once the deployment domain is known. The cost of this parameterizability is not run-time performance or power, but development time.

4. **Iterative decision-making** : both adaptive systems and design teams are often limited by compute power and the vast search space of possible configurations. Due to limited visibility into the target domain, neither can claim to reach optimal system configurations. Instead, both system design teams and adaptive systems iteratively evaluate best-guess configurations, collect data on goal satisfaction, and repeat the process. For adaptive systems, this process appears as scheduling-time or run-time adaptation. For system design teams, this process involves deploying systems, collecting workload and environment data, updating their model of workload distributions, and repeating the process.

5.1.2 Actuators in Design-Time Optimization

The difference between static parameterized systems and adaptive systems is that static systems can change configurations during the design process (before fabrication) to better fit some target workloads, while adaptive systems may change configurations after fabrication as well. Viewed at longer timescales, a system is not just an individual fabricated and deployed device, but can also be a generation of physical systems being deployed and replaced with newer versions. In that view, a system represents the logical design, not the physical component, and that logical design is not static, but instead changes over time, although over longer timescales. For example, Figure 1.3

illustrates such a system that adapts at three timescales: in milliseconds at runtime, in minutes or hours between workloads, and over months or years with new software or hardware deployments.

5.1.3 Sensors in Design-Time Optimization

Adaptive systems may build sensors that measure (i) workload properties (e.g., average access latencies, number of branches or memory accesses, instruction dependency distribution, etc.) (ii) system properties (e.g., number branch mispredictions and cache misses), or (iii) environmental properties (e.g., channel jitter, shared resource contention, etc.). Similarly, system design teams may build sensors to learn more about the deployment domain. For example, sets of benchmarks that teams evaluate their designs on may not be representative of workloads a system will encounter in the field. Teams may therefore collect application binaries or instruction traces after deploying a generation of their systems, so that future iterations of the system may benefit from increased domain insight. Teams may also build hardware modifications such as Hardware Performance Counters (HPCs) and have deployed systems communicate back their observations.

5.2 Capabilities and Limitations of Design-Time System Optimization

I distinguish between two types of system design processes aimed at (i) one-off designs, and (ii) families of designs. A one-off design is a system that after deployment does not receive further improvements, any insight gained in the field about the target domain is unused, and no next-generation systems are developed following

system completion. A design within a chain or a family of designs on the other hand is planned to receive updates, improvements, and deployments of next-generation systems. They family may be designed with parameterizability in mind, and built with data collection subsystems that send deployment-time domain information back to the design team.

One-off design-time systems should not be considered adaptive at any timescale, but families of designs may be. Here I investigate some of the difficulties these systems face.

5.2.1 Limited Design-Time Domain Visibility

Given a non-adaptive parameterized system design, system designers must select a specific configuration of parameters before system manufacturing or deployment. Given perfect knowledge of the target domain, designers may execute a parameter search and find an optimal parameter configuration. However, this approach is typically not feasible for several reasons:

- Designers are unlikely to have perfect knowledge of the target domain. For example, the set of all workloads that will be executed on a specific CPU being developed is too great to enumerate. Additionally, this set must be *weighted*, since not all workloads will be run the same amount of times, nor do they have the same priority.
- The target domain is not stationary: workloads and system conditions change over time and in unpredictable ways.
- Even with perfect knowledge of the target domain, the design parameter space may be too large to exhaustively test and optimality cannot be guaranteed.

- Additionally, perfect knowledge of workloads may be invalidated by unknown run-time conditions.

5.2.2 Need for Better Design Processes

While scheduling-time and run-time adaptive systems solve many of the problems listed above, frameworks that treat system design as a continuous process spanning multiple generations of deployments may offer similar benefits. The framework I propose aims to enable designers to formalize, collect, quantify, and adapt to: (i) changing end-user goals, (ii) the unknown and shifting distribution of target domain workloads, (iii) yet to be determined performance and technological barriers of individual systems, (iv) the unknown or changing space of environmental effects affecting the system. In such a framework, deploying a system is not treated as just a goal, but also as a data-collection experiment used to better understand the target domain.

5.3 Holistic and Automated Design Space Exploration

Similarly to how adaptive systems use runtime decision-making to find beneficial configurations, optimizing and adapting systems at design time can be automated. While some human interaction is necessary for e.g., building new workload and system sensors, analyzing system bottlenecks, and expanding the configuration space, once the system configuration parameter space is defined, optimizing system configurations may be best left to machines.

In this section I present `hoppi`, a holistic optimal pipeline system design space ex-

ploration tool. `hoppi` automatically optimizes systems within some design constraints, and aims to help dataflow and pipeline system designers to build optimal systems and answer questions such as: (i) what is the expected performance of a specific system configuration? (ii) What are the system bottlenecks given this configuration? (iii) What are the trade-offs between the competing system objectives? (iv) What is the system Pareto-frontier given this space of configurations? (v) How sensitive is the system to configuration parameters?

5.3.1 `hoppi` Motivation and Goals

`hoppi` was primarily developed to help teams of system designers working on independent system submodules align their goals and optimize for the same global objective. The main goal of `hoppi` to provide a shared infrastructure for rapid system refinement and communication of priorities. Here, different teams express their submodules as white, black or grey box models, using analytical equations (white box), machine learning models (grey box), or software simulations (black box) that tie into `hoppi`'s interface, allowing `hoppi` to evaluate the success of the full system. `hoppi` then communicates back to individual teams which objectives and constraints are affecting their submodules, and each team may receive different feedback.

The main goal of `hoppi` is to either provide an exhaustive Pareto-frontier for a given system formulation, or inform the user that the current system simulation cannot be exhaustively searched. Exhaustive exploration is important as system designers may want to know that no hidden regions of (possibly superior) solutions exist, or at least be informed that further model refinement and more compute power is needed to find them.

5.3.2 Multi-Objective Optimization

When designing systems, designers typically have multiple competing objectives such as throughput, latency, energy efficiency, cost, security, error resilience, etc. The system may also have a number of constraints, e.g., latency, maximum power draw, cost, or mean time to failure. These objectives and constraints are said to be competing since for the majority of top-performing designs, improving one objective necessarily negatively impacts some other objective.

When comparing two designs d_1 and d_2, each of which is attempting to maximize objectives $o_1(d_i)$, $o_2(d_i)$, ..., $o_n(d_i)$, d_1 is said to be dominating d_2 if

$$\forall x \in \{1, 2, ..., n\},\ o_x(d_1) > o_x(d_2) \tag{5.1}$$

Here, design d_1 is objectively better than d_2 since it achieves higher satisfaction of all of the n objectives. A design d_1 is said to be Pareto-optimal if there exists no other design d_2 that dominates d_1. In other words, given some Pareto optimal design d_1, any change made on d_1 that improves one objective must hurt at least one other objective.

Optimizing around multiple objectives and constraints complicates both design optimization and system interpretation. Since the majority of optimization methods (e.g., hill climbing, simulated annealing, genetic algorithms, neural networks, etc.) work on a single objective, optimizing multi-objective systems requires either converting the n objectives into a single objective, or restricting the choice of optimization method to only those which support multiple objectives. A naive approach typically weighs different objectives using some user-selected weights $c_1, c_2, ..., c_n$ as:

$$o(d) = \sum_{i=1}^{n} c_i o_i(d) \tag{5.2}$$

This approach is problematic for three reasons: (i) the weights are selected by the designer, but the designer may not know the acceptable trade-offs between different weights until later in the design process, (ii) the objectives may have highly nonlinear interactions, and small changes in weights may have large and discontinuous impacts on the final design, and (iii) optimization methods may be sensitive to weights, and performing a grid search over combinations of weights within a certain range may not find all Pareto optimal solutions.

When a user does not have a specific objective trade-off in mind, they may seek to collect a population of Pareto optimal designs which form a Pareto curve. Given n objectives, a Pareto curve is an $(n-1)$-dimensional manifold consisting of Pareto optimal solutions. By visualizing the Pareto curve, a user that does not know the final objective trade-offs can nonetheless make meaningful progress in system design. By observing any nonlinearities in the curve, the user may search for 'knees' — sharp bends in the manifold typically indicative of good middle ground between competing objectives. However, for $n > 4$, the Pareto curve is four- or higher-dimensional, making it difficult to visualize and interpret. When projected into two or three dimensions, these curves typically form dense 2D or 3D regions that may appear opaque due to high sampling density. At that point, the usability of multi-objective optimization quickly drops off, leading users to commonly hand select a number of objective trade-offs, lowering n to an acceptable level. These hard-coded objective trade-offs must later be evaluated and the results possibly discarded if the trade-offs are considered suboptimal once a better objective weighing is known.

5.3.3 Problem Definition

System design space exploration can be formulated as a multi-objective, mixed-integer and nonlinear optimization problem (MINLP) of the form:

$$\begin{aligned} \min_{d} o_i(d),\ i \in \{1, 2, ..., n\},\ d \in \mathbb{D} \\ g_j(d) \geq 0,\ j \in \{1, 2, ..., p\} \\ c_k(d) = 0,\ k \in \{1, 2, ..., q\} \end{aligned} \tag{5.3}$$

Here, n is the number of objectives, p is the number of inequality constraints, and q is the number of equality constraints. $\mathbb{D}$ is the space of system configurations, and in general is mixed-integer and non-convex.

5.3.4 Proposed Solution

Users define their system and subsystem models in `hoppi` either analytically (white box models), through machine learning modules (grey box models), or by encapsulating simulations as black boxes. Through API-agnostic integration with third-party software such as system emulators and simulators, users can gain accurate data about those systems and avoid the difficulty of modeling or porting those systems to `hoppi`. By connecting external software and expressing it in terms of input metrics and constraints and output objectives, these third-party tools can be probed by `hoppi` during the optimization process.

Through hierarchical system modeling where system models can contain white box, grey box or black box subsystems, system models can be modularized and used as parts of other systems, enabling modular refinement and allowing system designers to quickly gain high-level insight into the general range of feasible solutions. Before

`hoppi` can evaluate the whole model, the systems models are translated into sets of equations, constraints, and objectives specific to a chosen back-end solver that `hoppi` will use. Supporting multiple solvers allows `hoppi` to cover linear (LP), quadratic (QP), convex (CP), nonlinear (NP), mixed-integer (MIP), and mixed-integer nonlinear (MINLP) programming solvers.

`hoppi` supports several features that enhance user productivity and debugging models. The systems models defined in the Python-based domain specific language (DSL) can be directly diagrammed and plotted, and the interactive visualization allows users to collapse subsystems in order to hide model complexity. Once a user completes a model, the model can be type-checked for convexity, allowing the user to learn which expressions are non-convex and cannot be efficiently solved. Additionally, `hoppi` offers optional dimensionality analysis, where users can assign units to expressions (e.g., megabytes per second), and `hoppi` can confirm that all expressions have matching dimensions. Finally, once `hoppi` has found the Pareto front, the front is visualized using an interactive web-based parallel coordinates plot across all metrics, constraints and objectives, with controllable ranges and optional 2-dimensional scatter plots.

5.3.5 System Model Definition

`hoppi` introduces a Python-based Domain-Specific Language (DSL) with the goal of simplifying defining system models to the end users. The DSL defines 5 primitives:

- Metrics: metrics are user-controllable (or when `hoppi` is optimizing the system, `hoppi`-controllable) system parameters or 'knobs' that control system behavior. `hoppi` explores how metrics affect constraints and objectives, and searches for

Pareto-optimal configurations of metrics. Metrics are represented symbolically, and do not have values until the solver finds them.

- Expressions: `hoppi` redefines basic operators such as multiplication, exponentiation, etc. to perform symbolic manipulation of the operands, which can be metrics, expressions, or constants. Expressions are symbolic objects that represent an operation over one or two operands, and are unresolved until all leaf nodes (metrics and constants) are resolved, and the solver has found a configuration of values for them.
- Objectives: objectives extend expressions and set optimization goals for the solver. The solver is tasked with minimizing or maximizing all objectives that are defined in a top-level system or its subsystems.
- Constraints: constraints extend expressions with equality and inequality operators. All system constraints must be satisfied for a solution to be considered feasible.
- System: systems hold metrics, objectives or constraints, as well as other subsystems. They connect metrics between systems into a cohesive model that can be fed into mathematical solvers.

In Figure 5.1 I provide an example system definition that describes a hierarchical system with nonlinear objectives and constraints, explores the design space and arrives at a Pareto optimal curve.

5.3.6 Design Methodology

In `hoppi`, users define a hierarchy of systems and subsystems, each governed by some internal equations or simulations, and the subsystems interact through strictly

Figure 5.1: Example model declaration defining a system with two subsystems.

defined interfaces, i.e., lists of input and output metrics. `hoppi` encourages modular refinement of a hierarchical design, where users first define a top-level system model with simple approximations for subsystems. For example, In Figure 5.2 I show an example hierarchical system diagram `hoppi` generated from the model shown in Figure 5.1. Here, the circles are constants and expressions, diamonds are objectives, while green and red boxes are metrics and constraints, respectively. `hoppi` then

evaluates this model and learns which subsystems have the largest impact on global behavior, and allows the user to later refine subsystems through more accurate analytical models or more complex simulations.

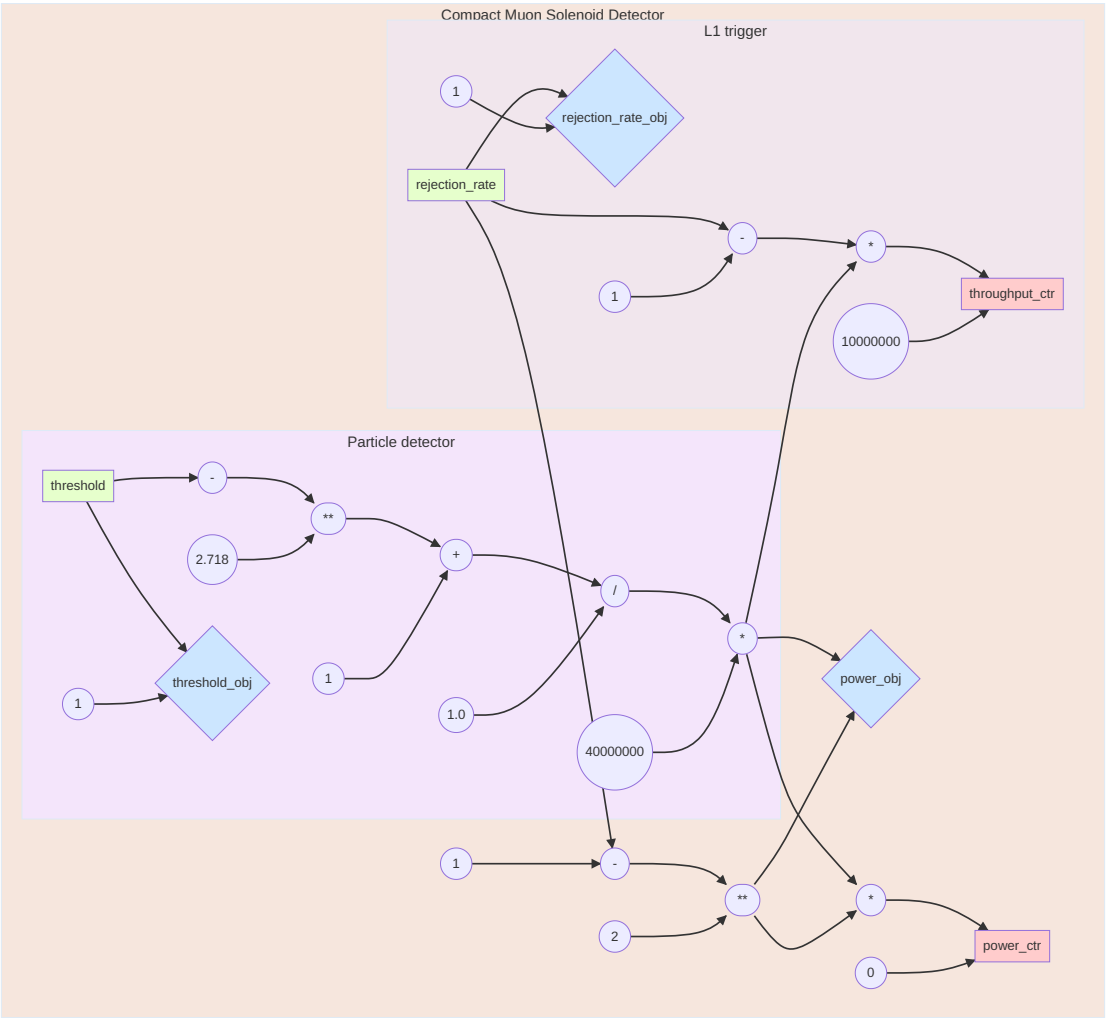


Figure 5.2: Generated diagram from the system definition in Figure 5.1.

User studies using the tool have revealed that defining large, multi-team systems may seem daunting to users, so `hoppi` proposes several guidelines:

- Think in terms of black boxes: when subsystems are logically or physically separate from the other systems, they may be expressed as black boxes with

a small number of input and output metrics, objectives, or constraints. If system internals can be hidden behind such interfaces, this simplifies both model definition, and solving the model.

- Encapsulate complexity: `hoppi` cannot represent systems that depend on some temporal component for simulation. Instead, `hoppi` chooses to integrate and encapsulate such time-sensitive simulations behind black box models. All that `hoppi` requires from time-sensitive (sub)systems is that they export a list of feasible Pareto optimal configurations from which `hoppi` can select configurations most useful to the global system.
- Focus on data flows: `hoppi` treats all (sub)systems as acyclical message-passing systems where arbitrary latency in terms of buffers or "FIFOs" can be added between subsystems. As the subsystem connectivity graph is acyclical, `hoppi` can independently optimize subsystems without risk of subsystems interacting or exchanging information through any channel except the defined interfaces.

5.3.7 Multi-Objective Optimization and Visualization

When `hoppi` solves a system, it produces not just one solution, but a front of Pareto-optimal solutions. For a solution to be Pareto-optimal, it must satisfy the property that further optimizing the value of one of n objectives will necessarily hurt at least one other objective. Pareto curves are useful when a user knows the set of objectives a system has, but cannot (yet) decide the acceptable trade-offs between those objectives. The curve offers the user a 'menu' of solutions which are guaranteed to be (Pareto) optimal, i.e., whatever the final user trade-off is, the optimal solution is guaranteed to lie on the Pareto curve. In Figure 5.3 I show a parallel coordinate

plot (top) over model metrics, objectives and constraints, and a scatter plot (bottom) displaying a Pareto curve obtained by solving the model shown in Figure 5.1.

5.4 Application to General-Purpose System Modeling

To illustrate how `hoppi` is applied to the domain of general-purpose processor design and design-time adaptation, using `hoppi` I model a simple system containing a compute node, an I/O subsystem, and a storage cache between them. The compute node is running some set of workloads and accessing external storage, connected to an I/O subsystem maintaining that storage, and a I/O burst buffer acting as a storage cache connected between the compute and I/O subsystems. To illustrate the three types of `hoppi` systems (analytical models, ML models, and external simulations), I model the compute node using dynamic binary instrumentation which allows me to collect all POSIX requests sent to the I/O subsystem. Since `hoppi` does not model time (time-sensitive systems must be modeled externally), the compute node only exposes the distribution of workload's POSIX access sizes. Next, the burst buffer can be analytically modeled as a cache (Agarwal, Hennessy, and Horowitz 1989), exposing predicted miss rates with respect to cache parameters. This miss rate will affect both latency and throughput of the channel between the compute node and the cache, as well as the channel between the cache and storage. Finally, the I/O subsystem can be modeled using machine learning, exposing expected I/O bandwidth and latency given specific access size distributions. The system is given several objectives: minimize access latency for the compute node, maximize I/O bandwidth between the compute node and the cache, and maximize I/O bandwidth between the cache and the I/O

subsystem. The system is expressed as a set of nonlinear mixed-integer equations, and solved using the IPOPT solver (Wächter and Biegler 2006).

Given perfect knowledge of a specific workload, this `hoppi` system can predict I/O access latencies and bandwidth utilization with accuracy approaching those of full-system simulations. By evaluating the throughput and latency gradients with respect to e.g., cache parameters, designers can learn how increasing cache size will impact the objectives, and may model cost-performance trade-offs of such decisions. By identifying parameters that have non-zero gradients, designers can understand which parts of the system are bottlenecks and which are not, allowing them to focus resources on subsystems that have the highest potential to improve goal satisfaction. Additionally, by comparing accuracies of full system simulations versus `hoppi` estimates, designers can learn which subsystems need better sensors or modeling.

5.4.1 Need for Local Workload Models

Analyzing a range of models shows significant diversity in prediction accuracy across different workloads. This variance can be traced to the fact that for some workloads, I/O accesses are relatively independent and can be modeled using a simple statistical distribution, while other workloads may contain long dependence chains of accesses where modeling the time component is necessary for accurate system behavior prediction.

The necessary treatment to improve predictions on different sets of workloads heavily depends on the specifics of each set. Some workloads need to be described with additional features which may help improve predictions, some workloads need

different types of models, and some workloads a resistant to modeling altogether and require full system simulation.

Using design-time knowledge to model workloads is possible when designing application-specific systems, and `hoppi` has been successfully applied to modeling the data processing pipelines of the CERN Large Hadron Collider's (LHC) Compact Muon Solenoid (CMS) detector. However, there exists significant difficulty in modeling general-purpose systems due to the lack of knowledge about which workloads these systems will execute while deployed. While building separate models for different classes of workloads may be possible, these models do not help design-time adaptation since only stale workload distribution data is available during the design process. Design-time adaptation is fundamentally limited by the lack of insight into workloads, and more accurate models cannot be utilized without scheduling-time sensors observing workload distribution. Additionally, systems models that attempt to learn a wide variety of general-purpose tasks quickly explode in complexity since in general only models that accurately model time can fully describe general-purpose workloads. While the majority of workloads we observe *can* be described with simple time-agnostic models, this property is not general. Building local models for each class of workloads reduces model complexity and decomposes the problem into more practical and learnable units.

5.5 Results and Insights

By treating system design as an iterative process of (i) developing and configuring a system, (ii) deploying a system, (iii) collecting data about both the workloads and the system, (iv) and applying this data to future generations of the system, `hoppi`

allows design teams to treat the process of designing a family of systems as an adaptive system in itself. Within this framework, treating system deployment can be analyzed in terms of questions such as: what information will be gained once a system is deployed? What system sensors are most beneficial and cost effective? Is the system generation release cadence fast enough to track a changing distribution of workloads and user goals?

5.5.1 Application to System Sensors

While accurate models (whether white box, black box, or grey box) *can* be developed during design time, applying appropriate models requires insight into workloads running on the system. Design-time adaptation can utilize predictions about the future workload distribution in order to provision appropriate system resources, however, static (non-adaptive) systems are fundamentally limited by the lack of ability to adapt to workloads, and shifts in workload distribution make most design-time optimizations moot.

Two conclusions can be gained from this chapter: (1) scheduling-time insight into workloads is necessary to unlock potential optimization and adaptation benefits in domains where workloads can and do often change, and (2) building multiple independent models for different classes of workloads is necessary due to the exploding complexity of models that attempt to fit all general-purpose workloads.

5.5.2 Application to Decision-Making Systems

Many of the design-time decision-making problems such as finding optimal system configurations for a given workload or identifying system bottlenecks can be directly postponed to scheduling-time. However, design-space exploration may be too computationally expensive to perform at scheduling-time and performing it then may outweigh the benefits of scheduling-time adaptation. Local models may simplify or even solve scheduling-time decision making by solving subproblems that can be more exhaustively evaluated at design-time. These subproblems typically revolve around systems whose behavior does not depend on workload behavior, and may be explored before the exact workload distribution is known. Even if workload information is necessary, local models that target specific groups of workloads may be able to find local solutions or reduce the scheduling-time decision-making complexity to a practical level.

5.5.3 Application to System Actuators

Actuators available to design-time optimization are not strictly defined as in the case of scheduling and run-time adaptation. During the design process, system architects have the ability to change or discard any part of the system without any run-time penalty (i.e., changing a system between generations does not incur e.g., performance penalties).

If the system is designed modularly, every new generation of the system consists of a number of subsystems drawn from a pool of candidates for each role. These subsystem candidates may differ by their run-time performance, latency, energy efficiency, etc.,

and if two subsystems serving in the same role (e.g., two competing L1 cache designs) both offer Pareto-optimal but significantly different functionalities, including both designs in the final system will increase the system's behavior space. More likely, these designs have overlapping functionalities which may be shared between them in order to conserve e.g., hardware area and cost. Modular refinement and subsystem reuse is not the only method for designing adaptation actuators though, and designers may explicitly venture to design these system 'knobs'.

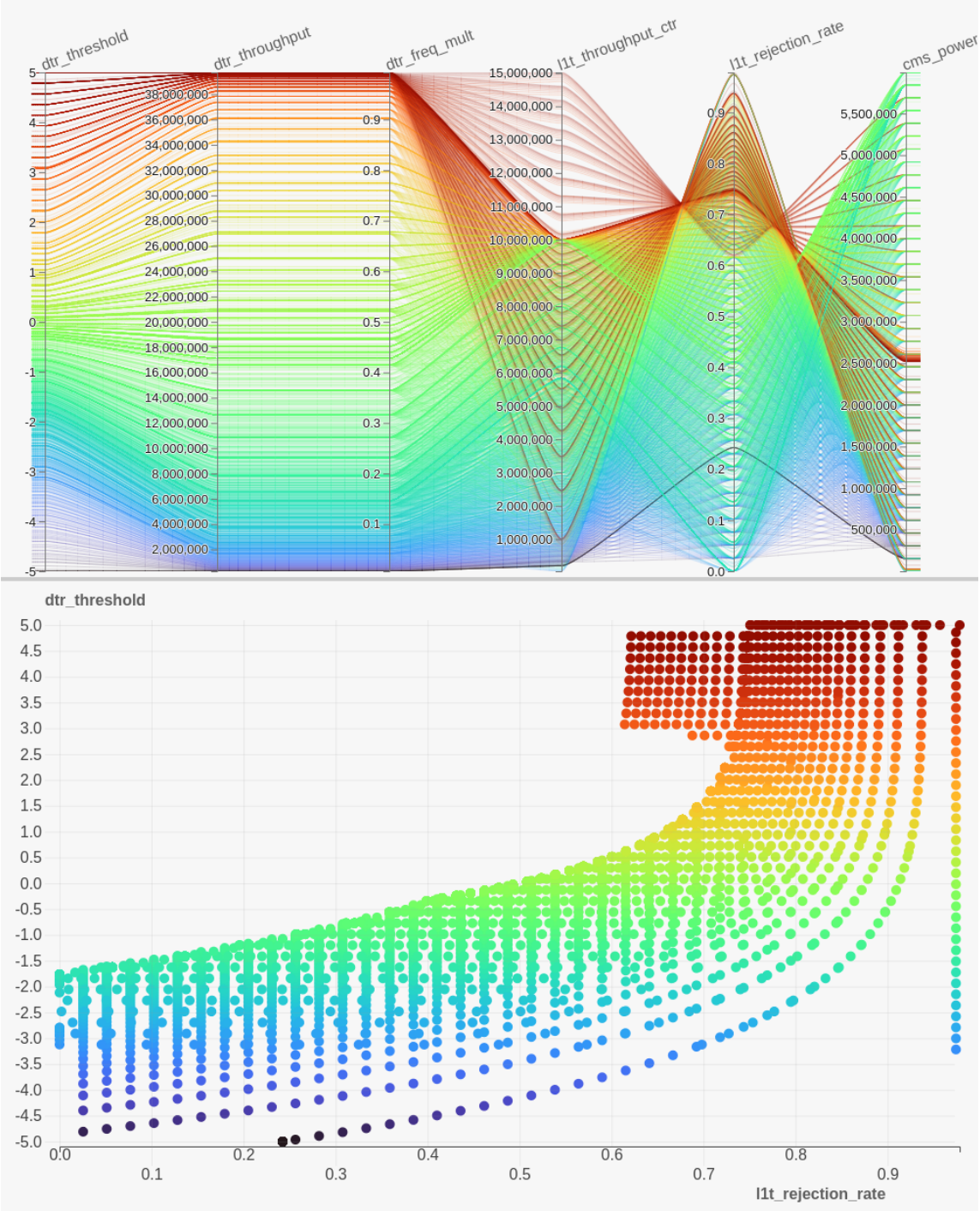


Figure 5.3: A parallel coordinate plot (top) and a scatter plot (bottom) of the set of Pareto optimal solutions over three objectives of the system model shown in Figure 5.1.

Chapter 6

STATIC SYSTEM ADAPTATION AND PRE-EMPTIVE APPLICATION OF SYSTEMS MODELS

6.1 Introduction

In this chapter we focus on applications of machine learning models *before* a workload has begun execution. This includes scheduling the workload at an opportune time, statically configuring the system ahead of time to best fit the workload, affecting future scheduling behavior to help workloads avoid contention, etc.

6.1.1 The Problem of I/O Modeling

Because of the scale and evolving complexity of high-performance computing (HPC) systems, critical gaps still remain in our understanding of HPC applications' runtime behaviors, specifically, compute, communication, and storage behaviors. This situation is further complicated by the fact that HPC applications come from a diverse set of scientific domains, can have vastly different characteristics, and are executed simultaneously, thereby contending for shared resources.

One such gap is the understanding of I/O utilization in these systems. Currently, application programmers and systems administrators still heavily rely on limited observations, anecdotes, and scattered experiences to develop design patterns for applications and manage their runtime performance either at the node level or at the system level. This approach is tractable only to the extent permitted by limited

application developer and facility support staff resources and their expertise. Therefore, automated data-driven methods are needed to streamline this process and reduce the turnaround time from capturing information to understanding and enacting improvements in I/O utilization and efficiency. One intuitive way to approach this situation is not by simply considering application performance in isolation but by identifying commonalities that reduce the volume of characterization data, simplify performance modeling efforts, and exploit opportunities for performance improvement across application domains.

Machine learning (ML) is a promising approach for the data-driven analysis of I/O performance data. This is evidenced by the growing interest in the design and development of ML-based methods for various I/O performance analysis and modeling tasks (Dorier et al. 2014; Xie et al. 2017; Madireddy et al. 2018b; D. Li et al. 2019; Pavan et al. 2019; Snyder et al. 2016). However, analyzing I/O performance is not trivial. Figure 6.1 shows that I/O throughput spans almost 14 orders of magnitude and can vary as much as five orders of magnitude for jobs with the same amount of I/O volume. Given the complexity of the I/O performance data, the relationship between the I/O performance and the factors that affect it are often nonlinear. Consequently, there is a trade-off between explainability and predictive accuracy when out-of-the-box ML methods are adopted for I/O performance analysis. In particular, the models that are explainable and intuitive to I/O experts are often simple and based on linear models. The models that have high predictive accuracy are often black box and cannot be used directly for explaining the I/O performance.

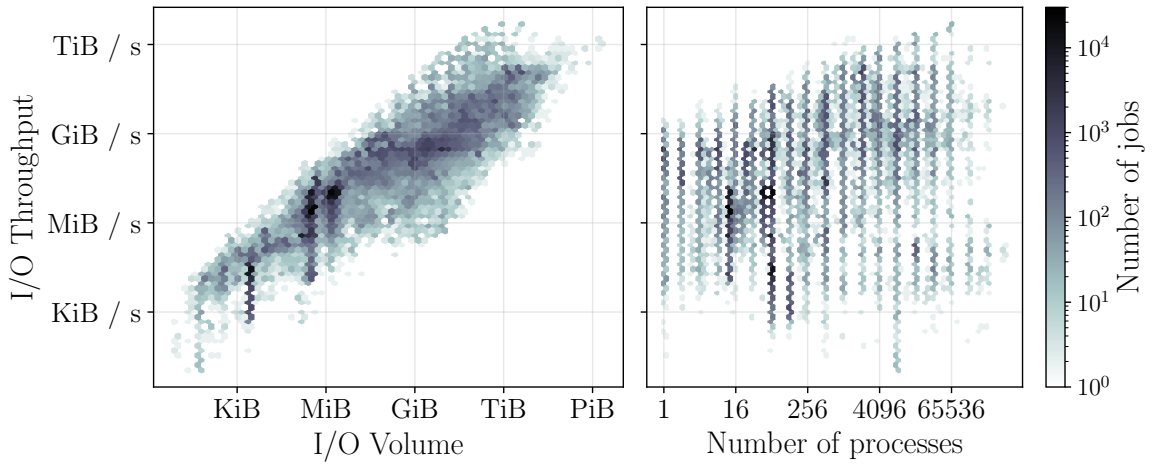


Figure 6.1: Frequency of jobs with respect to I/O throughput and the total number of bytes transferred. Data are collected from the Argonne Leadership Computing Facility (ALCF) Theta supercomputer. Note that the color bar is logarithmic.

6.1.2 ML Modeling Goals

One approach to building adaptive systems would be: (1) build a system, (2) insert best guess at needed sensors and actuators, (3) train decision-making at run-time using e.g., reinforcement learning, (4) evaluate benefits and go to step 2. However, there are several issues with this approach: (i) significant effort may be needed to build the sensors and actuators, and much of this effort may be discarded in later steps, (ii) no guarantee that the reinforcement learning is achieving a good performance exists, i.e., no baseline nor upper bound on adaptation benefits has been set, and (iii) system evaluation does not directly present an answer to what should be changed about the sensors and actuators.

This dissertation explores an alternative approach, where knowledge about a system is built using a data-driven approach, and this understanding is queried to design sensors, actuators, and decision-making mechanisms. Machine learning is a good candidate because general-purpose computing systems are complex enough that

analytically solving them is not viable. Both supervised and unsupervised learning approaches will be used to model workloads and the system. Supervised learning tasks typically attempt to learn some target value or label. Models are built that predict the target feature based on other, input features, predict model uncertainty, understand relationships between input and output features, etc. On the other hand, unsupervised learning deals with unlabeled data, and is limited to clustering data, describing the data distribution, attempting to reduce data dimensionality, or finding outliers.

6.1.2.1 Workload Clustering

Facilities such as Argonne Leadership Computing Facility (ALCF) and the National Energy Research Scientific Computing Center (NERSC) have on-staff supercomputing and I/O experts that maintain the HPC systems and work with scientists to improve the applications running on the supercomputers. The sheer number of applications makes analyzing individual jobs difficult, since expert insight does not scale with the growing number of new applications. In computer architecture, a similar problem exists: applications may be tuned with tools such as Intel VTune or NVIDIA Nsight, but this analysis can only be applied post-hoc, and on select applications.

Workload clustering — a procedure where similar workloads are grouped together — has been proposed as a method to analyze related workloads in bulk, scaling system expert insight to larger amounts of jobs. Furthermore, analyzing groups of similar jobs may reveal hidden patterns not visible when analyzing individual instances.

6.1.2.2 Workload–System Interaction Modeling

Clustering is an unsupervised learning method, and does not take advantage of any labeled data from the system. There are several such labels that may be used however, so supervised learning becomes a viable option. For example, the runtime of a job highly depends on both the system and the workload characteristics. If a machine learning model is able to accurately predict a job runtime directly from workload and system features, the model has built an understanding of how the two components interact.

6.1.3 Deliverables

We develop an explainable ML platform for I/O performance analysis to answer a number of I/O performance questions: how can we cluster applications together? Given an application or task execution, what existing I/O behavior cluster does the job fall into? What are the key characteristics of the cluster itself? Does it match the expected execution or performance profile, for example, the requested resources and the optimality of those resources' utilization? How does this job's performance rank with the rest of the cluster? What parameters influence the job placement within the cluster? How does the cluster rank with other clusters?
The goal of this work is to answer these questions, and to this end our contributions are as follows:

- We introduce a log-based feature engineering pipeline for HPC applications. Our analysis uses 89,844 Darshan logs of I/O volume greater than 100 MiB collected

on the Argonne Leadership Computing Facility (ALCF) Theta supercomputer from 2017 to 2020.

- We show that agglomerative clustering can reveal a large amount of structure in the dataset and that training models on fine-grained (local) clusters instead of on the whole dataset yields more robust and useful predictions.
- Using different ML methods, we demonstrate that despite I/O throughput varying across many orders of magnitude, we can on average predict individual job I/O throughput within ∼20% of the real value. Furthermore, we show how the interpretation of ML prediction models can yield useful advice for increasing application performance.
- To validate the practicality of the proposed feature engineering pipeline and the clustering techniques, we introduce Gauge, an exploratory I/O throughput analysis tool with adjustable data granularity and interpretable I/O throughput models. We illustrate how it can be used by system owners and I/O experts to optimize the HPC clusters for the workload present or by application developers to optimize their jobs.
- We release a web-based version of Gauge, information about the tool is available at http://gaugeviz.org.

6.1.4 Application to Adaptive Systems

The goal of modeling workloads and systems is to better design adaptive systems. By understanding workloads and the system through data-driven modeling, insights can be gained that improve all four of the adaptive system stages (sensing → deciding → adapting → learning → ...):

Application to sensing mechanisms: By building models of the system as well as models of the workload distributions, these models can be queried, and connections between input features (e.g., hardware performance counters) and target features (e.g., system utilization or job runtime) can be discovered. These input features may not necessarily exist in a given system, but knowledge of these connections can inform a designer which sensors / monitors to create and expose to the adaptive system.

Application to decision-making mechanisms: System models can be analyzed and interpreted to discover *why* a certain system behaves as it does. Building black-box models of systems may be more practical than analyzing real systems, both because of the computational difficulty of simulating complex systems, and because ML model interpretation techniques are applicable to the black-box models. Furthermore, the same models can be later used in run-time decision making. For example, a model of the system can be used to predict e.g., the job runtime for different system adaptations, so that the best adaptation can be chosen. Finally, system models can be evaluated by their predictive accuracy, and fundamental limits of decision-making capability can be discovered. Adaptive systems can use this information at run-time to evaluate decision-making uncertainty.

Application to adaptation mechanisms: By having knowledge of the workload distribution as well as a set of sensors and actuators, different adaptive system proposals can be evaluated, and benefit-complexity trade-offs can be made. The workload distribution can be used to weigh benefits of rare workloads versus the increased decision-making complexity. Proposals for new actuators can be evaluated in the context of available sensors and the decision-making capability, providing an estimate of benefits new actuators can bring. The models of the system can be

queried to evaluate different adaptation options, in a more efficient manner than simply simulating the whole system.

Application to learning mechanisms: By building models exposed to limited sets of data, model generalization capability can be exposed. This generalization capability is a proxy for how well the decision-making mechanisms will behave when faced with new workloads. If generalization is limited, the decision-making mechanisms may need to learn at run-time (typically called "on-line learning"). Benefits of on-line learning can be evaluated using the same models through training with a new cross-validation regime. For example, models may be trained on and evaluated on job logs sorted by time. This training regime would be representative of what a real adaptive system would face once deployed into the field.

6.2 Static Adaptation Use Cases

Here I provide several use cases of why sensing and decision making are beneficial when scheduling workloads:

6.2.1 Pre-emptive System Reconfiguration

Computing systems may offer configuration knobs that modify system behavior, and can be changed at startup or during execution. A statically-adaptive system with some configuration capability may be reconfigured at startup or before running a workload to achieve better user goal satisfaction. The system configuration knobs can be treated separately from workloads, as the workloads and runtime knowledge / constraints may change independently. Some examples of such knobs are e.g., BIOS settings allowing system administrators to turn off microarchitectural features such as multithreading or limit the range of frequencies a processor can run at, or MPI hints that inform the I/O subsystem of workload access patterns and provide suggestions on how to handle certain operations. For example, a program may inform the I/O subsystem of a preferred striping unit, which specifies how data is split across individual drives. The hint may override a default system value, but the system is free to ignore this hint, as correctness is preserved either way.

Not all configuration knobs are necessarily beneficial when used, e.g., in the case of data sieving or access coalescing, where many small accesses may be buffered and sent through as a single, larger access. Since sieving may happen on multiple levels of the I/O subsystem hierarchy, a user forcing the system to coalesce accesses may not yield performance benefits, or may even decrease throughput. Similarly, certain ISAs offer

prefetch instructions, which preemptively load data into caches. With the advance of hardware prefetchers, ISA-level prefetching has largely been rendered useless, and only causes an increase in instruction cache misses. Modern CPUs often ignore these instructions and treat them as no-ops (no operation). Experience has shown that simply exposing more system configuration options does not lead to increased goal satisfaction, as only hardware designers and low-level system programmers have the necessary expertise to set these options.

Nonetheless, providing greater amount of system configurability can be highly valuable in certain domains, e.g., real-time computing, where users may prioritize predictability over raw performance, and removing e.g., highly-speculative behavior is warranted. Historically however, system configuration knobs are rarely used, as the knowledge necessary to understand their inner workings and impact on programs requires possibly proprietary insight into the system systems typically run at same configuration for all workloads, and cannot change their behavior with changing workload or user needs.

Static system configuration can be considered a form of static adaptation. The system may have a wide variety of possible behavior modes, but has to select one for the duration of core time allocation or the lifetime of a process.

6.2.2 Improving Quality-of-Service with Intelligent Scheduling

Multi-socket systems and larger processors may have multiple NUMA (Non-Uniform Memory Access) domains. Here, the processors have multiple memory controllers, each connected to separate DRAM modules. The processors maintain a uniform view of memory, i.e., each core is able to access memory on any of the DRAM modules.

Due to the physically distributed nature of DRAM memory, a processor has faster access to closer DRAM modules, and may need to send requests over the Network on Chip (NoC) or even to other sockets in order to fetch distant memory.

For more than a decade, the process scheduler in the Linux kernel was unaware of process memory placement, and would often place processes on the least-busy core, possibly far from the processes' memory, causing large performance degradations (Lozi et al. 2016). Recent NUMA-aware schedulers attempt to keep processes local, and make more complex trade-offs between optimizing for CPU utilization and memory access locality. Solving NUMA-aware scheduling required manual diagnosis, architectural insight, and hard-coded implementation. There currently may exist other unknown performance degradations that require similar changes, but are undetected due to the sheer complexity and opaqueness of modern computing systems.

If the scheduler had learning capabilities and the right sensors, it may have noticed that a certain process has higher or lower commit rates on different cores, as loads and store instructions are on the critical path and stall the pipeline. The scheduler may have experimented with different core-process placements at runtime, in order to explore the decision state space. Even without explicit NUMA-awareness, the above listed issue may have been solved by having sensing and learning as part of the scheduling algorithm.

This intelligent scheduling can be considered a form of static adaptation. While measuring process commit rates requires runtime sensing, the only decision making and configuration happens at scheduling time.

6.2.3 Isolation of Resource-Heavy or Misbehaving Jobs

On conventional processors, only one process uses a core at a time (or two or more on cores that support simultaneous multithreading) thereby monopolizing core-local resources for a period of time, other resources are shared between the processes. These may include any shared caches, e.g., L3 caches (and less commonly L2 caches on some generations of processors), memory controllers, network on chip, other I/O such as networking and storage, etc. This contention exists in the spatial domain, i.e., these resources are being used simultaneously by different physical cores. Contention can also exist on the temporal domain, e.g., in the case of two processes alternatively being scheduled on the same core, both of which are using a large portion of e.g., L1 cache, and which observe large amounts of conflict misses after being scheduled. In the I/O domain, jobs that have disproportionate amounts of impact on neighboring jobs are called 'bully' jobs (Xu Yang et al. 2016). I/O subsystems are particularly sensitive to contention due to their nature and complexity, since a bully job can asynchronously issue many I/O operations and is not limited by the response time of the system they are abusing.

The above listed situations are problematic for a number of reasons: First, while some programs have legitimate reasons to attempt to monopolize a resource, many misbehaving programs *do not* achieve good system utilization, and are wasting system resources while preventing other programs from making progress. Second, although fairly portioning e.g., processing time is trivial, achieving fair distribution of other resources is more difficult, e.g., in the case of network or storage bandwidth. While mechanisms could be plausibly built to ration these resources, rationing certain (espe-

cially low-level) resources may incur an unacceptable performance penalty. Therefore, fairness is violated by bully jobs.

One solution to monopolization-induced contention and unfair resource distribution is job isolation. While the majority of jobs are unhindered by the system and allowed to work as normal, once identified, bully jobs access to share resources is rationed, *at the expense of the bully*. By not imposing a performance penalty on other jobs on the system, this approach may be appropriate in many domains, e.g., I/O rationing or microarchitectural attack prevention.

This isolation method can be considered a form of static adaptation. While runtime detection is necessary to spot bully jobs, jobs are placed in isolated hardware or software environments at scheduling time.

6.3 Workload Clustering

I use the term 'clustering' to mean a bipartite mapping between one set of clusters and one set of samples. A 'clustering' is therefore a set of all clusters, and a cluster 'contains' all samples it is connected to in the bipartite graph. Commonly, each sample belongs to zero or one cluster, though this requirement is sometimes relaxed, e.g., in the case of edge samples between two clusters. The process of clustering involves finding the set of clusters according to some measure of clustering quality.

6.3.1 Clustering Algorithm Requirements and Methods

Clustering has been extensively explored, with hundreds of published algorithms working under vastly different assumptions and offering different results. Several requirements limit which clustering algorithm can be applied to workload logs.

First, several clustering algorithms such as k-Means assume that the number of clusters is known ahead of time. When modeling workloads, even if the number of applications running on the system is known, there is no guarantee that all jobs from the same application should belong to a single cluster. This requirement disqualifies clustering algorithms such as k-Means and k-Medoids, which require the number of clusters ahead of time.

Second, workload sampling or density is not a property of the domain itself (as would be in e.g., biology where some species are simply less prevalent). How many times an HPC workload is ran for is relatively independent of the workload behavior, and depends on research needs, potential impact, project funding, deadlines, etc. Therefore, certain workloads may be significantly overrepresented, and a clustering

algorithm must be able to work when clusters can have orders of magnitude different densities. This requirement disqualifies clustering methods such as Gaussian Mixture Models.

Third, the distribution of workloads may be heteroskedastic, i.e., the variance of certain random variable changes with another variable. For example, a common pattern in HPC is that certain applications may have large I/O throughput variance when their I/O volumes are low, but have consistent I/O throughput when jobs are past a certain size. The clustering algorithm must be tolerant of heteroskedasticity, which eliminates density-based clustering methods such as DBSCAN, as DBSCAN assumes a certain acceptable variance within a cluster set by its ϵ parameter.

6.3.1.1 Comparison of Clustering Algorithms

In (Rosario et al. 2020b), we compare four different clustering methods: k-Means, Mean Shift Clustering (MSC), Expectation Maximization using Gaussian Mixture Models, and Density-Based Spatial Clustering of Applications with Noise (DBSCAN). We use Variation of Information (VI), an information-theoretic measure that evaluates how much information is lost, and how much is gained when moving from one clustering to another.

By repeatedly clustering the Theta and Cori datasets with different random seeds, clustering stability can be evaluated. By clustering with different parameters (e.g., number of clusters in k-Means), sensitivity to parameters can be evaluated. In (Rosario et al. 2020b) we use VI to evaluate whether different clustering algorithms repeatedly arrive at the same clusterings. When VI is high, either one of the clusters has low entropy (all samples are in a single cluster or all samples are colored as outliers), or

the clusterings are significantly different. When VI is low, clusterings are similar. An appropriate clustering method will have low VI when comparing clusterings initialized with different seeds and parameters, i.e., it will not be sensitive to noise and poor parameter selection.

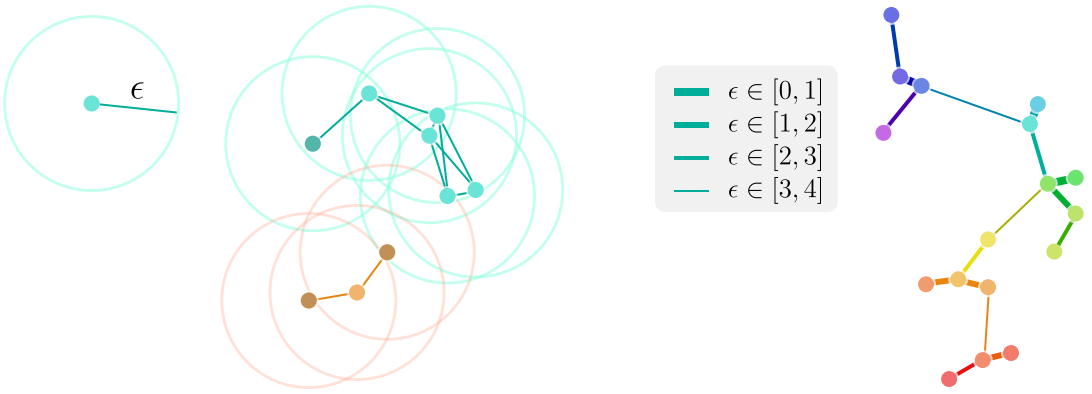
(Rosario et al. 2020b) shows that Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) (Campello et al. 2015) satisfies all of the above listed requirements, and reproducibly achieves a low VI measure between repeated clustering attempts.

6.3.1.2 DBSCAN and HDBSCAN

Since HDBSCAN is hierarchical version of the DBSCAN clustering algorithm, I will first explain DBSCAN. Density-Based Spatial Clustering of Applications with Noise (DBSCAN) (Ester et al. 1996) is an agglomerative, non-parametric, density-based clustering method. Due to its non-parametric nature, DBSCAN does not require tuning a parameter (unlike e.g., k-Means, where the number of clusters must be tuned), and typically works better on novel datasets than parametric clustering algorithms. DBSCAN has an average computational complexity of $O(nlogn)$, and a worst case complexity of $O(n^2)$.

DBSCAN is initialized with a distance parameter ϵ, and a minimal number of neighbors n. Executing DBSCAN involves the following steps:

1. Given some distance metric, for each sample in the dataset, find neighboring samples within a distance ϵ.
2. Build an undirected graph where samples are nodes in the graph and neighbors are connected with edges.



(a) An example DBSCAN clustering. (b) An example HDBSCAN clustering.

Figure 6.2: Non-hierarchical and hierarchical DBSCAN.

3. Color each node with more than $n-1$ neighbors as a 'core' node.
4. Color each node with less than $n-1$ neighbors as 'edge' node.
5. Observing only core nodes, find connected components in the graph, and report each component as a separate cluster.
6. Append edge nodes connected to a single connected component to that cluster.
7. Depending on implementation, append edge nodes connected to a multiple connected components to neither, both or one of the clusters.
8. Color edge nodes without core node neighbors as outliers.

The advantage of DBSCAN are that it makes no assumption about the shape of a cluster as long as the cluster maintains a density above ϵ. This property is beneficial when encountering new datasets where no domain knowledge exists. Figure 6.2a illustrates an example DBSCAN clustering, where the ϵ value is set by the radius shown on the left, and $n = 2$. Notice that DBSCAN has no difficulties clustering a crescent-shaped cluster (teal color).

HDBSCAN is an extension of DBSCAN that instead of producing a set of individual clusters, effectively runs DBSCAN at every ϵ value between 0 and infinity. When $\epsilon = 0$,

DBSCAN will color each sample as an outlier, while when $\epsilon = \infty$, DBSCAN will group the whole dataset into a single cluster. HDBSCAN tracks the transition between the two extremes, and marks at which ϵ values do small clusters merge into larger clusters. At the core of HDBSCAN is the minimum spanning tree algorithm: when ϵ values are low, only the closest samples will cluster together, and as ϵ grows, the closest unconnected clusters or samples merge into larger clusters. Figure 6.2b illustrates such an example, where a minimum spanning tree is built and edge thicknesses depict distances.

The benefit of HDBSCAN over DBSCAN is that no knowledge about data density is needed to perform clustering, and users are free to explore the hierarchy instead of a fixed 'cut' at a certain ϵ value. A typical method for analyzing an HDBSCAN clustering is by visualizing a tree, where the y coordinate represents ϵ values, the root node represents the whole dataset in a single cluster and is positioned at the smallest ϵ value where the whole dataset is in a single cluster, and the tree bifurcates whenever ϵ is low enough for clusters to split.

6.3.2 Gauge: a Workload Exploration and Analysis Tool

In (Rosario et al. 2020b) we introduce Gauge, an interactive, web-based, data-driven HPC I/O exploration tool. The goal of Gauge is to enable HPC facility staff to analyze large amounts of unsorted workload logs, understand current system behavior, diagnose performance problems in workloads, and help scientists optimize their applications.

Gauge works by hierarchically clustering HPC job logs using HDBSCAN and presenting a tree-based visualization as shown in Figure 6.3. Each node in the

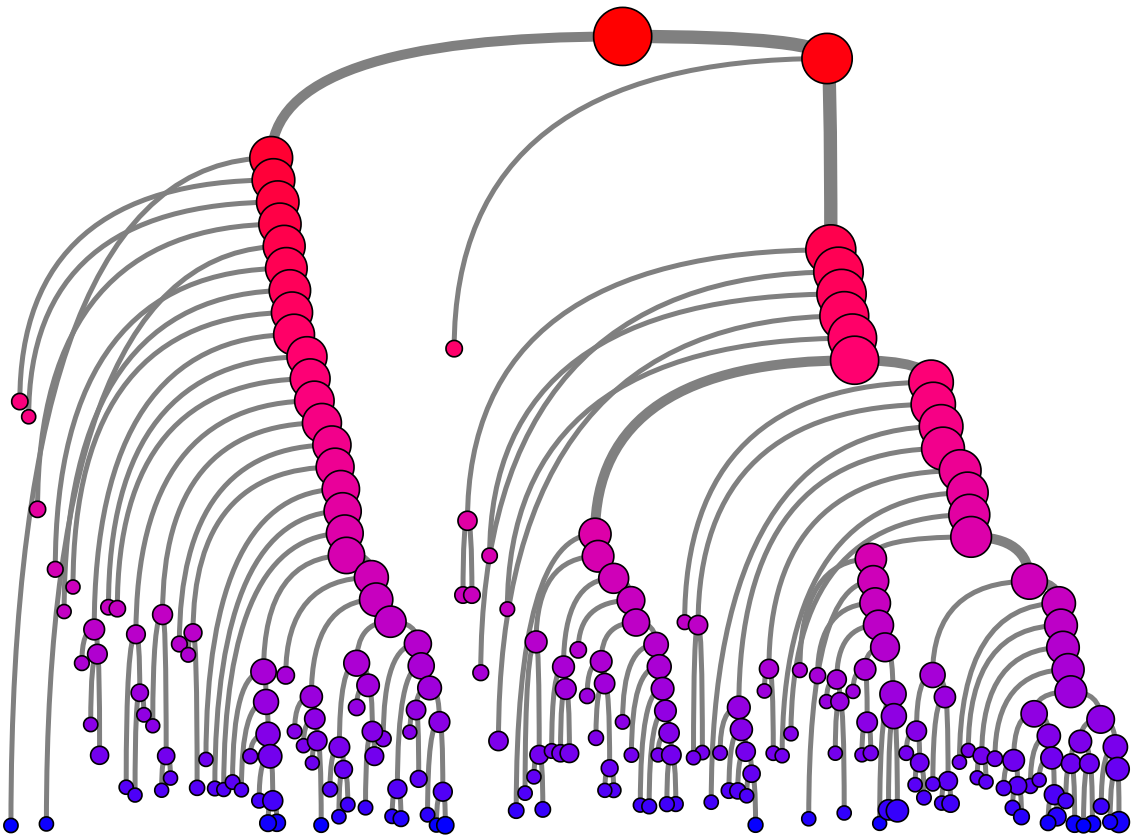


Figure 6.3: The Gauge cluster hierarchy on the Theta dataset.

hierarchy represents a cluster, where the node y position represents the effective ϵ value at which DBSCAN would split that cluster, the x position does not carry any information, and the node size reflects the number logs in that cluster. The top node in the hierarchy contains the whole dataset, i.e., all of the logs are in the same cluster. As the ϵ parameter is decreased, nodes get split into smaller, more dense clusters.

Gauge is interactive, and allows users to click on up to four clusters in the hierarchy, which opens a panel as seen in Figure 6.4. Each panel has a number of graphs: the distribution of applications the job logs belong to and the users that ran the jobs, a parallel coordinates graph of ratio features (features bounded between 0% and 1%), a parallel coordinates graph of absolute-valued features (unbounded features whose

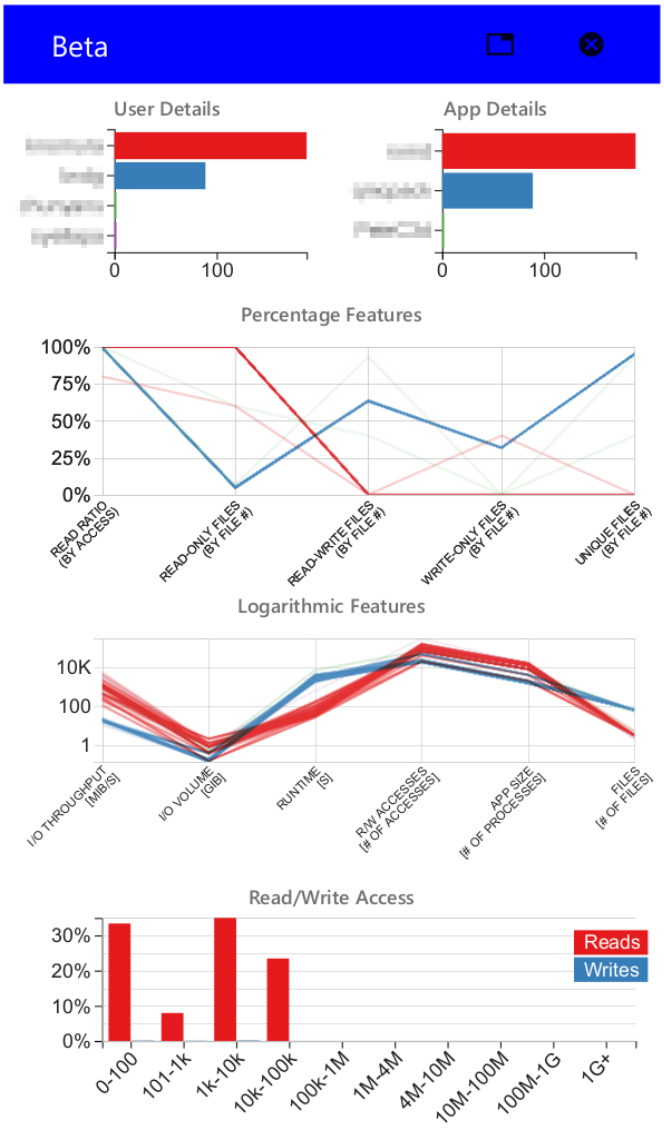


Figure 6.4: An example Gauge cluster column plot with five graphs.

values are shown on a logarithmic scale), as well as a histogram of common POSIX access sizes.

6.3.2.1 Interpreting Dataset Structure

While Figure 6.3 reveals the existence of a rich structure in the dataset, it does not increase our understanding of it. To get better intuition about the data distribution, we perform a simple experiment: since every node of the HDBSCAN single linkage tree consists of a number of smaller merged clusters, we train a decision tree that predicts where each job in the cluster will end up once the cluster splits. To help interpretability, we train decision trees only of depth 1, namely, trees with only 2 leaves and a single decision splitting the dataset. The annotations and arrows pointing to nodes in Figure 6.5 explain what the decision trees at those nodes have learned.

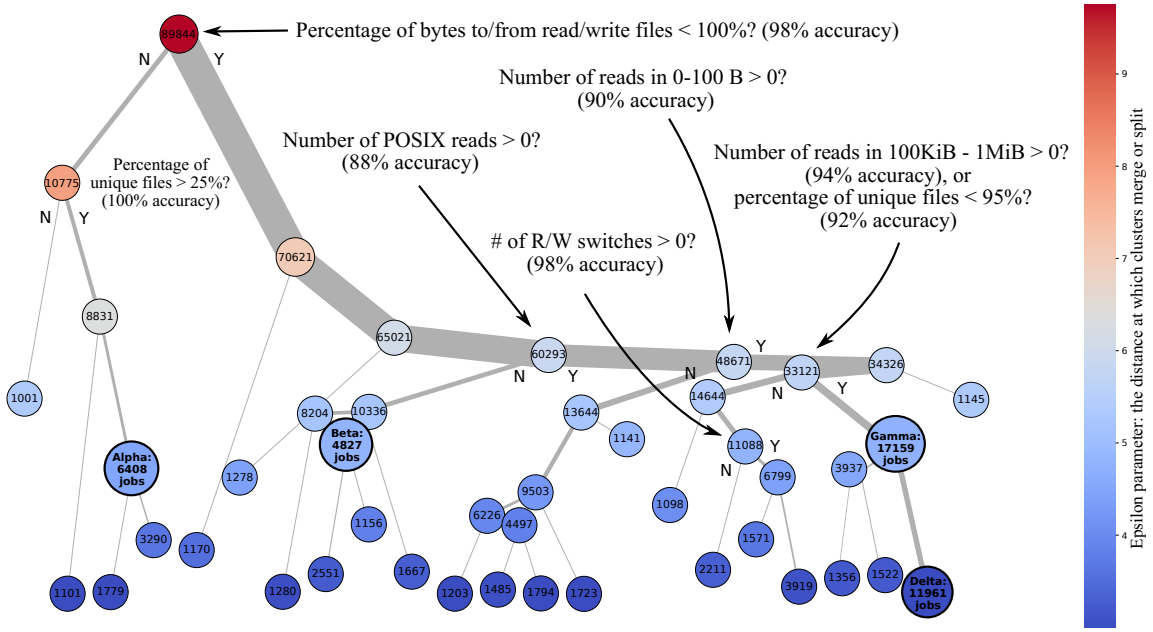


Figure 6.5: HDBSCAN single linkage tree, pruned of clusters smaller than 1,000 jobs and of clusters clustered at $\epsilon < 3$. Note the four clusters marked Alpha to Delta. These clusters are hand-selected using this tree and are used later in the analysis.

Right away, the clustering splits the dataset into jobs that use only read/write files and jobs that also use read-only and write-only files (top cluster's annotation).

Although this split is imbalanced (70K jobs vs. 10K jobs), the accuracy of the decision tree (98%) is high enough to still be informative. Looking at its smaller child (node with 10,775 jobs), we see that it splits the dataset with perfect accuracy into jobs that have more or less than 25% of unique files (unique files are files accessed only by a single process). Note that other nodes' decisions might not be so accurate. This is due to our choice of using a decision tree with a depth of 1. If we allow deeper decision trees, the accuracy of decisions increases, but interpreting these models becomes more tedious. In practice, we use the HDBSCAN tree in a more interactive process, allowing us to test different decision trees using different features, depths, and acceptable accuracies.

Through analyzing this tree in more depth, we concluded that the clusters in the dataset occupy very distant spaces, showing different behaviors across several and often tens of different features. As we decrease ϵ, the clusters get smaller, and the jobs within them more and more similar, until we arrive at just dozens of almost identical runs. We claim that to analyze throughput and extract insight out of a cluster, first we must select the right granularity at which to make the analysis. Too fine, and we may be learning the behavior of a single job, not any general trends shared by different applications running on the HPC system. Too coarse, and we may arrive at very general interpretations that are not specific to the jobs we are interested in.

6.3.2.2 Deeper Insight with Interactive Parallel Coordinate Plots

If column panels do not show sufficient information to understand or diagnose a cluster, users may build a larger parallel coordinates plot with features of their choice through HiPlot (Haziza, Rapin, and Synnaeve 2020). HiPlot is an interactive, web-

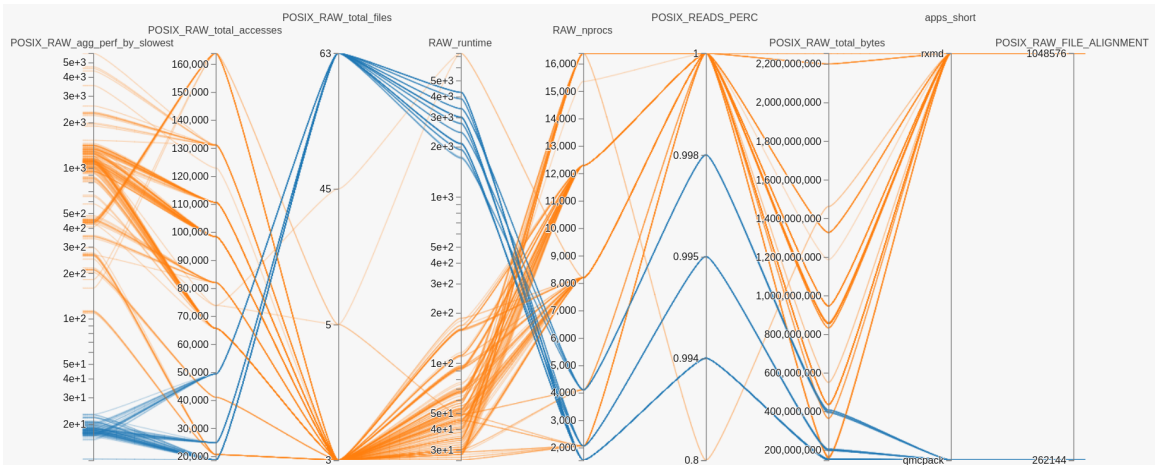


Figure 6.6: Example HiPlot parallel coordinates graph with user-selected features.

based parallel coordinates graph tool that offers flexible selection of axes, interactive selection of feature ranges, optional histograms and 2-dimensional scatter plots, and easy integration into existing web applications. An example HiPlot graph with (1) I/O throughput, the number of total POSIX (2) accesses, (3) files, (4) total runtime, (5) number of processes, (6) the percentage of POSIX accesses that are read operations, (7) number of total bytes read or written, (8) application names, and (9) the common file alignment. Here, HiPlot automatically decided which features to present on a linear, logarithmic, or a categorical scale, e.g., in the case of I/O throughput, total POSIX accesses, and application names, respectively.

6.3.3 Grapes: a Domain-Agnostic Dataset Exploration Tool

While Gauge initially targeted the HPC I/O domain, all of the steps Gauge performs can be generalized to other system analysis domains such as compute and network bottleneck analysis, microarchitectural modeling, malware detection, side-channel attack prevention, etc. While different systems may record different features

which may require new types of preprocessing, and different domains may have different metrics of how samples are compared, these nuances can be user-specified while the core of the Gauge tool is shared across domains.

Grapes is a generalized, domain-agnostic implementation of Gauge that replaces I/O specific components of Gauge with parameterizable, user-specified options. While Gauge only supports tabular (matrix) data, where each column is a different feature and each row is a different job, Grapes supports other formats such as images, temporal sequences such as sounds or stock prices, or even videos. As long as the user provides either (1) a metric function that can compare two samples, or (2) a distance matrix, Grapes can perform the clustering. It is the user's responsibility to provide a domain-appropriate metric function. For example, when individual samples are images, the Manhattan distance would likely not be an appropriate metric function, since two images might be very similar but have significantly different L1 distances.

6.4 ML Modeling of General-Purpose Computing Systems

There are several reasons why practitioners might build machine learning models:

- These models may be used to automate processes typically performed by humans, as in the case of optical character recognition systems.
- These models may be used in scenarios where human intervention is too slow or unscalable, e.g., in the case of high-frequency trading.
- Models may be analyzed to better understand the underlying process being modeled, as in the case of ML-based medical diagnosis interpretability.
- Models may be built to quantify uncertainty during decision-making in a certain domain, e.g., in the case of ML-guided prediction markets.

All four use cases are of importance in adaptive systems: (1) adaptive systems need ML to predict which configurations will be most beneficial in the future, (2) these states cannot be decided at design-time by computer architects, but must be chosen automatically, (3) these models must be analyzed during the design process to understand what sensors, actuators, and model architectures they can benefit from, and (4) by providing uncertainty estimates, improved quality of service can be achieved.

Therefore, this dissertation will broadly focus on three ML model use cases:

1. Predictive models and model-based control, whose goal is to map system and workload inputs to decisions,
2. Model interpretability, whose goal is to learn some part of system behavior and explain it,

3. Uncertainty quantification, whose goal is to evaluate system output sensitivity to external system inputs and internal system states.

6.4.1 System and Workload Modeling Formulation

This dissertation focuses on models that learn to map inputs to outputs, e.g., by predicting the effect of certain workloads to system states. In this chapter we use workload and system profiles described in Section 4.1 as inputs, and use machine learning models to predict the measured system I/O throughput.

To predict I/O performance, we first need to define a metric for evaluating predictive accuracy. Common metrics such as L1 or L2 loss might not be a good fit for our data because the throughput ranges from KiB/s to TiB/s and hence will penalize jobs with higher throughput more than the low- and medium-range ones, forcing our models to ignore the latter. Therefore, we adopt root mean squared logarithmic error (RMSLE) and mean absolute logarithmic error (MALE) functions:

$$\begin{aligned} RMSLE(y, \hat{y}) &= \sqrt{\frac{1}{n}\sum_{i=0}^{n}(log_{10}(y_i) - log_{10}(\hat{y}_i))^2} \\ &= \sqrt{\frac{1}{n}\sum_{i=0}^{n} log_{10}\left(\frac{y_i}{\hat{y}_i}\right)} \end{aligned} \tag{6.1}$$

$$\begin{aligned} MALE(y, \hat{y}) &= \frac{1}{n}\sum_{i=0}^{n} |log_{10}(y_i) - log_{10}(\hat{y}_i)| \\ &= \frac{1}{n}\sum_{i=0}^{n} \left| log_{10}\left(\frac{y_i}{\hat{y}_i}\right)\right|. \end{aligned} \tag{6.2}$$

Both of these equations penalize the *ratio* of the prediction vs. the real value. Hence the ML model is scale independent and should equally penalize relative prediction errors on both small and large throughput jobs. Since our ML models already receive

percentage and logarithmic features and are tasked with predicting the logarithmic I/O throughput value, they can natively use MSLE/MALE; that is, we never have to convert the predictions back to the raw values. In subsequent text, we choose to use MALE over RMSLE, since MALE is less sensitive to outliers and is more directly interpretable. Using MALE allows us to directly translate median errors to English, for example, by saying that the model on average predicts I/O throughput with $1.15\times$ error. For example, if a model predicts a throughput of $13GiB/s$ for a job that in reality achieves only $10GiB/s$, the MALE error is $MALE(10 \times 10^9, 13 \times 10^9) = |-0.114|$. To interpret this loss, we calculate the relative error $10^{|-0.114|} = 1.3\times$. The same value is calculated if we swap the prediction and target value; that is, this model may underestimate or overestimate throughput. In either case, we have a good estimate of the range of the target value.

6.4.2 Black-Box vs. White-Box System Modeling

Two design choices decide how a system can be modeled. The first question is whether domain knowledge can used to build models, i.e., if this knowledge can inform the designer which machine learning class or architecture to use? Secondly, is the system a black-box, i.e., its internals are opaque, or are inner mechanisms visible and can be measured? When domain knowledge is available, designers can build analytical models, where model outputs are limited by an analytical model of the underlying process. For example, analytical models of HPC I/O have been studied in (Isaila et al. 2015), and physics-constrained models in (Karniadakis et al. 2021). When domain knowledge is not available, general, data-driven models may be used. Many classes of machine learning models have formal proofs of universal approximation – a

property where the model can learn *any* continuous function. This proof guarantees that a model can learn system behavior in response to inputs, given enough noiseless data. For this dissertation's purposes, systems are too diverse and the scope too large to build encompassing analytical models of both e.g., HPC systems and processor microarchitectures. That is why this dissertation focuses on data-driven instead of analytical models.

The second questions revolves around whether systems are opaque or (semi-) transparent. Opaque systems may not be modelable if they exhibit non-deterministic behavior, and even if they are deterministic, the amount of data needed to correctly model them grows exponentially with the amount of hidden state or memory they possess [9] Nonetheless, we assume that the systems we are studying are not actively adversarial and are not specifically designed to be opaque. As we will show, ML models are able to accurately predict the behavior of such systems up to a certain accuracy, after which these assumptions need to be reconsidered, as we will do in Chapter 7.

[9] A sketch proof of this property involves a simple Moore machine implementing a combination lock. The machine has an n-bit input $I \in 0, 1^n$, a single-bit output, m-bit state $S \in 0, 1^m$, an output function $o(I, S)$, and state transition function $t(I, S)$. The combination lock functionality outputs a '1' when the last k inputs were some correct combination of values (i.e., a key or a passcode), and otherwise outputs a '0'. The questions is: how complex of a code can the machine implement given only m bits of state, i.e., how large can k be?

The system is considered opaque since it reveals no information about the closeness of the input sequence to a target passcode. Any model of the system cannot accurately learn the behavior of this system *unless* the input-output dataset contains the passcode as an input. As the passcode grows with k, the probability of the correct passcode being randomly generated when sampling the system exponentially decays.

6.4.3 Modeling I/O Performance

We now attempt to create accurate I/O throughput prediction models. The goal of modeling I/O performance, instead of simply observing it, is to potentially develop accurate models and analyze them for underlying causes of over-/underperforming I/O throughput. If the model has good predictive power and generalizes well (can accurately predict I/O throughput of new jobs), we can apply ML model interpretation and explanation methods to answer questions such as what parameters influence this job's I/O throughput the most and what steps we should take to improve performance. We have evaluated a number of different types of ML models, such as linear regression, decision trees, random forests, gradient boosting machines, and neural networks, and have chosen to use XGBoost (Chen and Guestrin 2016) for our predictions. XGBoost is a gradient boosting machine library that shows excellent performance on tabular data and is simpler to tune compared with other powerful models such as neural networks.

In Figure 6.7, we present training and test errors of an XGBoost model trained on the whole training set (box plots on the right marked "Global"), as well as a number of XGBoost models trained on clusters of various granularities (discussed later). The global model achieves a median error of less than 1.2×; that is, half of the predictions are less than 1.2× off the true value. Through discussion with HPC domain experts and system owners, we learned that this sort of misprediction is acceptable, since I/O throughput can vary by orders of magnitude. Furthermore, pushing this error lower may be difficult because of external, unobservable factors such as I/O weather (Lockwood et al. 2017). Since our analysis here does not take into account

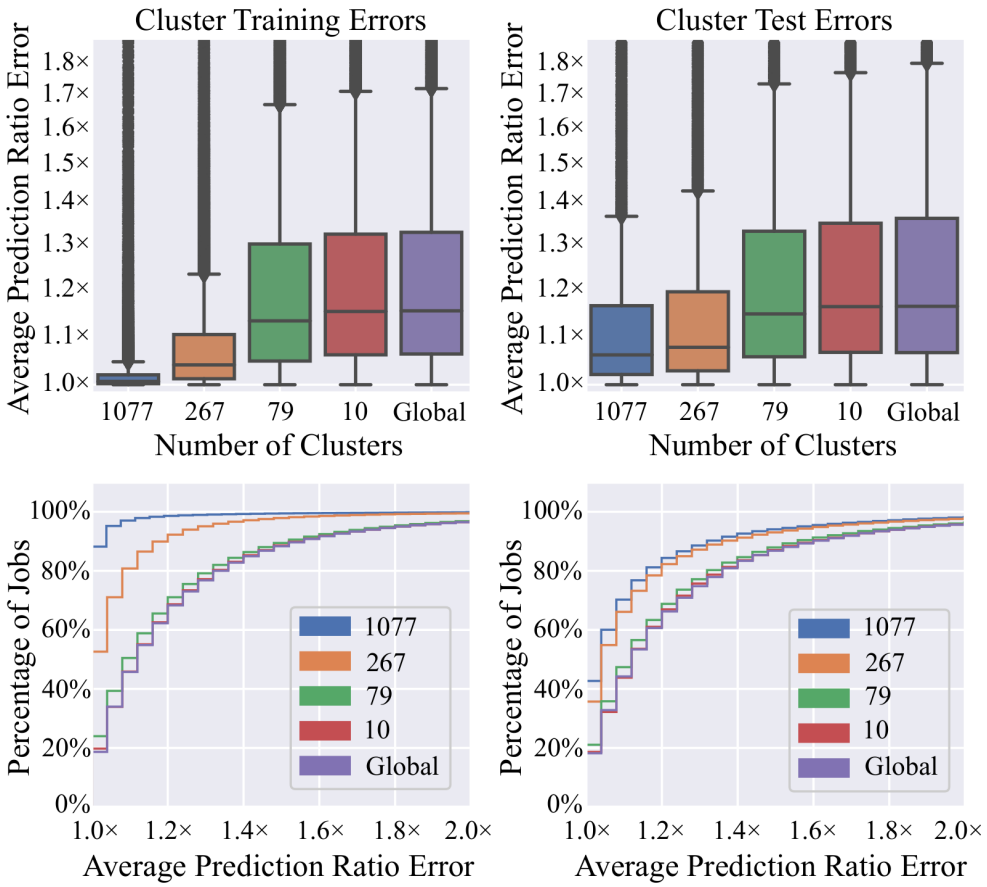


Figure 6.7: I/O throughput prediction results. Top row shows the training and test error distribution of XGBoost models trained on clusters of various granularities. The rightmost model is a single model trained on the whole dataset. The bottom row shows the cumulative histogram of the above error.

the system's I/O contention present during the job's execution, we fundamentally cannot achieve better predictions than I/O weather would allow.

6.5 Model Interpretation and Knowledge Extraction

Colloquially, computer science algorithms follow a "algorithms in - answers out" approach, i.e., programmers write algorithms that process data and arrive at answers. Machine learning algorithms can be seen as the opposite, "answers in - algorithms out" approach, where ML practitioners supply data, and arrive at algorithms or 'models'.

Model interpretation techniques attempt to either understand (1) how a machine learning model arrives individual predictions (e.g., explain a single medical diagnosis), or (2) derive a transparent program from an opaque ML model. These techniques can play a central part in ML-guided system design, as I will explain.

6.5.1 Feature Importance Evaluation

Given that an adaptive system may have tens of thousands candidate hardware sensors or monitors, finding the optimal subset of them to implement is crucial. Understanding which sensors or monitors allow the decision-making system to choose good adaptation states is essential for two reasons: first, ranking sensors and monitors by importance allows system designers to dedicate resources and effort to the most useful sources of information. Second, the same ranking allows experimentation with new sensors, where designers can insert potential sensor candidates and evaluate how they fare against existing ones. In (Isakov et al. 2020), we propose that Permutation Feature Importance (PFI), a method for ranking ML model input feature importance, can be used to decide which set of sensors a system should implement.

6.5.1.1 Permutation Feature Importance (PFI)

When a working machine learning model loses access to one of the many input features it uses to make predictions, its accuracy will be degraded. Removed features that cause a greater impact on performance can be said to be more important. PFI uses this observation to iteratively rank features, starting from the least important.

Directly removing features is not straightforward though, since 'removing' a feature is not typically defined. A naive approach would be to replace all values of a given feature with some constant, e.g., zero. This approach is flawed, since different constants will have a different effect on the model, and zero may not always be the best choice. For example, assume that a feature has a mean value of $x >> 0$, and the model is a neural network with ReLU activations. If the feature is removed, many of the neurons counting on the feature will now have significantly lower activation sums, and ReLU may always saturate neuron outputs to zero, thereby preventing the network from making *any* inference. PFI proposes that instead of replacing values with a constant, the values should be randomly permuted. There benefits of permuting values are two-fold: (1) the model still receives values with the same mean and variance on average, (2) since the permutation is random, the new feature *carries no information*. PFI has the following steps:

1. For each of the n features in the dataset, calculate importance:
2. Permute values of feature n for all rows in the test set.
3. Evaluate new model accuracy.
4. Tag feature with lowest impact on accuracy as least important
5. Remove feature, retrain model, repeat

PFI is useful due to its efficient nature, since ranking features only requires

evaluating the model on the test set n^2 times, where n is the number of features, and the model does not require retraining. The problem of PFI is that certain classes of models may adversely react to feature removal, and require retraining to properly evaluate impact of each feature. An alternative implementation simply removes a column (feature) from a dataset and retrains models. This approach has the added benefit of allowing a model to adapt to a new dataset, and not suffer from errors that it could otherwise remove with retraining. It is especially well suited to ML models that may learn complex relationships between features, compared to weak learners such as e.g., random forests. However, it requires training n^2 separate models, which may not be possible due to computational limitations.

6.5.1.2 PFI Case Study on Theta Logs

In (Isakov et al. 2020), we use PFI to evaluate the importance of features on I/O throughput prediction. This ranking allows I/O experts to design better logging tools and decrease the performance impact of these tools by removing superfluous features. The questions we ask are: which features are most important in understanding I/O throughput? How many features are needed to achieve different levels of prediction accuracy?

We apply PFI to two datasets: the first dataset (orange) consisting of post-processed features where many features were hand-selected. The second dataset (blue) contains all of the features collected by Darshan, many of which are timing features (e.g., measuring runtime, cumulative I/O read time, etc.). Figure 6.8 provides the reasoning for why timing features were removed from the I/O datasets. Here, models fed only the top n features (x-axis) are evaluated and their error is plotted on the y

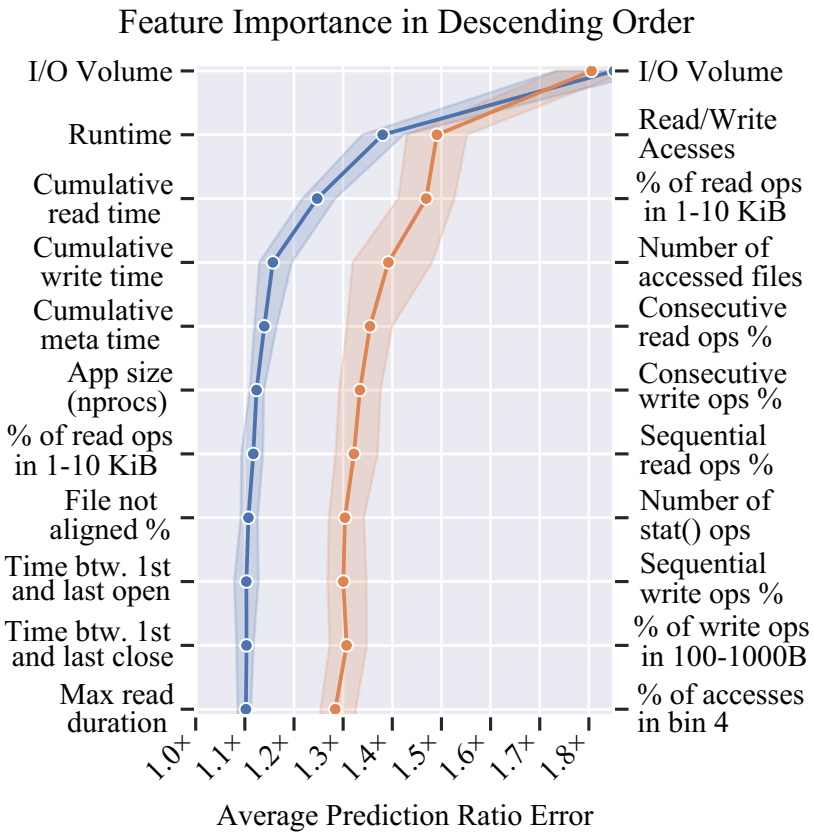


Figure 6.8: Darshan feature ranking according to Permutation Feature Importance (PFI).

axis. The top row shows models that are fed only the I/O volume feature, the second row shows models that are fed both I/O volume and either runtime or R/W ratio, respectively, etc. As more features are added, models asymptotically approach the error of models trained on the whole dataset with all the features.

When timing features are removed, PFI and I/O experts agree on which features should have the most impact on the model: more I/O volume allows the system to 'warm up' and achieve greater performance, read operations don't have coherency problems as writes do, and therefore improve throughput, small I/O operations are less efficient than large I/O operations, and have a significant impact on performance,

etc. Note that these features have the *largest* impact on I/O throughput, and not necessarily the most *positive* impact (as in the case of small I/O operations).

When timing features are added to the dataset (blue), PFI and I/O experts arrive at a significantly different results, as PFI seems to highly prioritize timing features over any actual I/O behavior. Further investigation shows that when exposed to I/O timing features, ML models exposed to timing features do not learn how the system responds to different workloads, but instead learn *how Darshan implements I/O throughput estimates*. Without ranking features, this would not be obvious, since the models are fed all the features (both workload behavior and timing) and are opaque in how they use them. Through feature ranking, insights into feature usage allow not only narrowing down the list of features to implement, but prevent degenerate

6.5.2 Local I/O Model Interpretation

PFI provides insight into which features are impactful across a whole dataset, and these relationships do not necessarily have to hold for every specific data sample. The relationships between a feature and target output may be nonlinear, as well as depend on the specific values of other inputs. This locality property must therefore be studied locally by observing small regions of similar application and system states.

In this section we focus on developing local models of I/O throughput and interpreting them. By local, we mean cluster-specific models, instead of the whole dataset. As seen in Figure 6.7, by using different ϵ values in DBSCAN clustering, we have selected several clustering granularities that result in splitting the whole dataset into different numbers of clusters. At each granularity, we split each cluster into a training and a test set, with a $70 - -30$ ratio. We train one XGBoost model per cluster and

predict I/O throughput on the cluster's test set. In Figure 6.7, we present a box plot of the concatenated errors of all clusters. As we can see, with higher granularity we achieve better I/O predictions. Note that since we are evaluating the models on a different set of data from what the model was trained on, we are confident that the model is not simply memorizing job-throughput pairs but has generalized well enough. However, notice that in Figure 6.7 the global model and models trained on coarse-grained clusters (red and green bars) achieve similar performance on both the training and test sets. On the other hand, when the dataset is split into hundreds of clusters (orange and blue bars), where each of the per-cluster models is trained on a small portion of the total data, the models that have excellent performance on the training set have a considerably worse performance on the test set. This is evidence of overfitting, pointing to the conclusion that for smaller clusters we should use simpler models or stronger regularization. Even with possible overfitting, however, on the test set these small-cluster models achieve considerably better accuracies compared with the global model. Therefore we seek to interpret these local models. For the interpretations, we use SHapley Additive exPlanations (SHAP) (Lundberg and Lee 2017; Lundberg et al. 2020), a game theoretic approach to interpreting black-box ML models. SHAP allows us to gain insight into the impact of each feature on a per-job level, providing us with information not only about which features are important but also about how they affect the prediction and how they react to other features.

To apply SHAP on these local models, we design an interactive HPC job analysis tool we call Gauge. It allows system administrators and I/O experts to select clusters from the HDBSCAN tree from Figure 6.3 and plot each cluster's information on a dashboard. In Figure 6.9, we present a screenshot of Gauge's dashboard, showing

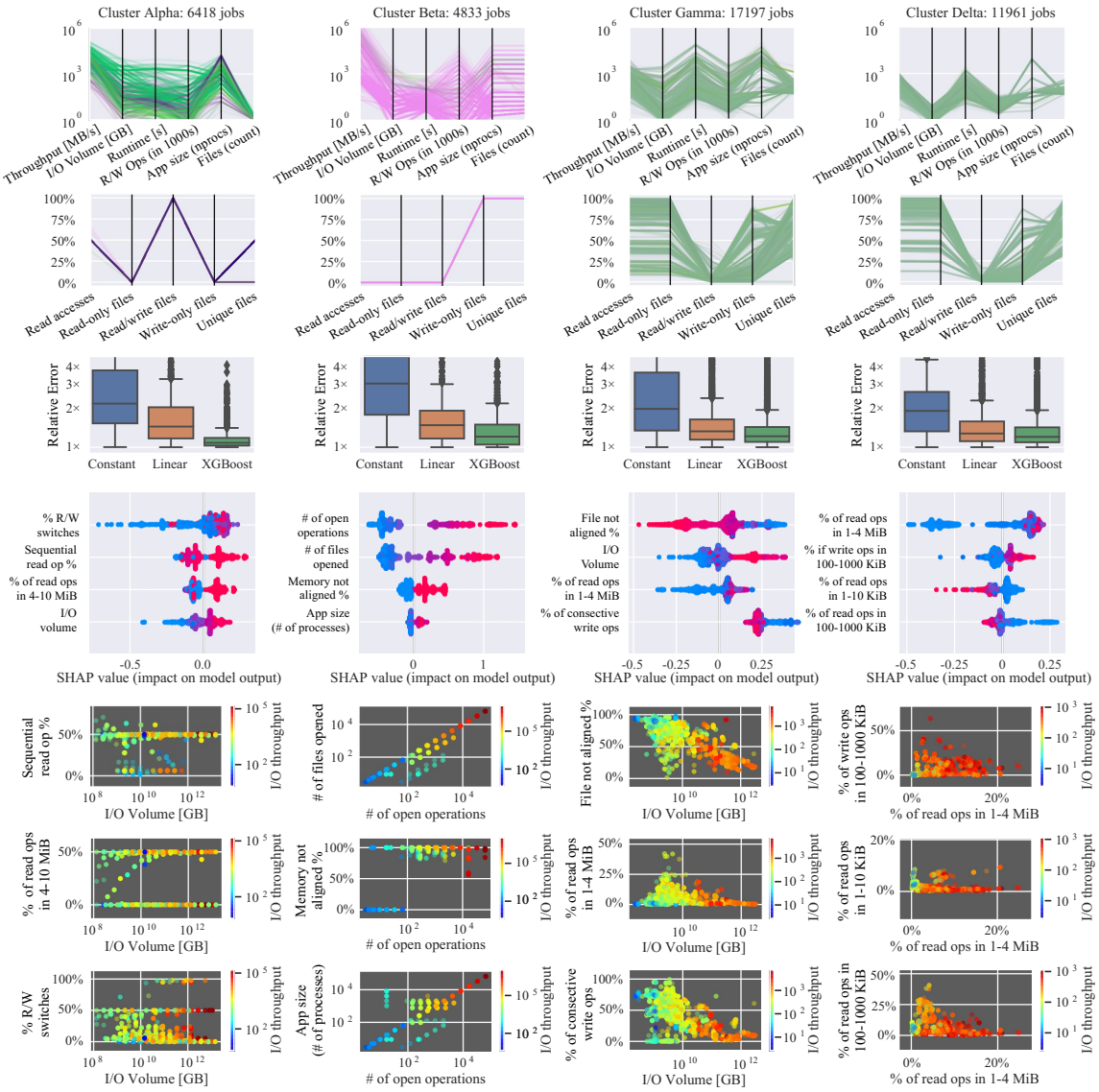


Figure 6.9: The Gauge Dashboard. The first and second rows show parallel coordinate plots of the logarithmic and percentage features for four different clusters. The third row shows the performance of three different ML models trained on each cluster. The fourth row shows the SHAP summary plot for each of the clusters. The last three rows show scatter plots of features selected by SHAP, with the color indicating the I/O throughput.

information about four clusters, with one cluster per column. Gauge succinctly shows the general information about each cluster in the first two rows.

6.5.3 Gauge Dashboard

The first row shows parallel plots of logarithmic features deemed most important by I/O experts. These plots allow the user to quickly gain insight into the cluster's I/O throughput volume, numbers of files, processes, and their relationships. Note the different units: we display throughput in MiB / s, volume in GiB, and the number of accesses in thousands, while numbers of processes and files are not modified. We selected MiB / s, GiB, and accesses in thousands so that the values can be plotted on the same logarithmic scale.

The second row shows another parallel plot, this time for ratio features. Note that jobs within the same cluster often have similar percentage features but differ in logarithmic ones. The reason is that jobs from, say, the same application may exhibit identical behavior but, because they were run with different inputs, show different IO volumes, throughputs, runtimes, and so forth.

The third row shows the error distribution of three ML models for predicting I/O throughput. The first is simply a median predictor—its prediction is a constant, selected as the median I/O throughput for the whole cluster. While trivial, this predictor is useful because it often outperforms other predictors once the cluster is very fine-grained, and it can serve as a baseline. The second classifier is a linear regression, and the third one is XGBoost. The linear regression performs well on medium-granularity clusters (with hundreds or thousands of jobs), where typically only one application's jobs exist in the cluster and XGBoost may overfit. For anything larger, XGBoost outperforms the constant and linear predictors, as can be seen in the third row of Figure 6.9.

The fourth row shows SHAP's summary plot for the ML models. These graphs

should be interpreted as follows, There exist four rows per plot (though this is parameterizable), and each row represents a feature. Red markers correspond to jobs with higher values and blue markers to lower values of that feature. The position of these markers indicates SHAP's predicted impact of the feature on the model's prediction. Markers on the right indicate that that job's feature has a positive impact on predicted I/O throughput, and markers on the left indicate that the impact is negative. As an example, the first column's "Data volume row" has red markers (high values) on the right, indicating that higher data volume results in higher throughput for that cluster. Alternatively, the "% R Ops 1–10 KiB" row in the fourth column has red markers on the left. The mean that larger numbers of reads in the [1–10] KiB range result in decreased I/O throughput. Note that the scale allows us to compare the impact of different features on I/O throughput.

Rows five, six, and seven show scatter plots of different features, colored with I/O throughput. The top feature according to SHAP is used for the x axis on all three scatter plots, while for the three y axes we use the second, third, and fourth most important features, respectively. These scatter plots are useful for getting a better understanding of any correlations or relationships between features. For example, looking at the bottom three plots in the second column (cluster Beta) we can immediately spot a linear relationship between the number of `open()` operations and the number of files accessed, as well as the number of processes. Gauge supports increasing the numbers of SHAP features and scatter plots, as well as using other types of plots such as correlation matrices showing feature correlations, but these are not shown because of space constraints.

Note that Gauge is an interactive tool. The user is expected to explore different clustering granularities (perhaps around a certain job or application) and different

cluster sizes, analyze ML models at those granularities, compare local and global patterns, and explore the (often nonlinear) relationship between the features using the scatter plots. Next, we provide a case study using Gauge to analyze jobs from ALCF's Theta supercomputer.

6.5.4 I/O Expert Case Study on Using Gauge

The conventional approach to I/O performance analysis in the context of user/facility interactions is to work collaboratively with individual users to address specific concerns or improve the productivity of high-profile applications. This hands-on focus has proven effective in numerous examples (Latham et al. 2012; Kodavasal et al. 2016; Srinivasan, Sudheer, and Namilae 2016) but fails to capitalize on the potential of guidance derived from broader contextual analysis:

- Does a given application conform to a contemporary I/O motif at *this facility* that is amenable to known optimizations?
- Would novel improvements to this application likely be applicable to other production applications?
- Is this I/O motif widespread enough to warrant strategic adjustments to provisioning or procurement?
- Beyond ad hoc user feedback, how can administrators allocate limited support resources for maximum impact?

Facility experts at Argonne Leadership Computing Facility (ALCF) used Gauge to explore workloads running on their systems, find workloads that do not obey conventional wisdom in terms of I/O performance., and diagnose sources of suboptimal I/O throughput (Rosario et al. 2020b). To illustrate the potential of Gauge in

finding previously unknown problems, and helping facility staff better understand their systems, this exploration was open-ended, i.e., the experts did not receive requests from HPC users to help optimize their codes.

As an illustrative example, experts flagged a cluster shown in Figure 6.4 as needing further analysis. The cluster consists of jobs from two applications, one in the quantum chromodynamics and one in the quantum materials domain. Both applications have similar I/O volumes and number of processes, and are largely performing read operations, but differ by several orders in magnitude for both runtime and I/O throughput. These differences are uncommon since other clusters of the same granularity have less variance across these two features.

Through HiPlot, the I/O expert gained further insight into the cluster by adding additional features not shown on the main columns of Gauge, shown in Figure 6.6. By coloring jobs by application and moving the I/O throughput axis (POSIX_RAW_agg_perf_by_slowest) to the left, several conclusions can be made: (1) applications transfer data in the 200 MiB – 2 GiB range, (2) orange jobs I/O throughput is 5 times that of blue jobs, and (3) the blue jobs open a larger number of files. On a suspicion, the I/O expert added the POSIX_RAW_FILE_ALIGNMENT column to HiPlot (right axis). This axis strongly separates the two applications, as one application has 256 KiB file alignments, while the other one has 1 MiB file alignments. Filesystems commonly have a single file alignment, and these different values hint at the possibility that these applications might be using files on different filesystems. Further inquiry showed that the slower application primarily accessed files on the user's home filesystems, while the faster application used files on the Lustre distributed filesystem. The home filesystem is meant for experimentation, software development and general-purpose use, but cannot offer the needed I/O throughput

necessary for high-performance computing, so HPC users are typically advised to move their data to Lustre. Despite ALCF having mandatory training sessions, and the scientists being briefed on using the HPC system, scientists typically develop their applications on their local machines, and can easily make such a mistake.

There are several systemic reasons why such mistakes are not diagnosed and fixed. First, users of many newer applications do not have good estimates of what the expected runtime is, and may find significantly degraded performance acceptable. Second, when compute hours are plentiful, users may attempt to scale their applications to larger numbers of nodes instead of optimizing their applications. Third, no tooling or system utilization dashboard exists that may inform users that their applications are misappropriating HPC system resources, and it may be difficult for the scientist to debug this issue without the manual involvement of an I/O expert. Gauge offers a more efficient and scalable method for helping HPC system users to understand and accelerate their workloads, and for helping facility staff extract more performance out of their systems. Furthermore, Gauge provides rapports that can be used by the facility staff to visualize and substantiate their insights, improving communication between HPC users and HPC staff.

6.6 Results and Insights

6.6.1 Application to System Sensors

Model interpretation techniques such as SHAP and permutation feature importance are shown to be able to evaluate which logging or system sensors are useful for predicting job behavior. Since both of these techniques are automated, they open the possibility of developing sensors and logging utilities without human supervision. Additionally, by pruning unused features, workload and system logs can be compressed, allowing for higher-frequency monitoring or additional, useful features to be collected.

6.6.2 Application to Decision-Making Systems

Clustering logs may accelerate workload analysis but still requires human intervention at this moment. While SHAP-based local model interpretation can offer some insights into workload or system bottlenecks, connecting these conclusions to actuator control is still under investigation.

Another issue that stands in the deployment of models presented in this chapter is the fact that the logging and decision-making are performed post hoc. Since the decision-making system has only limited information about a given scheduled job (user, program, input dataset, etc.) it can only assume what the scheduled job's logs will contain after the job is completed. If the job belongs to a common application, predicting job features may not be a problem, but this approach does not work for novel, out-of-distribution jobs.

A better approach would actively monitor workloads and the system, and dynamically adapt the system when the scheduling-time predictions are deemed unsatisfactory.

6.6.3 Application to System Actuators

Scheduling-time actuators are largely limited to configuring the system before a job is run, optimizing allocation of shared resources between workloads, and isolation of misbehaving or possibly malicious jobs. While these actuators are acceptable when the decision-making system makes decisions at scheduling-time, any runtime decision-making system would not be able to utilize existing system actuators. In that case, it may be beneficial if the system is extended with new actuators that allow for e.g., moving jobs between nodes during job execution, dynamically adapting the I/O subsystem, or physically separating workloads that do not interact well over shared resources.

Chapter 7

RUNTIME MODELING AND ADAPTATION

Adaptive systems can broadly be classified into statically-adaptive and dynamically-adaptive systems. This chapter explores dynamically-adaptive systems: systems which can exercise their adaptive ability before as well as during workload execution. In Section 7.1 I present a case study in deploying ML-based scheduling systems and showcase the need for dynamic adaptation. In Section 7.2 I analyze why ML-based scheduling systems fail in production and introduce a method that quantizes model error by source.

In Section 7.3 I present several obstacles to deploying machine learning models on real systems and outline solutions for solving these obstacles and evaluate the advantages of dynamic versus static adaptation.

7.1 Limits of Scheduling-Time Adaptation

The experimental setup in this chapter closely follows the setup from Chapter 4.1, with a couple of significant differences. First, all analysis in the previous chapter was performed post-hoc, with the goal of understanding why workloads are behaving on the systems as they do, and what insight into workloads and systems is necessary to facilitate modeling. The resulting models are never directly applied, but are only dissected to better understand the domain, i.e., the ML models built in the previous chapter are not the goal of ML modeling. In this chapter, similar HPC I/O prediction models are applied to queued jobs, with the goal of understanding how to place them, how they will interact with other jobs, how sensitive to contention are they, etc.

7.1.1 Deployment Scenario

The experimental setup includes a computing system (a leadership-class supercomputer in our case), a job scheduler, and a queue of jobs. The scheduler is tasked with (1) optimizing system resource utilization, while (2) maintaining fairness between scheduled jobs. Optimizing resource utilization is the task of utilizing as large percentage of available compute resources as possible, where compute resources Here, the smallest compute resource allocation is a node - a (potentially multi-socket), multi-core machine with a separate memory address space and a separate address on the network. This is a significantly coarser allocation granularity than e.g., on OS or cloud schedulers, where individual cores may be divvied up. Node-level allocations are easier to model as node-level compute, memory and networking belong to a single

job. Therefore, any resource contention in the system is isolated to the networking and storage subsystems.

The scheduler uses an ML model of the system to predict workload behavior once executed. The model may be built to predict e.g., I/O throughput, job runtime, runtime variability, performance impact on other jobs running on the system, etc. In this section, we will focus on I/O throughput predictions, and will explore impact and variability in Chapter 7. Once the model has made a prediction for all candidate queued jobs, the scheduler can use the model output to decide in what order to execute the jobs, how many resources to dedicate to each job, whether to potentially isolate jobs predicted to misbehave or separate pairs of incompatible jobs, etc.

7.1.2 Experimental Setup

When two I/O-heavy jobs are scheduled at the same time on the system, they may be I/O bound, and spend significant amounts of time idling and waiting for responses from the I/O subsystem. Since supercomputing resources are typically measured in node-hours, HPC system users may inefficiently spend allocated node-hours, as the node CPUs will have low CPU utilization. The goal of the HPC job scheduler is to accurately estimate job I/O throughput in order to schedule I/O-heavy jobs separate from each other, in order to allow each I/O-heavy job to fully saturate available I/O bandwidth. By pairing I/O-heavy jobs with more lightweight jobs, good utilization of both compute and storage resources may be achieved.

We will observe I/O throughput prediction models fed the same Darshan features shown in Table 4.1. While such features are not available until after a job has completed, they serve as a good proxy for other features a scheduler may possess. We

will use the same model architectures from the previous chapter, specifically XGBoost with fine-tuned hyperparameters.

In typical machine learning experimental setups, cross-validation is used to create training, validation and test sets. Models are trained on training sets, their performance is evaluated on the validation tests, after which model hyperparameters can be updated, and training can be repeated. Once the training and hyperparameter tuning is complete, the model is evaluated on the test set. By evaluating the model on the test set only once, information leakage from the test set to model hyperparameters can be prevented.

To simulate a realistic environment the scheduler may exist in, we use a modified version of cross-validation. While the HPC datasets we possess contain jobs ran throughout 2017 to 2020, we wish to simulate a model deployment environment with freshly collected data. To do so, we select a timestamp in the 2017 to 2020 range, and split the existing dataset into two portions, one before the timestamp and one after. First, the model will get trained, validated and tested on the pre-timestamp dataset. Once some fine-tuned model has received approval, it may be deployed, at which point it is evaluated on the post-timestamp dataset. We use the post-timestamp dataset as a 'deployment' test set, since if scheduler's ML model is only exposed to the pre-timestamp dataset, and has no knowledge of upcoming test set jobs. Therefore, there are effectively two different test sets the model is evaluated on: first is a pre-timestamp 'original' test set, which is a random subset of the whole pre-timestamp dataset, and the second is a 'deployment' test set, which is the whole post-timestamp dataset.

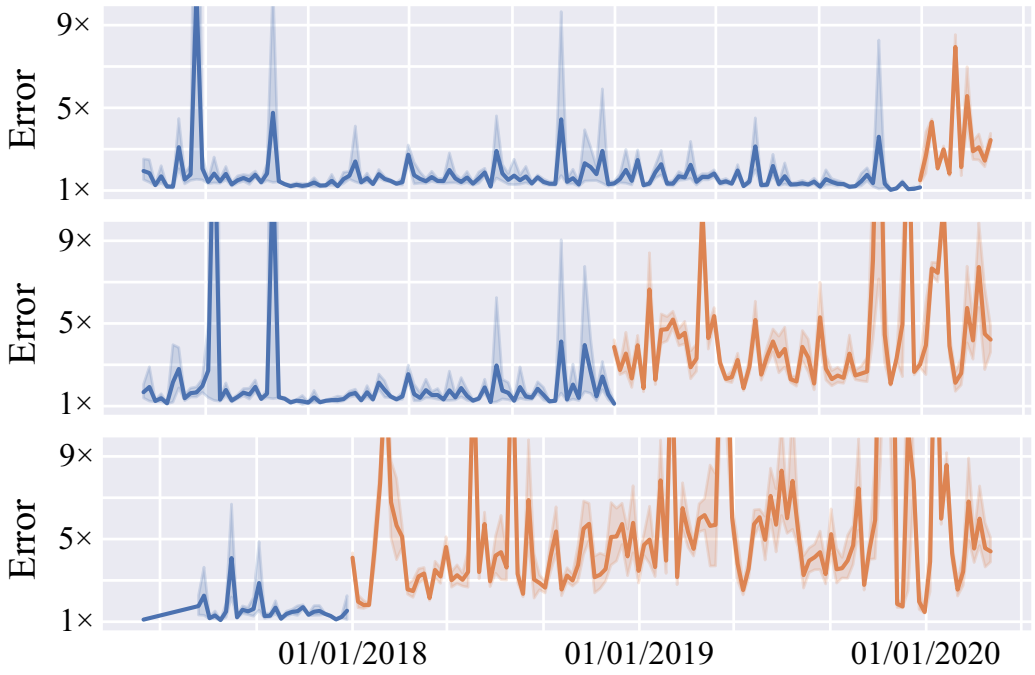


Figure 7.1: I/O throughput prediction error for three different dates used for the training / test set splits.

7.1.3 Real-world Deployment Results

To evaluate how scheduling ML models behave, we initially set the timestamp to January 1st, 2020, and split the dataset across it. We train and evaluate a model on HPC jobs collected in between January 2017 and December 2019, with test set errors shown in the top row of Figure 7.1 (blue line). We then evaluate the same model on the remainder of the data (the deployment test set), from January 2020 to December 2020 (orange line).

From the perspective of a scheduler programmer or ML practitioner, the model performs well on the original test set, and receives approval to be used in production. Since the test set is randomly drawn from the pre-timestamp dataset, and is only evaluated once, the practitioner is confident that its information did not 'leak', and

that it is representative of the deployment environment. However, immediately after deployment, the model's error grows an order of magnitude, from 10%-30% error on average, to 200%-500% on average. This is unexpected, and may be caused by two different effects: (1) either the job distribution changed significantly after model deployment, or (2) the original test set is not representative of the deployment environment.

To test the first hypothesis, we repeat the experiment two additional times, this time setting the timestamp to January 1st of 2019 and 2018 (rows two and three in Figure 7.1). The effect persists when different timestamps are used, pointing to the fact that models do not generalize to deployment workloads. In the next section, I explore the sources of the sources of this discrepancy.

7.1.4 Modeling Assumptions Leading to Poor Prediction Accuracy

To model system behavior, we adopt the system modeling formulation from Chapter 3, expressing the relationship between an HPC job j and its I/O throughput on the system $\phi(j)$ as:

$$\phi(j) = f(j,\ \zeta,\ \omega) \tag{7.1}$$

Here, j represents HPC job behavior (e.g., I/O volume and access patterns, distribution of POSIX operations, etc.), ζ represents system state (e.g., file system health, system configuration, node availability, etc.) and system behavior (e.g., the behavior of other applications co-located with the modeled application during its run, contention from resource sharing, etc.) *at a given time.* ω represents the randomness acting on the system. The system ζ can be further decomposed as:

$$\zeta = \zeta_g(t) + \zeta_l(t,\ j) \tag{7.2}$$

The component $\zeta_g(t)$ represents the *global system impact* on all jobs running on the system (e.g., a service degradation that equally impacts all jobs) and is only a function of time t. The component $\zeta_l(t,\ j)$ represents the *local system impact* on the I/O throughput of job j caused by resource contention and interactions with other jobs running on the system. Contrary to the $\zeta_g(t)$ component, $\zeta_l(t,\ j)$ is job-specific and depends on the behavior of the current set of applications running on the system and their location relative to j, the sensitivity of j to resource contention and noise, etc. Without loss of generality, the I/O throughput from Equation 7.1 can be expressed as:

$$\begin{aligned}\phi(j) &= f(j,\ \zeta_g(t),\ \zeta_l(t,j),\ \omega) \\ &= f_a(j) + f_g(j,\ \zeta_g(t)) + f_l(j,\ \zeta_l(t,\ j)) + f_n(j,\ \zeta,\ \omega)\end{aligned} \tag{7.3}$$

The task of modeling a system's I/O throughput involves predicting the behavior of the system when tasked with executing a job from some application on some data. Modeling I/O throughput requires modeling both the HPC system and the jobs running on it. Machine learning models used in this work attempt to learn the true function ϕ by mapping observable features of the job j and the system ζ to measured I/O throughputs $\phi(j)$. A model $m(j_o,\ \zeta_o)$ is tasked with predicting throughput $\phi(j)$, where $j_o \subseteq j$ and $\zeta_o \subseteq \zeta$ are the observable job and system features.

When designing ML models, the choice of model architecture and model inputs is based on implicit assumptions about the process that generates the data. When incorrect assumptions are made about the domain, the model will suffer from errors that cannot be fixed within that modeling framework, e.g., through hyperparameter tuning or further data collection. We investigate four common assumptions about the HPC domain, shown by the branches in Figure 7.2.

All data is in-distribution: a common assumption that ML practitioners make is that all model errors are the product of insufficiently trained models, inadequate

model architectures, or missing discriminative *features*. However, some jobs in the dataset may be *Out of Distribution (OoD)*, that is, they may be collected at a different time or environment, or through a different process. The model may underperform on OoD jobs due to the lack of similar jobs in the training set and not due to lack of insight (features) into the job. The cause of the problem is *epistemic uncertainty (EU)* - the model suffers from *reducible uncertainty*, i.e., lack of knowledge, since a broader training set would make the OoD jobs in-distribution (ID). In the HPC domain, epistemic uncertainty is present in cases of rarely ran or novel jobs or uncommon system states. Without considering the possibility that a portion of the error is a product of epistemic uncertainty, practitioners may put effort into tuning models instead of collecting more underrepresented jobs. Referring to Equation 7.1, this assumption may be expressed as: deployment time j_d and ζ_d are drawn from a different distribution from training time j_t and ζ_t.

Noise is absent: all systems have some inherent noise that cannot be modeled and will impact predictions. *Aleatory uncertainty* (AU) refers to *irreducible uncertainty* which stems from inherent noise or lack of insight into jobs on the system. Modeling errors due to aleatory uncertainty are different from epistemic uncertainty because collecting more jobs may not reduce AU, and these errors may be fundamentally unfixable. Understanding and characterizing a system's inherent I/O noise is necessary to quantify ML model uncertainty, and because the amount of noise in the data has a strong effect on the optimal choice of ML model. HPC I/O domain experts note that certain systems do have significantly higher or lower I/O noise (Wan et al. 2017; Xie et al. 2017), but I/O modeling works rarely attempt to quantify ML model uncertainty (Madireddy et al. 2018a). The assumption that noise is not present in the dataset can be expressed as follows: The practitioner assumes that the data-generating

process ϕ has the form of $\phi(j) = f(j, \zeta)$ instead of $\phi(j) = f(j, \zeta, \omega)$, i.e., that the inherent noise impact is zero: $f_n(j, \zeta, \omega) = 0$.

Sampling is independent: running a job on a system can be viewed as sampling the combination of application behavior and system state and measuring I/O throughput. Most I/O modeling works implicitly assume that multiple samples taken at the same time are independent of each other. The system is modeled as equally affecting all jobs running on it, that is, the placement of different jobs on nodes, the interactions between neighboring jobs, network contention, etc. *do not affect the job*. This assumption can then be expressed as: the process has the form of $\phi(j) = f(j, \zeta_g(t))$, not $\phi(j) = f(j, \zeta, \omega)$ i.e., that the resource contention impact is zero: $f_l(j,\ \zeta_l(t,\ j)) = 0$.

Process is stationary: a common assumption ML practitioners make is that the data-generating process is stationary, and that the same job ran at different times achieves the same I/O throughput. As hardware fails, as new nodes are provisioned, and shared libraries get updates, the system evolves over time. The stationarity assumption is therefore incorrect, and ignoring it by e.g., not exposing the ML model to *when* a job is run may cause hard-to-diagnose errors. This assumption implies sampling independence and absence of noise, and can be expressed using the system modeling formulation as: $\phi(j) = f(j)$ and $f_g(j, \zeta_g(t)) = 0$.

7.2 Understanding Sources of Model Error

During the development of an adaptive general-purpose computing system, hundreds of different architectural designs may be prototyped, iterated on, and evaluated. As hardware is typically built to be parametrizable, and differently-parametrized designs may have different behavior, separate ML models of these systems must be developed and evaluated. While ML configurations may be automatically explored through Hyper-Parameter Optimization (HPO) and Neural Architecture Searches (NAS), when these approaches converge, ML practitioners may need to manually diagnose systemic errors. Due to the large number of complex systems in existence, manually building and debugging ML models of these systems is a laborious and unscalable task.

In this section, we seek a systematic, automated, and generalizable method for diagnosing and measuring sources of model error. The goal of such a method is to automatically detect when models require further tuning, when additional system sensors are necessary to achieve good control, when the system is too noisy to be accurately controlled, when workloads are novel and the system cannot hope to achieve good control due to lack of training data, etc. With automated diagnostics, ML practitioners do not need to be a part of the design iteration loop, and allow rapid prototyping of adaptive systems.

7.2.1 Conventional Methods for Diagnosing Model Errors

When dealing with an underperforming model, ML practitioners may hand-craft, grid search, or evolve new model architectures, they may attempt to gain more data

samples, or add more information (features) per sample. If this does not work, they may manually analyze mispredicted samples, potentially find mislabeled samples in the training or test sets, try and understand training dynamics, e.g., by observing gradient magnitudes in neural networks, attempt to classify the problem as e.g., underfitting or overfitting, possibly perform predictions on the test set themselves in order to establish a human baseline, etc. All of these methods are applied *ad hoc*, without a defined order or unifying framework for diagnosing errors. ML practitioners typically informally develop experience and 'know-how' in dealing with certain domains, e.g., image processing or tabular data.

This set of approaches may work on difficult to collect datasets such as e.g., X ray scans that do not change often. This dissertation targets system modeling, where the object being modeled (e.g., an HPC system or a CPU microarchitecture) may change daily. Therefore, in this domain, experts may need to repeatedly re-apply a set of 'tricks', and due to the weakly-defined error diagnosis procedure, may not be able to automate them.

7.2.2 Classifying I/O throughput prediction errors

No matter the problem to which machine learning is applied, a systematic characterization of the sources of errors is crucial to improve model accuracy. While there is no substitute for 'looking at the data' to understand the root cause of the problem, this approach does not scale for large datasets. We seek a systematic way to understand the barriers to greater accuracy and improve ML models applied to systems data.

While the work presented here can be generalized past just I/O to e.g., compute or

network modeling, we study I/O because I/O bottlenecks are more difficult to diagnose than compute bottlenecks, and because I/O has a coarser temporal granularity allowing software to observe I/O subsystems without the need for e.g., hardware performance counters or binary instrumentation. The key questions we ask in this work are: What are the impediments to the successful application of learning algorithms in understanding I/O? Should ML practitioners focus on acquiring more data on HPC applications or the HPC system? How much of the error stems from poor ML model architectures? How much of the error can be attributed to the dynamic nature of the system and the interactions between concurrent jobs? How much of the performance variation is caused by the system? What fraction of jobs exhibit truly novel I/O behavior compared to jobs observed thus far? At what point are the applications *too novel*, so much so that users should no longer trust the predictions of the I/O model? We now describe five error classes and dive deeper into error attribution in Sections 7.2.4, 7.2.5, 7.2.7 and 7.2.6.

The lack of application and system observability, the interaction between running jobs, the inherent system noise, and the novel or rare applications prevent ML models from fully capturing system behavior, causing errors. We define the I/O throughput prediction error of a model m in a job j as:

$$e(j) = \phi(j, \zeta, \omega) - m(j_o, \zeta_o) \tag{7.4}$$

Following the $\phi(j)$ terms from Eq. 7.3 and including the out-of-distribution error, the error can be broken down as follows:

$$e(j) = e_{app} + e_{system} + e_{ood} + e_{contention} + e_{noise} \tag{7.5}$$

Here, the application modeling error e_{app} is caused by a poor model fit of application behavior ($f_a(j)$ component), the global system error e_{system} is caused by poor

predictions of global system impact ($f_g(j, \zeta_g(t))$ component), the out-of-distribution error e_{ood} is caused by weak model generalization on novel applications or system states, the contention error $e_{contention}$ is caused by poor predictions of job interactions ($f_l(j, \zeta_l(t, j))$ component), and the noise error e_{noise} is caused by the inability of any model to predict inherent noise ($f_n(j, \zeta, \omega)$ component). These five classes of errors are shown as leaf nodes at the bottom of Figure 7.2. While attributing cumulative job error to each class may be difficult on a per-job basis, we will show that estimating each component across a whole dataset is possible.

7.2.3 I/O Model Error Taxonomy and Litmus Tests

We adopt the term litmus test to mean a test that evaluates the presence, amount, or ratio of a certain quantity. In the following sections, we introduce a four litmus tests that split the error from Equation 7.5 into five separate classes. The error classes in Equation 7.5 must be estimated in the order shown in the bottom row of Figure 7.2 due to the specifics of individual litmus tests. For example, before the effect of aleatory and epistemic uncertainty can be separated, a good model must be found (Egele et al. 2021). Similarly, before global and local system modeling errors can be separated, OoD jobs must be identified.

Application modeling errors: ML models can have varying expressivity and may not always have the correct structure or enough parameters to fit the available data. Models whose structure or training prevents them from learning the shape of the data-generating process are said to suffer from *approximation errors*. Approximation errors cannot be classified as epistemic or aleatory in nature because no new features or jobs are necessary to remove this error. To estimate AU and EU in the dataset,

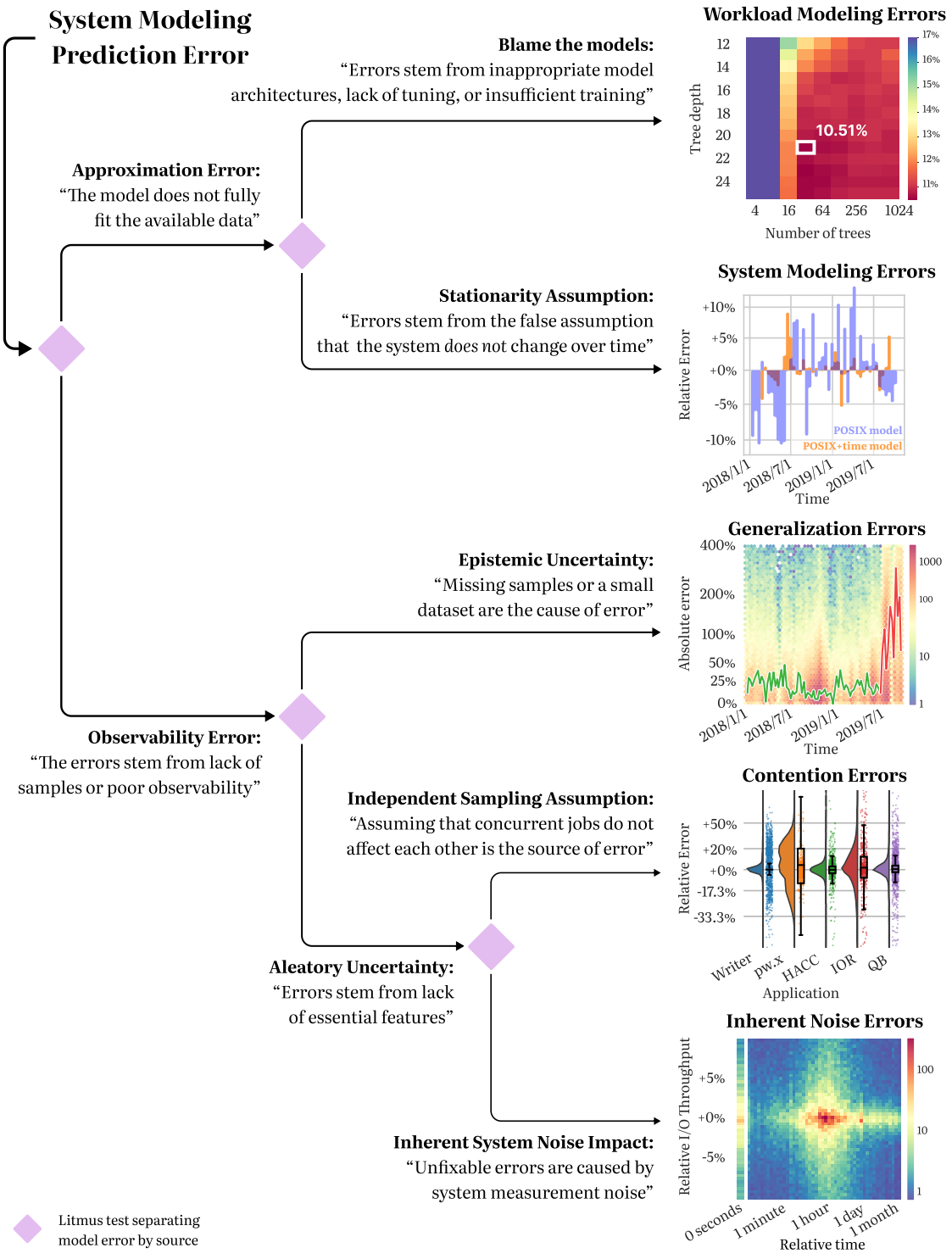


Figure 7.2: Taxonomy of I/O throughput modeling errors, with examples of the effects of each error class shown in the column on the right.

methods such as AutoDEUQ (Egele et al. 2021) first require that an appropriate model architecture is found and trained, placing approximation errors as the first branch of the taxonomy.

Approximation errors are further divided into *application* and *system modeling errors*. Application modeling errors are caused by poor predictions of application behavior which can be fixed through hyperparameter searches or better model architectures. The first column of Figure 7.2 illustrates the impact of application modeling errors with an example hyperparameter search over two XGBoost parameters on the Theta dataset (introduced in the next section). The best configuration found by the grid search has 32 trees with a depth of 21, while the default XGBoost configuration uses 100 trees of depth 6.

System modeling errors: system behavior changes over time due to transient or long-term changes such as file system metadata issues, failing components, new provisions, etc. (Lockwood et al. 2017). A model that is only aware of application behavior, but not of system state implicitly assumes that the process is stationary. It will be forced to learn the *average* system response to I/O patterns, and will suffer greater prediction errors during periods when system behavior is perturbed. *System modeling errors* occur due to poor (or complete lack of) modeling of the global system component $\zeta_g(t)$. To illustrate this class of errors, the second experiment in Figure 7.2 shows the per-week average error of two models trained to predict job I/O throughput. The blue model can be written as $m(j_o)$, i.e., it is only exposed to observable application behavior j_o. The orange model can be written as $m(j_o, t)$, i.e., it also knows the *job start time* t. During service degradations, the blue model has long periods of biased errors while the orange model does not, since it knows when the degradations happen.

Generalization errors: ML models should perform well on data drawn from the same distribution from which their training set was collected. When exposed to samples highly dissimilar from their training set, the same models tend to make mispredictions. These samples are called 'out-of-distribution' (OoD) because they come from new, shifted, distributions, or the training set does not have full coverage of the sample space. While models that generalize (perform well on OoD data) may exist, mispredictions on OoD samples are not always the fault of the model, and in those cases the only recourse is to (1) detect and exclude samples suspected as out-of-distribution, (2) seek an expanded training set covering those regions, or (3) apply domain-specific knowledge. In order not to pollute other classes of errors, samples that show high epistemic uncertainty must be detected and their error counted towards *generalization errors* before other errors are estimated. As an example, the third column of Figure 7.2 shows model error before (green) and after (red) deployment, with the error significantly rising when the model is evaluated on data collected outside the training time span.

Contention and resource sharing errors: a diverse and variable number of applications compete for compute, networking, and I/O bandwidth on HPC systems and interact with each other through these shared resources (B. Li et al. 2019; X. Yang et al. 2016). Although the global system state will impact all jobs equally, the impact of resource sharing is specific to pairs of jobs that are interacting and is harder to observe and model. Prediction errors that occur due to lack of visibility into job interactions are called *contention errors* and are shown in the fourth column of Figure 7.2. Here, the I/O throughputs of a number of identical runs (same code and data) of different applications illustrate that some applications are more sensitive to contention than others, even when accounting for global system state.

Inherent noise errors: while hard to measure, contention and resource sharing errors can be potentially removed through greater insight into the system and workloads. What fundamentally cannot be removed are *inherent noise errors*: errors due to random behavior by the system (e.g., dropped packets, randomness introduced through scheduling, etc.). Inherent noise is problematic both because ML models are bound to make errors on samples affected by noise and because noisy samples may impede model training. The fifth column of Figure 7.2 shows the I/O throughput and start time differences between pairs of identical jobs. The leftmost column contains identical jobs that ran at exactly the same time, which often experience 5% or more difference in I/O throughput.

7.2.4 Application modeling errors

When an ML practitioner is tasked with a classification or a regression problem, the first model they evaluate will likely under-perform on the task, due to e.g., inadequate data preprocessing, architecture, or hyperparameters. Therefore, the model will suffer from *approximation errors*, which can be removed by tuning the model hyperparameters or finding more appropriate domain-specific ML model architectures. Since the choice of model architecture and parameters typically has a dominant effect on model error, approximation errors must be resolved before more subtle classes of errors become a limiting factor in improving model performance.

Approximation errors can be split into errors caused by poor modeling of the available data (i.e., applications), and into errors caused by implicit assumptions about the domain (e.g., that I/O behavior of a system does not change over time). In

this section we analyze application modeling errors, and in Section 7.2.5 we analyze system modeling errors.

This section asks the following questions: do I/O models build faithful representations of application behavior? What are the limits of I/O application modeling? In practice, do I/O models faithfully learn application behavior? Can I/O application modeling benefit from extra hyperparameter fine-tuning or new application features?

7.2.4.1 Estimating limits of application modeling

Here we develop an application modeling error litmus test which separates the application modeling error e_{app} from the other four error classes in Equation 7.5. To do so, we seek a 'golden model' (GM) that predicts I/O throughput as accurately as possible given the observable application behavior. Application modeling error of a practical ML model is then estimated by comparing its error rate with that of a golden model.

To build this 'golden model', we rely on a property of synthetic datasets where the data-generating process can be freely and repeatedly sampled. When analyzing HPC logs, it is common to see records of the same application ran multiple times on the same data, or data of the same format. For example, system benchmarks such as IOR (Shan and Shalf 2007) may be run periodically to evaluate file system health and overall performance. We call these sets of repeated jobs *'duplicate jobs'*. Pairs of jobs are duplicates if they belong to the same application and all of their *observable* application features are identical, typically because the application was ran with the same configuration and input data. Because jobs from the same set of duplicates appear identical to an ML model, the model cannot distinguish between them. Given

a training set that only contains sets of duplicate jobs, the highest possible accuracy can be achieved by mapping jobs from each individual set of duplicates to the set's mean I/O throughput. A model that does not learn to predict a set's mean value is said to suffer from application-modeling error.

By restricting the training set to only sets of duplicates, a golden model with a median absolute error e^g can be built for which $e^g_{app} = 0$. This golden model performs only memorization and does not generalize at all, but is nonetheless useful for comparison against real ML models. Any practical model with a median error e^p can then learn its application modeling error e^p_{app} on the restricted training set by comparing against the golden model e^g as $e^p_{app} = e^p - e^g$. Since duplicate sets can have as few as two jobs, I/O throughput estimates for duplicate sets are biased, and the golden model (GM) may appear to perform better on small sets than on large sets. By applying Bessel's correction (Bishop 2006), this effect is mitigated, and the litmus test is administered as:

Application modeling error litmus test:

1. Find sets of duplicate jobs. For each set:
    1.1. Calculate the set's mean I/O throughput;
    1.2. Apply Bessel's correction to mean;
    1.3. Use mean as golden model prediction;
    1.4. Calculate per-set mean GM absolute error;
2. Calculate median of real and GM mean set errors;
3. Calc. model's e_{app} as difference between the two;

Assuming that duplicate jobs are drawn from the same distribution of applications

as the rest of the dataset, the golden model median absolute error represents the lower bound on median absolute error a model can achieve on the whole dataset. Note that different applications may have different distributions of duplicate I/O throughputs, as shown in the fourth column of Figure 7.2. For this litmus test to be accurate, a large sample of applications representative of the HPC system workload must be acquired. When applied to Theta, 19010 duplicates (23.5% of the dataset) over 3509 sets show a median absolute error of 10.01%. Cori has 504920 duplicates (54%) in 77390 sets with a median absolute error of 14.15%. If the litmus test is applied correctly, practical ML models may approach the golden model's error but cannot surpass it.

7.2.4.2 Minimizing application modeling error

The next question is whether ML models can practically reach the error lower bound estimate e^g. Several I/O modeling works have explored different types of ML models: linear regression (Isakov et al. 2020), decision trees (Tuncer et al. 2017), gradient boosting machines (Isakov et al. 2020; Tuncer et al. 2017; Xie et al. 2021), Gaussian processes (Madireddy et al. 2018b), neural networks (Madireddy et al. 2019b), etc. Here, we explore two types of models: XGBoost (Chen and Guestrin 2016), an implementation of gradient boosting machines, and feedforward neural networks. These model types are chosen for their accuracy and previous success in I/O modeling.

Neither type of model achieves ideal performance 'out of the box'. XGBoost model performance can be improved through hyperparameter tuning, e.g., by exploring different (1) numbers of decision trees, (2) their depth, (3) the features each tree is exposed to, and (4) part of the dataset each tree is exposed to. Neural networks are more complex, since they require tuning hyperparameters (learning rate, weight decay,

dropout, etc.), while also exploring different architectures (number of layers, their size, type, and connectivity). In the case of XGBoost, we exhaustively explore four hyperparameters listed above, for a total of 8046 XGBoost models. In the case of neural networks, exhaustive exploration is not feasible due to state space explosion, so we use AgEBO (Egele et al. 2020), a Network Architecture Search (NAS) method that trains populations of neural networks and updates each subsequent generation's hyperparameters and architectures through Bayesian logic.

The leftmost column of Figure 7.2 shows a heatmap of an XGBoost exhaustive search over two parameters on the Theta dataset, with the other two parameters (% of columns and rows revealed to the trees) selected from the best possible result found. The best performing model has an error of 10.51% - close to the predicted bound of 10.01%. The Cori search arrives at a similar configuration with an error of 14.92%.

In the case of neural networks, Figure 7.3 shows a scatter plot of test set errors of 10 generations of neural networks on the Cori system, with 30 networks per generation. The networks are evolved using a separate validation test to prevent leakage of the test set into the model parameters. Networks approach the estimated error limit, and the best result achieves a median absolute error of 14.3%. After extensive tuning both neural networks and XGBoost models asymptotically approach the estimated limit in model accuracy. Despite the 300 trained neural networks, NAS does little to improve models, since only 6 out of 300 different models improve on previous results (gold stars in Figure 7.3). This suggests that both types of ML models are impeded by the same barrier and that the architecture and the tuning of models are not the fundamental issue in achieving better accuracy, i.e., that the source of error lies elsewhere.

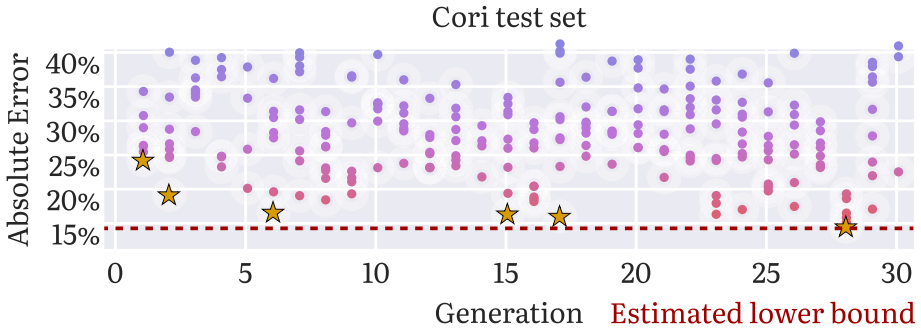


Figure 7.3: Results of the Neural Architecture Search (NAS), with the estimated error lower bound highlighted in red.

7.2.4.3 Increasing visibility into applications

While hyperparameter and architecture searches approach but do not surpass the litmus test's estimated lower bound on error, this is not conclusive evidence that all application modeling error has been removed and that error stems from other sources. Possibly, there exist missing application features that might further reduce errors. We explore two such sets of features: MPI-IO logs and Cobalt scheduler logs.

Figure 7.4 shows the absolute error distribution of hyperparameter-tuned models trained on three Theta datasets: POSIX, POSIX + MPI-IO, and POSIX + Cobalt (Cori excluded because of the lack of Cobalt logs). None of the dataset enrichments help reduce error, corroborating the conclusion that poor application modeling is not a source of error for these models, and further insight into applications will not help. Note that this absence of evidence does not imply evidence of absence, i.e., it does not prove that there exist no features that may help improve predictions. However, this experiment does present a best-effort attempt at exposing novel features, and the model's predictions stay within predicted limits.

Adding Cobalt logs *does* reduce the error on the training set, and ablation studies show that the job start and end time features are the cause. Once timing features

are present in the dataset, no two jobs are duplicates due to small timing variations. While previously the ML model was not able to overfit the dataset due to the existence of duplicates, this is no longer the case, and the ML model can differentiate and memorize each individual sample. In (Isakov et al. 2020) authors remove timing features for a similar reason: ML models can learn Darshan's implementation of I/O throughput calculation and make good predictions without observing job behavior.

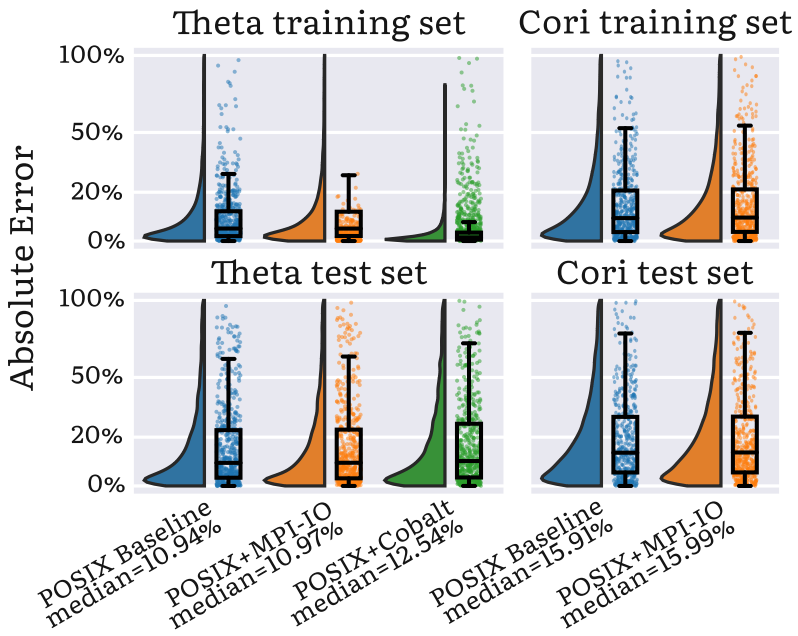


Figure 7.4: Error distribution of models trained on POSIX, POSIX + MPI-IO, and POSIX + Cobalt feature sets.

7.2.5 Global system modeling errors

The second part of the approximation error in the taxonomy is the global system modeling error. This error refers to I/O climate and I/O weather effects (Lockwood et al. 2017) that affect all jobs running on the system, and corresponds to the second component in Equation 7.3. While global and local system impact on job performance

have complex and overlapping effects, factorizing system impact into impact applied to all jobs versus the impact that is dependent on pairs of concurrent jobs is useful for modeling purposes. The main difference between the two is that modeling local system impact requires modeling relationships between all pairs of concurrent jobs, while modeling global system impacts requires modeling only a single but pervasive influence. In other words, global system impact modeling is insensitive to the number of concurrent jobs running on the system, and can be seen as a form of lossy compression of system state and contention impact on jobs.

We now ask: How does I/O contention impact job I/O throughput prediction? What are the limits of global system modeling? Can I/O models approach this limit? What I/O subsystem features can help improve I/O throughput predictions?

7.2.5.1 Estimating limits of global system modeling

Global system impact $\zeta_g(t)$ on job j from Equation 7.3 can be formalized as some function $\zeta_g(t) = g(J(t))$ where J is the set of jobs running at time t. Since jobs have a start and end time, given a dataset with a dense enough sampling of J, $g(J(t))$ can be calculated for every point in time. During periods of time where e.g., the file system is suffering a service degradation, all jobs on the system will be impacted with varying severity. A model of the system does not need to understand how and why the degradation happened, it only needs to know degradation start and end times, and how different types of jobs were impacted. This time-based model is useless for predicting future performance, and its only utility is in evaluating how much of the degradation can be described as purely a function of time. A deployed model does not have insight into the future and will still need to observe the system.

To evaluate the global system impact, a golden model that exhibits no global modeling error is developed, against which other, 'real' ML models can be compared. Since the global system impact $\zeta_g(t)$ only depends on time t and may ignore the set of all jobs J, only application behavior j and the job start time feature are exposed to the golden model. Both real and golden models have optimized hyperparameters and should have $e_{app} = 0$, but only the golden model has $e_{system} = 0$ (assuming enough data to memorize $\zeta_g(t)$ is available). The litmus test compares these two models to determine $e^p_{system} = e^p - e^g$. Here, a golden model is an XGBoost model fine-tuned on a validation set and evaluated on a test set. Assuming that the golden model is exposed to enough jobs throughout the lifetime of the system, it will learn the impact of $\zeta_g(t)$ even without having access to the underlying system features causing that impact. This golden model is used in the following litmus test:

System modeling error litmus test:

1. Run grid search on real model, find lowest e^p;
2. Insert job start time feature t into the dataset;
3. Run grid search on golden model, find lowest e^g;
4. Calc. system modeling error $e^p_{system} = e^p - e^g$;

If the litmus test is applied correctly, the golden model only suffers from the last three classes of errors: poor generalization, local system impact, and inherent noise. Note that the litmus test is applied on the whole dataset, and not just duplicates, because the less numerous duplicate jobs do not cover the whole lifetime of the system well. In Figure 7.5 we evaluate a baseline model (blue) and a model enriched with the job start time (orange). Adding a start time feature has a large impact on error: on

Cori, the error drops 40%, from 16.49% down to 10.02%, while on Theta the error drops by 30.8%. To obtain this higher accuracy on the POSIX+time dataset, a far larger model is needed, i.e., one that can remember the I/O weather throughout the lifetime of the system.

Note that the timestamp feature fed to the golden model serves no purpose at deployment time, since the ML model cannot learn the state of the system as it is happening. This golden model *is* useful to retrospectively analyze past states and validate that deployment-time models are not suffering from system modeling errors.

7.2.5.2 Improving modeling through I/O visibility

With an estimate of minimal error achievable assuming perfect application and global system modeling, we investigate whether I/O subsystem logs can help models approach this limit. Since Theta does not collect I/O subsystem logs, we analyze Cori, which collects both application and I/O logs. Figure 7.5 shows the XGBoost performance of three models: a baseline where $e_{app} = 0$ (blue), the litmus test's golden model where also $e_{system} = 0$ (orange), and a Lustre-enriched model (green). Cori's median absolute error is reduced by 40%, from 16.49% down to 9.96%. The Lustre-enriched results are surprisingly close to the litmus test's predictions, and suggest that predictions cannot be improved through further I/O insight since the litmus test's prediction is reached.

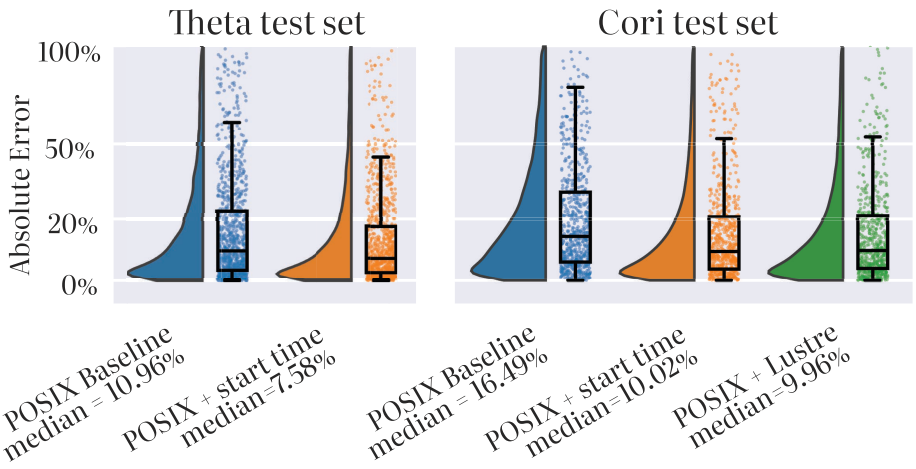


Figure 7.5: Error distribution of models trained on (1) POSIX, (2) POSIX + the start time feature, and (3) Darshan and Lustre.

7.2.6 Generalization errors

The remaining three classes of error are caused by lack of data and not poor modeling, as the top branch of the taxonomy shows. While I/O contention and inherent noise errors are examples of aleatory uncertainty and are caused by lack of insight into specific jobs, generalization errors stem from epistemic uncertainty, i.e., the lack of other logged jobs around a specific job of interest. To motivate this section, in the third graph of Figure 7.2 we show error distribution of a model trained on data from January 2018 to July 2019. When evaluated on held-out data from the same period, the median absolute error is low (green line). Once the model is deployed and evaluated on the data collected after the training period (July 2019 and after), median error spikes up (red line).

7.2.6.1 Estimating generalization error

Estimating the amount of out-of-distribution error e_{ood} is important because any unaccounted OoD error will be classified as noise or contention. This will make systems that run a lot of novel jobs appear to be more noisy than they truly are. Because OoD and ID jobs likely have a similar amount of I/O and contention noise, false positives (ID jobs classified as OoD) are preferable over false negatives, since false negatives contribute to overestimating I/O noise. To estimate the impact of out-of-distribution jobs on error e_{ood}, we aim to quantify how much of the error is epistemic and how much is aleatory in nature, as shown in Figure 7.2 (upper right). The leading paradigm for uncertainty quantification works by training an ensemble of models and evaluating all of the models on the test set. If the models make the same error, the sample has high aleatory uncertainty, but if the models disagree, the sample has high epistemic uncertainty (Lakshminarayanan, Pritzel, and Blundell 2017). The intuition is that predictions on out-of-distribution samples will vary significantly on the basis of the model architecture, whereas predictions on ID but noisy samples will agree and exhibit the same bias. Since this method relies on ensemble to have great model diversity, several works have explored increasing diversity through different model hyperparameters (Wenzel et al. 2020), different architectures (Zaidi et al. 2020), or both (Egele et al. 2021). We choose to use AutoDEUQ (Egele et al. 2021), a method that evolves an ensemble of neural network models and jointly optimizes both the architecture and hyperparameters of the models. While in theory any type of machine learning model can be used for the model population, neural networks are attractive due to their high hyperparameter count, diverse architectures found in practice, and high generalization capability. Additionally, AutoDEUQ's Neural

Architecture Search (NAS) is compatible with the NAS search from section 7.2.4, reducing the computational load of applying the taxonomy. Note that in order for AutoDEUQ to correctly split error into e_{ood} vs. $e_{contention} + e_{noise}$, first all application and system modeling errors e_{app} and e_{system} must be removed. Therefore, the function of the NAS is two-fold in this litmus test: (1) eliminate application and system modeling errors, and (2) create a diverse model population. Figure 7.6 shows the distribution of epistemic (EU) and aleatory uncertainties (AU) of Theta and Cori test sets. For both systems, aleatoric uncertainty is significantly higher than epistemic uncertainty. Furthermore, *all* jobs seem to have AU larger than about 0.05, hinting at the inherent noise present in the system. The inverse cumulative distributions on the margins (red) show what percentage of total error is caused by AU / EU *below* that value. For example, for both systems 50% of all error is caused by jobs with EU below 0.04, while in case of AU, 50% of error is below AU=0.25. The low total EU is expected since the test set was drawn from the same distribution as the training set, and increases on the 2020 set (omitted due to space concerns).

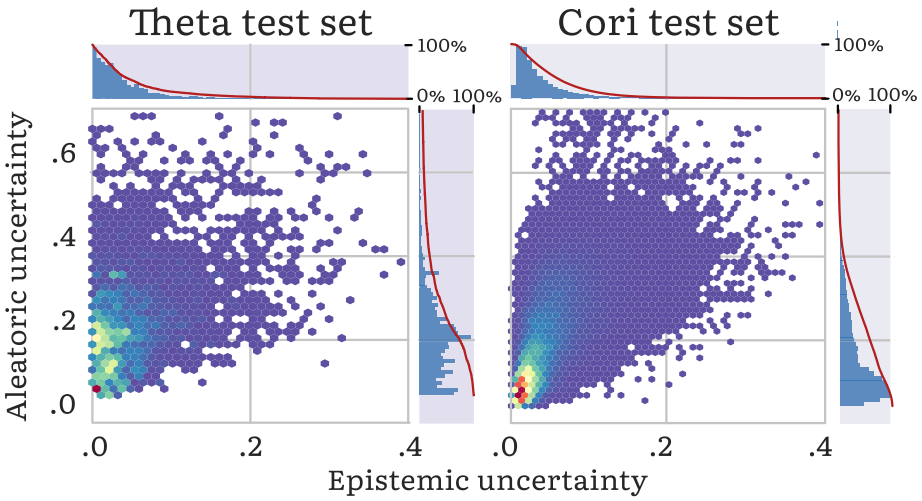


Figure 7.6: Distribution of prediction aleatory and epistemic uncertainties for the two systems, with marginal distributions (blue) and inverse cumulative error (red) shown on the margins.

Epistemic uncertainty does not directly translate into the out-of-distribution error e_{ood} from Equation 7.5. When a sample is truly OoD, it may not be possible to separate aleatory and epistemic uncertainty, since a good estimate of AU requires dense sampling around the job of interest. Therefore, we choose to attribute all errors of a sample marked as out-of-distribution to e_{ood}. This error attribution requires classifying every test set sample as either in- or out-of-distribution, but since EU estimates are continuous values, an EU threshold which will separate OoD and ID samples is required. Although this threshold is specific to the dataset and may require tuning, the quick drop or 'shoulder' in the inverse cumulative error graph around EU=0.1 in Figure 7.6 makes the choice of an EU threshold robust. A litmus test that estimates the error due to out-of-distribution samples has the following steps:

Out-of-Distribution error litmus test:

1. Run network architecture search:
   1.1. Minimize e^p_{app} and e^p_{system} for each model;
   1.2. Collect best performing models;
2. Estimate epistemic uncertainty using AutoDEUQ;
3. Find a stable epistemic uncertainty threshold;
4. Classify jobs as either ID or OoD based on threshold;
5. Calculate e_{ood} as the sum of OoD job errors.

On Theta, for an EU threshold of 0.24, .7% of the samples are classified as OoD, but constitute 2.4% of the errors, while on Cori 2.1% of error gets removed for the same EU threshold. In other words, the selected jobs have 3× larger average error than random samples. By visualizing the high-dimensional job features using the

Gauge tool (del Rosario et al. 2020a) and interactively exploring the types of jobs that do get removed, we confirm that OoD-classified jobs are typically rare or novel applications.

7.2.7 I/O Contention and Inherent Noise Errors

With the ability to estimate the amount of application and system modeling error, as well as detect outlier jobs, leftover error is caused by system contention or inherent noise. Both of these error classes are caused by aleatory uncertainty, since the model lacks deeper insight into jobs or the system, as opposed to the OoD case where the model lacks samples. While e.g., application error was explainable in terms of broad application behavior (e.g., this application is slow because it frequently writes to shared files, but the model fails to learn this effect), the impact of contention and noise on I/O throughput is caused by lower level, transient effects. Though it may be possible to observe and log such effects through microarchitectural hardware counters or network switch logs, such logging would require vast amounts of storage per job and may impact performance. Lack of practical logging tools makes the last two error categories typically *unobservable*. Furthermore, these two classes may only be separated in hindsight, and while I/O noise levels may be constant, the amount of I/O contention on the system is unpredictable for a job that is about to run.

The questions we ask in this section are: how can errors due to noise and contention be separated from errors due to poor modeling or epistemic uncertainty? Is there a fundamental limit to how accurate I/O models can become? What steps are necessary to quantify system I/O variability?

7.2.7.1 Establishing the bounds of I/O modeling

To separate contention and noise impacts from the first three classes of error, we develop a litmus test based on the test from Section 7.2.4. There, by observing sets of duplicates, the error of a golden model e^g was estimated, where $e^g_{app} = 0$. Comparing real models against this ideal model allows for calculating a real model's e_{app}. This litmus test works by 'holding constant' application behavior j within a set of duplicates, i.e., by preventing any input variance from reaching the model. The here introduced noise and contention litmus test seeks to hold constant not only application behavior, but also global system impact, and impact from poor generalization. We design a litmus test that works by enforcing a stronger requirement on duplicate sets, where pairs of jobs are duplicates only if they have the both same application behavior j and same start time t. The test assumes that identical jobs ran at the same time are exposed to the same global system impact $\zeta_g(t)$, but not necessarily the same local impact. The litmus test therefore estimates *the sum* of contention and noise error for a golden model, where only concurrent duplicates are observed and both application behavior j and global system behavior $\zeta_g(t)$ are held static for each duplicate set.

Contention and noise error litmus test:

1. Remove OoD jobs as per previous litmus test;
2. For each set of concurrent duplicate jobs ($\Delta t = 0$):
   2.1. Calculate the set's mean I/O throughput;
   2.2. Apply Bessel's correction to mean;
   2.3. Use mean as golden model prediction;
   2.4. Calculate per-set mean GM absolute error;
3. Calculate $e_{contention} + e_{noise}$ as median of golden model per-set errors.

In the fifth column of Figure 7.2 we show the distribution of I/O throughput differences $\Delta\phi$ and timing differences Δt between all pairs of Cori duplicate jobs, weighted so that large duplicate sets are not overrepresented. The vertical strip on the left contains Cori duplicate jobs that were run simultaneously, largely because they were batched together. These jobs share j and ζ_g, but may differ in ζ_l and ω. Due to the denser sampling around 1 minute to 1 hour range, it is not immediately apparent how the I/O difference changes between duplicates ran at the same time and duplicates ran with a small delay. By grouping duplicates from different Δt ranges and independently scaling them, a better understanding of duplicate I/O throughput distributions across timescales can be made, as shown in Figure 7.7 (Theta shown, Cori omitted due to lack of space). For both systems, the distributions on the right contain jobs ran over large periods of time where global system impact ζ_g might have changed, explaining the asymmetric shape of some of them. The left-most distributions are similar, since variance only stems from contention ζ_l and noise ω. While some distributions (e.g., the 10^5 to 10^6 second) show complex multimodal behavior, all of the distributions seem to contain the initial zero-second ($0s$ to $1s$) distribution.

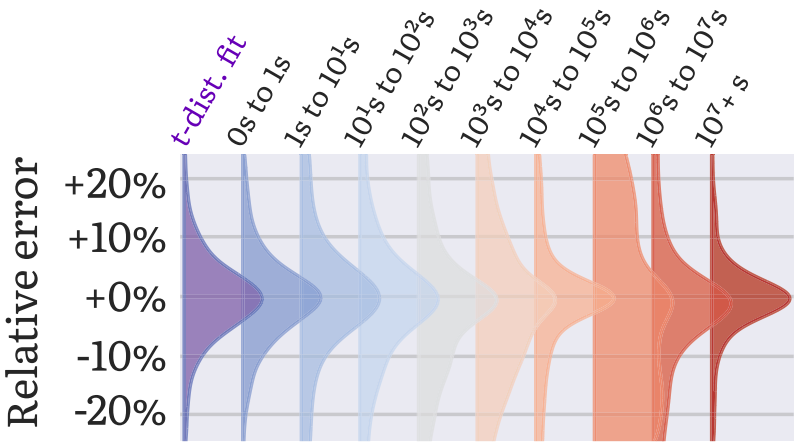


Figure 7.7: Distribution of errors for different periods between duplicate runs.

By fitting a normal distribution to the $\Delta t = 0$ distribution (0s to 1s) in Figure 7.7, we can both (1) learn the lower limit on total modeling error and (2) learn the system's I/O noise level, i.e., how much I/O throughput variance should jobs running on the system expect. However, upon closer inspection, the $\Delta t = 0$ distribution *does not* follow a normal distribution. This is surprising, since if noise follows some (not necessarily normal) stationary distribution, and is independent over time, and its effects are cumulative, according to the central limit theorem the total noise impact is a normal distribution. The answer lies in how the concurrent ($\Delta t = 0$) duplicates are sampled. When observing duplicates, in general, duplicate sets have between 2 and hundreds of thousands of identical jobs in them. However, in duplicate sets with identical start times on Theta, 70% of the sets only have two identical jobs, and 96% have 6 jobs or less, with similar results on Cori. The issue stems from how small (sub-30 sample) duplicate set errors are calculated: when only a small number of jobs exist in the set, the mean I/O throughput of the set is biased by the sampling, i.e., the estimated mean is closer to the samples than the real mean is. This causes the set I/O throughput variance to decrease and therefore duplicate error estimate will be

reduced as well. Student's t-distribution describes this effect: when the true mean of a distribution is known, error calculations follow a normal distribution. When the true mean is not known, the biased mean estimate makes the error follow the t-distribution. As the set size increases, the t-distribution approaches a normal distribution. However, naively taking the variance of the t-distribution will produce a biased sample variance σ^2, which can be de-biased by applying Bessel's correction as $\bar{\sigma}^2 = \frac{n}{n-1}\sigma^2$.

With de-biasing in place, we estimate the I/O noise variance of the two systems. Results show that a job running on Theta can expect an I/O throughput within ±5.71% of the predicted value *68% of the time*, or within ±10.56% 95% of the time. For Cori, these values are ±7.21% and ±14.99%, respectively. This is a fundamental barrier not just to I/O model improvement, but to predictable system usage in general. Although some insight into contention can be gained through low-level logging tools, noise cannot be overcome. I/O practitioners can use this litmus test to evaluate the noise levels of their systems, and ML practitioners should reconsider how they evaluate models, since some systems may be simply harder to model.

7.2.8 Applying the taxonomy

We now illustrate how the proposed taxonomy can be used in practice. In Figure 7.8, we show the steps a modeler can follow to evaluate the taxonomy on a new system. Step 1: The modeler splits the available data into training and test sets, and then trains and evaluates some baseline machine learning model on the task of predicting I/O throughput. This model does not have to be fine-tuned, as the taxonomy will reveal the main sources of error and approximately how much the quality of the model is at fault. Step 2.1: The modeler estimates application modeling errors by

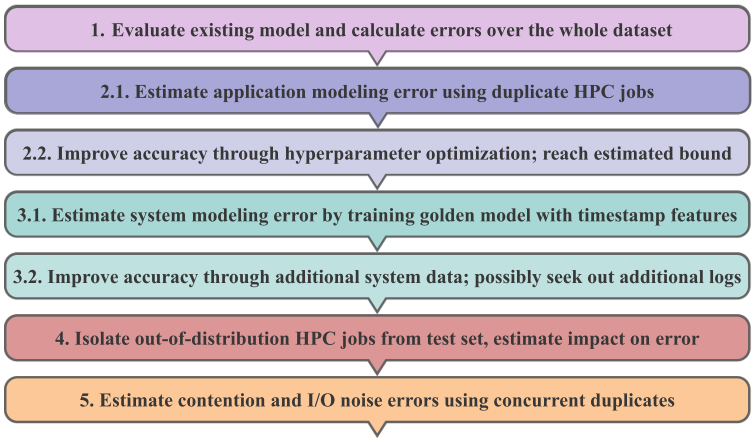


Figure 7.8: Framework for applying the taxonomy.

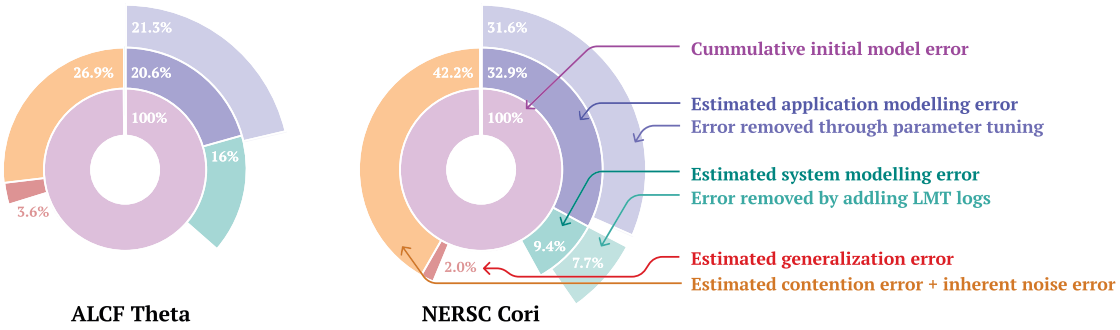


Figure 7.9: Results from ALCF Theta and NERSC Cori systems.

finding duplicate jobs and evaluating the mean predictor performance on every set of duplicates. Assuming that the distribution of duplicate HPC jobs is representative of the whole population of jobs, this step provides the modeler with a lower bound on the application modeling error. Step 2.2: By contrasting the baseline model error (Step 1) and the estimated application modeling error, the modeler can estimate the percentage of error that can be attributed to poor modeling. The modeler performs a hyperparameter or network architecture search and arrives at a good model close to the bound. Step 3.1: The modeler estimates system modeling errors by exposing the job start time feature to a golden model. This step requires that the modeler has developed a well-performing model in Step 2.2, i.e., one that achieves close to the

estimated ideal performance. The test set error of the model serves as an estimate of the application + system modeling lower bound. Step 3.2: The modeler explores adding sources of system data to improve the performance of the baseline model up to the estimated limit of application and system modeling. Step 4: The modeler identifies out-of-distribution samples using AutoDEUQ, calculates OoD error that stems from these samples, and removes them from the dataset. Step 5: The modeler estimates the error that can be attributed to contention and noise, as well as I/O variance of the system. This estimate is made by observing the I/O throughput differences between sets of concurrent duplicates, i.e., duplicate jobs ran at around the same time.

In Figure 7.9 we show the average baseline model error (inner pink circle segment) of both ANL Theta and NERSC Cori systems, and how that error is broken down into different classes of error. We do not focus on the cumulative (total) error value of the two systems; instead, we focus on attributing the baseline model error into the five classes of errors in the taxonomy (middle circle segments of the pie chart), and on the percentage of error that can be removed through improved application and system modeling (outer segments of the pie chart). The inner blue section of the two pie charts represents the estimated application modeling error, as arrived at in Step 2.1. The outer blue section represents how much of the error can be fixed through hyperparameter exploration, as explored in Step 2.2. The inner green section represents the estimated system modeling error, derived in Step 3.1. Note that the total percentage of system modeling error is relatively small on both systems; i.e., I/O contention, filesystem health, hardware faults, etc., do not have a dominant impact on I/O throughput. The outer green circle segment represents the percentage of error that can be fixed by including system logs (LMT logs in our case), as described in Step 3.2. Only the Cori pie chart has this segment, as Theta does not collect LMT

logs. On Cori, the inclusion of LMT logs helps remove most of the system modeling errors, reinforcing the conclusion that including other logs (i.e., topology, networking) may not help to significantly reduce errors. The inner red segment represents the percentage of error that can be attributed to out-of-distribution samples of the two systems, as calculated in Step 4. Finally, the yellow circle segment represents the percentage of error that can be attributed to aleatory uncertainty. For both Theta and Cori, this is a rather large amount, pointing to the fact that there exists a lot of innate noise in the behavior of these systems, and setting a relatively high lower bound on ideal model error.

The similarity between the modeling error estimates (Steps 2.1 and 3.1) and the actual updated model performance (Steps 2.2 and 3.2) is surprising and serves as evidence for the quality of the error estimates. However, the estimates of the five error classes *do not* add up to 100%. The first three error estimates are just that - estimates, derived from a subset of data (duplicate HPC jobs) that do not necessarily follow the same distribution as the rest of the dataset and may be biased. If we add the estimates, we see that on Theta 32.9% of the error is unexplained, and on Cori 13.5% of the error is unexplained. Cori's lower unexplained error may be due to the fact that we have collected some 1.1M jobs compared to 100K on Theta.

7.3 Results and Insights

Developing production-ready machine learning models that analyze HPC jobs and predict I/O throughput is difficult: the space of all application behaviors is large, HPC jobs are competing for resources, and the system changes over time. To efficiently improve these models, we present a taxonomy of HPC I/O modeling errors that enables independent study of different types of errors, helps quantify their impact, and identifies the most promising avenues for model improvement. Our taxonomy breaks errors into five categories: (1) application and (2) system modeling errors, (3) poor generalization, (4) resource contention, and (5) I/O noise. We present litmus tests that quantify what percentage of model error should be attributed to each class, and show that models improved by using the taxonomy are within a percentage point of an estimated best-case I/O throughput modeling accuracy. We show that a large portion of I/O throughput modeling error is irreducible and stems from I/O variability. We provide tests that quantify the I/O variability and establish an upper bound on how accurate I/O models can become. Our test shows that jobs ran on Theta and Cori can expect an I/O throughput standard deviation of 5.7% and 7.2%, respectively.

In future work, we plan to explore why error classes in Figure 7.9 do not add up to 100%. Our hypothesis that poor duplicate distribution is the source of this discrepancy, and that instead of duplicate jobs, a targeted set of repeated microbenchmarks may better inform the framework introduced in this work. By tuning and executing microbenchmarks representative of the system's application distribution, we hope to build a minimal set of workloads that evaluate system parameters such as I/O noise amount or application parameters such as I/O contention sensitivity. We also plan to

explore how transferable this set of benchmarks is, and whether different HPC system workloads can be accurately represented by a set of weighted microbenchmarks.

7.3.1 Application to System Sensors

As Figure 7.9 shows, a surprising amount of system unpredictability stems from low-level workload features that require runtime observations. The duplicate job litmus test shows that even adding e.g., hardware performance counters monitoring workload features will not help improve predictions on these jobs. The solution lies in improving observability of system features: this includes runtime measuring of cache and network (including I/O) latencies, monitoring compute node microarchitecture, and communicating this information between nodes in order to e.g., detect stragglers.

Since neither Cori not Theta litmus test account for 100% of errors, additional sensors may need to be developed in order to close the gap and provide greater confidence in error allocation. In future work I plan to explore automated and closed-loop systems that implement sensors, execute workloads, and evaluate the taxonomy, so that the necessary sensors may be developed automatically.

7.3.2 Application to Decision-Making Systems

Runtime decision-making systems have two main challenges: identifying jobs that can benefit from adaptation, and deciding on the best course of action for that adaptation. While the space of actions is highly-specific to the available system actuators, identifying underperforming jobs is system independent. Any set of duplicate jobs that has large performance variation is likely sensitive to some low-level system

effects, and with the correct decision-making and system actuators, all jobs within that set may be able to perform at the level of the 'best-in-class' job. In future work I plan to develop algorithms for not just predicting job performance, but also predicting best-case job performance. By comparing these two predictions, jobs with adaptation potential may be identified and their goal satisfaction may be improved.

7.3.3 Application to System Actuators

Since low-level environmental effects are a major contributor to system unpredictability, these effects cannot be controlled or adapted to at scheduling-time, and require dynamic adaptation. Due to the transient nature of these effects, low-latency actuators operating at the same timescale are as the environmental effects are necessary allow the system to react in time. In future work I plan to explore automated identification of beneficial actuators by building using model interpretation techniques to narrow down microarchitectural events causing workload unpredictability.

Chapter 8

CONCLUSION

In this dissertation, I have explored three classes of adaptive architectures, which adapt (i) during the design time across generations of a family of systems, (ii) at workload scheduling time between workload executions, and (iii) during workload execution.

Through the study of design-time general-purpose adaptive systems, I show that these systems suffer from lack of visibility into the changing distribution of workloads the system will encounter after deployment. Scheduling-time adaptive systems have greater visibility into system and workload conditions and can adapt to novel classes of workload and environments. However, even with scheduling-time adaptation, workloads exhibit significant unpredictability that cannot be accounted for.

In (Isakov et al. 2020) I develop a set of methods for clustering workloads in order to enable scalable analysis of large amounts of system logs. By building interpretable local models of small groups of similar jobs, I present methods that can identify which sensors or monitors are beneficial for understanding these jobs, as well as diagnose and explain workload and system bottlenecks. However, these methods arrive at fundamental limits of system predictability, i.e., they cannot predict workload and system behavior with less than $\approx 10\%$ variation.

To understand whether more workload logs, more insight into the system, or more system sensors are needed to improve models of systems and workloads, in (Isakov et al. 2022) I propose a methodology for classifying sources of system unpredictability. In this work I show that the majority of unaccounted uncertainty about how a system

will perform stems from low-level system effects that cannot be efficiently observed purely through software. With these conclusions, I show that low-level system and microarchitectural effects have a large but often unobserved impact on performance. Adapting to these effects requires both low-level sensors and actuators, for which I propose a RISC-V-based binary instrumentation tool that can observe binaries during runtime at the ISA-level, as well as a hardware architecture for low-impact monitoring of system microarchitecture.

8.1 Directions for Future Research

Modeling systems and extracting insight about any system bottlenecks, lack of visibility into workloads, and candidate system actuators is currently a slow and manual process. In future work, I plan to develop a system modeling framework that can: (i) automatically instrument workloads and implement system sensors, (ii) collect workload data and build system models, (iii) interpret models and present system bottlenecks and possible improvements, as well as (iv) report the sources of uncertainty by class. Through such a framework, system designers may be able to quantify the benefits of different classes of adaptation and whether their systems warrant statically or dynamic adaptation.

On a separate line of research, I plan to investigate how low-area and low-latency machine learning algorithms (Isakov, Ehret, and Kinsy 2018; Isakov and Kinsy 2020) can be used for runtime decision-making and online learning. By building tightly-integrated sense-decide-act loops in hardware, I aim to answer the question of how system adaptation timescales affect goal satisfaction, and how fast system actuators need to be to extract all adaptation benefits.

REFERENCES

Agarwal, A., J. Hennessy, and M. Horowitz. 1989. "An Analytical Cache Model." *ACM Trans. Comput. Syst.* (New York, NY, USA) 7, no. 2 (May): 184–215. https://doi.org/10.1145/63404.63407.

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Berlin, Heidelberg: Springer-Verlag.

Braam, Peter J, and Philip Schwan. 2002. "Lustre: The intergalactic file system." In *Ottawa Linux Symposium,* 8:3429–3441. 11.

Bruening, Derek L., and Saman Amarasinghe. 2004. "Efficient, Transparent, and Comprehensive Runtime Code Manipulation." PhD diss.

Campello, Ricardo J. G. B., Davoud Moulavi, Arthur Zimek, and Jörg Sander. 2015. "Hierarchical Density Estimates for Data Clustering, Visualization, and Outlier Detection." *ACM Trans. Knowl. Discov. Data* (New York, NY, USA) 10, no. 1 (July). https://doi.org/10.1145/2733381.

Carns, Philip, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. 2011. "Understanding and Improving Computational Science Storage Access through Continuous Characterization." *ACM Trans. Storage* (New York, NY, USA) 7, no. 3 (October). https://doi.org/10.1145/2027066.2027068.

Chen, Tianqi, and Carlos Guestrin. 2016. "XGBoost: A Scalable Tree Boosting System." In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* KDD '16.

del Rosario, E., M. Currier, M. Isakov, S. Madireddy, P. Balaprakash, P. Carns, R. B. Ross, K. Harms, S. Snyder, and M. A. Kinsy. 2020a. "Gauge: An Interactive Data-Driven Visualization Tool for HPC Application I/O Performance Analysis." In *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW),* 15–21. https://doi.org/10.1109/PDSW51947.2020.00008.

Demme, John, and Simha Sethumadhavan. 2011. "Rapid identification of architectural bottlenecks via precise event counting." In *2011 38th Annual International Symposium on Computer Architecture (ISCA).*

Dennard, R.H., F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. 1974. "Design of ion-implanted MOSFET's with very small physical

dimensions." *IEEE Journal of Solid-State Circuits* 9 (5): 256–268. https://doi.org/10.1109/JSSC.1974.1050511.

Dorier, Matthieu, Shadi Ibrahim, Gabriel Antoniu, and Rob Ross. 2014. "Omnisc'IO: a grammar-based approach to spatial and temporal I/O patterns prediction." In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis,* 623–634. IEEE.

Egele, Romain, Prasanna Balaprakash, Venkatram Vishwanath, Isabelle Guyon, and Zhengying Liu. 2020. "AgEBO-Tabular: Joint Neural Architecture and Hyperparameter Search with Autotuned Data-Parallel Training for Tabular Data." *CoRR* abs/2010.16358. arXiv: 2010.16358. https://arxiv.org/abs/2010.16358.

Egele, Romain, Romit Maulik, Krishnan Raghavan, Prasanna Balaprakash, and Bethany Lusch. 2021. "AutoDEUQ: Automated Deep Ensemble with Uncertainty Quantification." *CoRR* abs/2110.13511. arXiv: 2110.13511.

Ehret, Alan, Jacob Abraham, Mihailo Isakov, and Michel A. Kinsy. 2022. *Zeno: A Scalable Capability-Based Secure Architecture.* https://doi.org/10.48550/ARXIV.2208.09800.

Engelke, Alexis, Dominik Okwieka, and Martin Schulz. 2021. "Efficient LLVM-Based Dynamic Binary Translation." In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments.* VEE 2021. Virtual, USA: Association for Computing Machinery.

Ester, Martin, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. "A density-based algorithm for discovering clusters in large spatial databases with noise," 226–231. AAAI Press.

Ferdman, Michael, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware." http://infoscience.epfl.ch/record/173764.

Haziza, D., J. Rapin, and G. Synnaeve. 2020. *Hiplot, interactive high-dimensionality plots.* https://github.com/facebookresearch/hiplot.

Horspool, R. Nigel, and Nenad Marovac. 1980. "An Approach to the Problem of Detranslation of Computer Programs." *Comput. J.* 23:223–229.

Isaila, Florin, Prasanna Balaprakash, Stefan M. Wild, Dries Kimpe, Rob Latham, Rob Ross, and Paul Hovland. 2015. "Collective I/O Tuning Using Analytical and Machine Learning Models." In *Proceedings of the 2015 IEEE International Conference on Cluster Computing,* 128–137. CLUSTER '15. USA: IEEE Computer Society. https://doi.org/10.1109/CLUSTER.2015.29.

Isakov, Mihailo, Mikaela Currier, and Eliakin Del Rosario. 2022. *Gauge supporting experiments.* https://anonymous.4open.science/r/1c9a3777-0133-4db8-bff8-deffed0cad95/. Accessed: 2022-10-5.

Isakov, Mihailo, Mikaela Currier, Eliakin Del Rosario, Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert B. Ross, Glenn Lockwood, and Michel A. Kinsy. 2022. "A Taxonomy of Error Sources in HPC I/O Machine Learning Models." In *SC '22: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.*

Isakov, Mihailo, Alan Ehret, and Michel Kinsy. 2018. "ClosNets: Batchless DNN Training with On-Chip a Priori Sparse Neural Topologies." In *2018 28th International Conference on Field Programmable Logic and Applications (FPL),* 55–554. https://doi.org/10.1109/FPL.2018.00017.

Isakov, Mihailo, and Michel A. Kinsy. 2020. "NeuroFabric: Identifying Ideal Topologies for Training A Priori Sparse Networks." *CoRR* abs/2002.08339. arXiv: 2002.08339. https://arxiv.org/abs/2002.08339.

Isakov, Mihailo, Eliakin del Rosario, Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert B. Ross, and Michel A. Kinsy. 2020. "HPC I/O Throughput Bottleneck Analysis with Explainable Local Models." In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis.* https://doi.org/10.1109/SC41405.2020.00037.

Karniadakis, George Em, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. 2021. "Physics-informed machine learning." *Nature Reviews Physics* 3, no. 6 (June): 422–440.

Kodavasal, Janardhan, Kevin Harms, Priyesh Srivastava, Sibendu Som, Shaoping Quan, Keith Richards, and Marta García. 2016. "Development of a Stiffness-Based Chemistry Load Balancing Scheme, and Optimization of Input/Output and Communication, to Enable Massively Parallel High-Fidelity Internal Combustion Engine Simulations." 052203, *Journal of Energy Resources Technology* 138, no. 5 (February). eprint: https://asmedigitalcollection.asme.org/energyresources/article-pdf/138/5/052203/6148052/jert_138_05_052203.pdf.

Lakshminarayanan, Balaji, Alexander Pritzel, and Charles Blundell. 2017. "Simple and Scalable Predictive Uncertainty Estimation Using Deep Ensembles." In *Proceedings of the 31st International Conference on Neural Information Processing Systems,* 6405–6416. NIPS'17. Long Beach, California, USA: Curran Associates Inc.

Latham, Rob, Chris Daley, Wei-keng Liao, Kui Gao, Rob Ross, Anshu Dubey, and Alok Choudhary. 2012. "A case study for scientific I/O: improving the FLASH astrophysics code." *Computational Science & Discovery* 5, no. 1 (March): 015001.

Laurenzano, Michael A., Mustafa M. Tikir, Laura Carrington, and Allan Snavely. 2010. "PEBIL: Efficient static binary instrumentation for Linux." In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS),* 175–183.

Li, Boyang, Sudheer Chunduri, Kevin Harms, Yuping Fan, and Zhiling Lan. 2019. "The Effect of System Utilization on Application Performance Variability." In *Proceedings of the 9th International Workshop on Runtime and Operating Systems for Supercomputers.* ROSS '19. Phoenix, AZ, USA.

Li, Dongyang, Yan Wang, Bin Xu, Wenjiang Li, Weijun Li, Lina Yu, and Qing Yang. 2019. "PIPULS: Predicting I/O Patterns Using LSTM in Storage Systems." In *2019 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS),* 14–21. IEEE.

Lockwood, Glenn K., Wucherl Yoo, Suren Byna, Nicholas J. Wright, Shane Snyder, Kevin Harms, Zachary Nault, and Philip Carns. 2017. "UMAMI: A Recipe for Generating Meaningful Metrics through Holistic I/O Performance Analysis." In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems.* PDSW-DISCS '17.

Lozi, Jean-Pierre, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. "The Linux Scheduler: A Decade of Wasted Cores." In *Proceedings of the Eleventh European Conference on Computer Systems.* EuroSys '16. London, United Kingdom: Association for Computing Machinery. https://doi.org/10.1145/2901318.2901326.

Luk, Chi-Keung, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation." *SIGPLAN Not.* (New York, NY, USA) 40, no. 6 (June): 190–200. https://doi.org/10.1145/1064978.1065034.

Lundberg, Scott M, and Su-In Lee. 2017. "A Unified Approach to Interpreting Model Predictions." In *Advances in Neural Information Processing Systems 30,* edited by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, 4765–4774. Curran Associates, Inc.

Lundberg, Scott M., Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. 2020. "From local explanations to global understanding with explainable AI for trees." *Nature Machine Intelligence* 2 (1): 2522–5839.

Luu, Huong, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. 2015. "A Multiplatform Study of I/O Behavior on Petascale Supercomputers." In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing,* 33–44. HPDC '15. Portland, Oregon, USA: Association for Computing Machinery.

Madireddy, Sandeep, Prasanna Balaprakash, Philip Carns, Rob Latham, Glenn Lockwood, Robert Ross, Shane Snyder, and Stefan Wild. 2019a. "Adaptive Learning for Concept Drift in Application Performance Modeling." August.

Madireddy, Sandeep, Prasanna Balaprakash, Philip Carns, Robert Latham, Glenn K. Lockwood, Robert Ross, Shane Snyder, and Stefan M. Wild. 2019b. "Adaptive Learning for Concept Drift in Application Performance Modeling." In *Proceedings of the 48th International Conference on Parallel Processing.* ICPP 2019.

Madireddy, Sandeep, Prasanna Balaprakash, Philip Carns, Robert Latham, Robert Ross, Shane Snyder, and Stefan Wild. 2018a. "Modeling I/O Performance Variability Using Conditional Variational Autoencoders." In *2018 IEEE International Conference on Cluster Computing (CLUSTER),* 109–113.

Madireddy, Sandeep, Prasanna Balaprakash, Philip Carns, Robert Latham, Robert Ross, Shane Snyder, and Stefan M. Wild. 2018b. "Machine Learning Based Parallel I/O Predictive Modeling: A Case Study on Lustre File Systems." In *High Performance Computing,* 184–204. Cham: Springer International Publishing.

McCalpin, John D. 2013. *Notes on the mystery of hardware cache performance counters.* https://sites.utexas.edu/jdm4372/2013/07/14/notes-on-the-mystery-of-hardware-cache-performance-counters/. [Online; accessed 29-September-2022].

Mirbagher-Ajorpaz, Samira, Gilles Pokam, Esmaeil Mohammadian-Koruyeh, Elba Garza, Nael Abu-Ghazaleh, and Daniel A. Jiménez. 2020. "PerSpectron: Detecting Invariant Footprints of Microarchitectural Attacks with Perceptron." In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

Novaković, Dejan, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. "DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments." In *2013 USENIX Annual Technical Conference (USENIX ATC 13),* 219–230. San Jose, CA: USENIX Association, June. https://www.usenix.org/conference/atc13/technical-sessions/presentation/novakovi%7B%5C'c%7D.

Nowak, Andrzej, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel. 2015. "Establishing a Base of Trust with Performance Counters for Enterprise Workloads." In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference.* USENIX ATC '15. USENIX Association.

Pavan, P. J., J. L. Bez, M. S. Serpa, F. Z. Boito, and P. O. A. Navaux. 2019. "An Unsupervised Learning Approach for I/O Behavior Characterization." In *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD),* 33–40.

Rosario, Eliakin del, Mikaela Currier, Mihailo Isakov, Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert B. Ross, Kevin Harms, Shane Snyder, and Michel A. Kinsy. 2020b. "Gauge: An Interactive Data-Driven Visualization Tool for HPC Application I/O Performance Analysis." In *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW),* 15–21. https://doi.org/10.1109/PDSW51947.2020.00008.

Rubin, N., T. Tye, J. Yates, M. Herdeg, A. Chernoff, R. Hookway, C. Reeve, and S. Yadavalli. 1998. "FX!32: A Profile-Directed Binary Translator." *IEEE Micro* (Los Alamitos, CA, USA) 18, no. 02 (March): 56–64. https://doi.org/10.1109/40.671403.

Ruiz-Alvarez, Arkaitz, and Kim Hazelwood. 2008. "Evaluating the impact of dynamic binary translation systems on hardware cache performance." In *2008 IEEE International Symposium on Workload Characterization.*

Schmuck, Frank, and Roger Haskin. 2002. "{GPFS}: A {Shared-Disk} File System for Large Computing Clusters." In *Conference on File and Storage Technologies (FAST 02).*

Shan, Hongzhang, and John Shalf. 2007. "Using IOR to analyze the I/O Performance for HPC Platforms" (June). https://www.osti.gov/biblio/923356.

Shoshitaishvili, Yan, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, et al. 2016. "SOK: (State of) The Art of War:

Offensive Techniques in Binary Analysis." In *2016 IEEE Symposium on Security and Privacy (SP),* 138–157. https://doi.org/10.1109/SP.2016.17.

Snyder, S., P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright. 2016. "Modular HPC I/O Characterization with Darshan." In *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT),* 9–17.

Srinivasan, A., C. D. Sudheer, and S. Namilae. 2016. "Optimizing Massively Parallel Simulations of Infection Spread Through Air-Travel for Policy Analysis." In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid),* 136–145. May.

Tuncer, Ozan, Emre Ates, Yijia Zhang, Ata Turk, Jim Brandt, Vitus J. Leung, Manuel Egele, and Ayse K. Coskun. 2017. "Diagnosing Performance Variations in HPC Applications Using Machine Learning." In *High Performance Computing,* 355–373.

Wächter, Andreas, and Lorenz T. Biegler. 2006. "On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming." *Math. Program.* (Berlin, Heidelberg) 106, no. 1 (March): 25–57.

Wan, Lipeng, Matthew Wolf, Feiyi Wang, Jong Youl Choi, George Ostrouchov, and Scott Klasky. 2017. "Comprehensive Measurement and Analysis of the User-Perceived I/O Performance in a Production Leadership-Class Storage System." In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS),* 1022–1031. https://doi.org/10.1109/ICDCS.2017.257.

Weaver, Vincent, Dan Terpstra, and Shirley Moore. 2013. "Non-determinism and overcount on modern hardware performance counter implementations." April.

Weaver, Vincent M., and Sally A. McKee. 2008. "Can hardware performance counters be trusted?" In *2008 IEEE International Symposium on Workload Characterization.* https://doi.org/10.1109/IISWC.2008.4636099.

Wenzel, F., Jasper Snoek, Dustin Tran, and Rodolphe Jenatton. 2020. "Hyperparameter Ensembles for Robustness and Uncertainty Quantification." *ArXiv* abs/2006.13570.

Xie, Bing, Yezhou Huang, Jeffrey S. Chase, Jong Youl Choi, Scott Klasky, Jay Lofstead, and Sarp Oral. 2017. "Predicting Output Performance of a Petascale Supercomputer." In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing,* 181–192. HPDC '17. Washington, DC, USA: Association for Computing Machinery.

Xie, Bing, Zilong Tan, Philip Carns, Jeff Chase, Kevin Harms, Jay Lofstead, Sarp Oral, Sudharshan S. Vazhkudai, and Feiyi Wang. 2021. "Interpreting Write Performance of Supercomputer I/O Systems with Regression Models." In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS),* 557–566. https://doi.org/10.1109/IPDPS49936.2021.00064.

Yang, X., J. Jenkins, M. Mubarak, R. B. Ross, and Z. Lan. 2016. "Watch Out for the Bully! Job Interference Study on Dragonfly Network." In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis,* 750–760. https://doi.org/10.1109/SC.2016.63.

Yang, Xu, John Jenkins, Misbah Mubarak, Robert B. Ross, and Zhiling Lan. 2016. "Watch Out for the Bully! Job Interference Study on Dragonfly Network." In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis,* 750–760. https://doi.org/10.1109/SC.2016.63.

Zaheer, Manzil, Satwik Kottur, Siamak Ravanbakhsh, Barnabás Póczos, Ruslan Salakhutdinov, and Alexander J. Smola. 2017. "Deep Sets." *CoRR* abs/1703.06114. arXiv: 1703.06114. http://arxiv.org/abs/1703.06114.

Zaidi, Sheheryar, Arber Zela, Thomas Elsken, Chris C. Holmes, Frank Hutter, and Yee Whye Teh. 2020. "Neural Ensemble Search for Uncertainty Estimation and Dataset Shift."