Research Statement

Rishabh Iyer

I am a computer systems researcher, and my work lies at the intersection of systems, networking, computer architecture, and formal verification. My research focuses on developing techniques that enable engineers to reason about the expected performance of their systems *before* they are deployed in production. My dissertation introduced the notion of **performance interfaces**: simple, succinct programs that enable engineers to reason precisely about a system's expected performance behavior just like *semantic interfaces* (e.g., code documentation, header files, and specifications) enable reasoning about system functionality.

The best way to understand performance interfaces, the problems they solve, and their potential impact, is by analogy to semantic interfaces. Semantic interfaces provide simple, succinct descriptions of a system's functionality, enabling engineers to reason about the system's semantic behavior without having to delve into its implementation. The introduction of semantic interfaces revolutionized systems development; engineers today routinely use third-party code, and infrastructure operators frequently deploy systems they did not build themselves, none of which would be feasible without semantic interfaces.

In contrast, there exist no equivalent interfaces that enable precise reasoning about *performance*, despite performance increasingly being a first-class citizen in system design. Widely used representations, such as performance envelopes and benchmarks, provide incomplete visibility, leading to frequent hiccups and meltdowns in production. Such events are costly; for instance, delays on the order of milliseconds can lead to millions of dollars in lost revenue [24,26], and the stakes are higher for emerging systems such as self-driving cars where delays can result in the loss of human life [2,16].

Approach. We aim to realize our vision of performance interfaces via two complementary thrusts.

- **Thrust 1.** First, we design techniques and tools that enable summarizing the performance of systems software and hardware in succinct yet precise interfaces (§1). Our key insight here is to *decouple performance behavior from functionality*, and we show that doing so enables the design of interfaces that capture all performance-relevant details of a system while being orders of magnitude simpler than the implementation. To make these interfaces immediately useful, we develop tools that extract such performance interfaces from the corresponding implementation for a wide range of systems software and hardware, ranging from packet processing applications [10,13], to low-level systems code such as OS system calls and cryptographic libraries [11], and finally specialized hardware accelerators for deep learning and system infrastructure tasks [12,18].
- **Thrust 2.** Second, we redesign systems such that their resulting performance behavior is *predictable*, and thus amenable to summarization and abstraction into performance interfaces (§2). A common theme here is the integration of techniques that span different domains (e.g., operating systems, formal methods, networking, and compilers) to unlock new possibilities to abstract and simplify the system in question.

Impact. The tools we developed as part of the first research thrust are being used by several academic groups [3, 17] and have attracted interest from practitioners at Meta, Alibaba, and Google. Furthermore, some of the systems we built as part of the second thrust have already been deployed in production. For example, our proposed load balancer has been deployed in production at Alibaba since 2021, and our work on OS kernel extensions has been upstreamed into the Linux kernel mainline [1,4,5] and is currently in the late prototype stage at Meta.

1 Performance Interfaces for Systems Software and Hardware

Performance interfaces for network functions [10,13]. We first concretized the notion of performance interfaces in the context of network functions (NFs), i.e., in-network packet-processing applications such as load balancers and firewalls. NFs are typically on the critical path of serving user requests and often face unpredictable user traffic, making their performance a key concern for operators [19].

Our work answers the question: *What should a human-readable performance interface look like?* The key challenge here is designing a representation that is both simple and accurate. These goals are conflicting because accuracy typically requires adding detail, which hurts simplicity.

We proposed that a performance interface for a program P be another *program* I_P that takes the same inputs as P and returns P's processing latency. I_P returns P's latency not as concrete numbers but as *formulae* expressed in terms of P's inputs, the state that P maintains, the configuration parameters it reads at startup, and the hardware it runs on. I_P also includes a *resolution* r that represents the smallest difference in latency that I_P can specify: if $\mathcal{L}(P(I))$ is P's latency given input I, then $|I_P(I) - \mathcal{L}(P(I))| < r$, $\forall I$.

We chose to represent the performance interface as a program because programs are easy for engineers to understand, and are executable, which enables empirical reasoning about factors that are hard to reason about analytically (e.g., microarchitectural details of different hardware platforms). Additionally, expressing performance as symbolic formulae enables the interface to succinctly capture P's performance behavior across different inputs and deployment environments. Finally, the resolution provides readers of the interface with the flexibility to choose between multiple levels of abstraction (i.e., trading off accuracy for improved readability) based on the performance variability they are willing to tolerate in their deployment scenarios. A performance interface at a specified resolution only differentiates between inputs whose performance differs by more than the resolution, and so implementation details that cause performance variability irrelevant to the reader are abstracted away.

We designed and implemented PIX: a tool that takes as input NF code written in C and outputs performance interfaces in the form of Python programs. Under the covers, PIX uses a combination of static analysis, symbolic execution, and binary instrumentation to automatically analyze the NF code. PIX-generated performance interfaces are accurate yet orders of magnitude simpler than the code, and can be used to identify performance regressions, diagnose and fix performance bugs, and evaluate the latency impact of NIC offloads. Since publication, PIX has been used by researchers at several universities, including Imperial College London [17], Politecnico di Milano [3], and Carnegie Mellon University.

Abstractions for reasoning about the CPU cache usage of systems code [11]. Since semantic interfaces describe not only a program's expected output(s) but also any related side effects (i.e., modifications to shared state), performance interfaces must also describe performance side effects. Performance side effects arise because programs running on the same hardware (e.g., caller and callee, application and OS) share microarchitectural resources (e.g., caches), which can lead to unpredictable performance behavior [23, 29].

We focus on a dominant source of performance side effects, namely the CPU cache. Our goal is to help engineers answer questions such as: "How does the code's cache usage vary as a function of the workload?" To answer such questions, engineers require visibility into how the code processes an *abstract* workload. However, existing tools, such as profilers and cycle-accurate simulators, only provide visibility into *concrete* workloads.

We designed and implemented CFAR—a tool that automatically processes systems code into answers to questions about how that code uses the cache. CFAR's processing works in two phases: First, CFAR extracts from the input code an abstract representation (a " memory distillate") of how the code accesses memory. Then, CFAR uses simple programs ("projectors") to transform the distillate into answers to specific questions about the code's cache usage. The distillate serves as a precise abstraction of the code's memory usage (i.e., it contains all the information relevant to how the code accesses memory), enabling developers to use projectors to answer diverse questions about cache behavior.

We demonstrated that CFAR enables engineers to not only identify performance bugs and security vulnerabilities in their code but also understand the performance impact of incorporating third-party code into their systems without extensive benchmarking. Since publication, CFAR has appeared at the Linux Plumbers Conference (the top venue for Linux practitioners) and has attracted interest from practitioners at Meta and Alibaba.

Performance interfaces for hardware accelerators [12, 18]. As a postdoc, I helped extend the notion of performance interfaces to hardware accelerators. While system designers are increasingly reliant on accelerators for performance improvements, building systems that use accelerators remains challenging because accelerators typically come with little actionable information about their expected performance.

To bring performance interfaces to accelerators, we proposed a new abstraction for representing hardware performance, which we call a Latency Petri Net (LPN). We needed a new abstraction since hardware's execution model is inherently parallel and asynchronous, which leads to complex performance dependencies such as queuing and backpressure. Petri Nets [21] provided a good starting point for our abstraction since they were designed to model concurrent systems, and so are suited to precisely capture hardware's parallel execution model.

We developed a toolchain (LTC) that automatically transforms the LPN into human-readable performance interfaces that engineers can use to make informed design decisions. LTC can also transform the LPN into other representations, such as a fast performance simulator that enables compilers to quickly yet accurately optimize code for the accelerator, as well as a verification condition that engineers can pass to an SMT solver to verify key performance properties of their system before deployment.

Since publication, the LPN abstraction has been integrated into Simbricks [22], a performance simulator developed by an eponymous startup that enables end-to-end simulation of accelerated systems. LPNs have also attracted initial interest from Google's platforms team to speed up the simulation of candidate designs for new accelerators on their ARM-based SoCs.

2 Designing Systems with Predictable Performance

While the previous thrust focused on abstracting a given system implementation, this thrust focuses on redesigning systems to ensure predictable performance behavior. To do so, we integrate techniques from various domains (e.g., operating systems, formal methods, compilers, and networking) to simplify the system and enhance performance predictability.

Fast, flexible, and practical kernel extensions [7]. OS kernel extensions enable applications to customize the OS to meet specific performance and functionality requirements. For instance, engineers can streamline the general-purpose kernel I/O stack or tweak OS scheduling policies to ensure predictable performance. Unfortunately, existing solutions constrain users either in the extent of the functionality that they can express or the performance overheads incurred by their extensions.

As a postdoc, I helped design KFlex, a new approach to kernel extensibility that strikes an improved balance between the expressivity and performance of kernel extensions. To do so, KFlex separates the management of kernel-owned resources (e.g., kernel memory) from extension-specific resources (e.g., extension memory). This separation allows KFlex to use distinct mechanisms to manage each class of resources—automated verification for the safety-critical kernel-owned resources and lightweight runtime checks for extension-owned resources—which enables users to customize the OS in ways that are infeasible today and provides significant end-to-end performance benefits for applications.

Several of KFlex's mechanisms have already been integrated into the Linux kernel, with efforts ongoing for the rest. KFlex was also awarded an inaugural eBPF Research Award by the Linux Foundation and is in the late prototype stage at Meta, who are using its improved flexibility to deploy complex scheduling policies fleetwide.

Efficient microsecond-scale scheduling [14]. In addition to enabling engineers to express diverse OS *policies* with KFlex, our work also addresses OS *mechanisms*. Specifically, we focus on improving the efficiency of the OS scheduler's mechanism for swapping out the currently running thread. The efficiency of this mechanism is crucial for latency-sensitive services like web search, where short requests should not be blocked by long-running ones.

We built Concord, an OS scheduler that can switch between threads 4× more efficiently than the state of the art. Concord separates two scheduling concerns, namely *when* to preempt a thread and *how* the thread should give up the CPU. In Concord, the OS scheduler is responsible for the former, as it has visibility into all requests in the system, while the threads are responsible for the latter and *cooperatively* yield the CPU to enable faster switching. Concord uses automated compiler instrumentation to facilitate transparent communication between the scheduler and threads over dedicated cache lines. Since dedicated shared cache lines are the fastest way for two threads to communicate on shared-memory hardware, Concord's software switching mechanism outperforms even Intel's latest hardware support for scheduling [27].

Eliminating load balancer bottlenecks for cloud services [15]. Load balancers are essential in cloud services, but can often become a bottleneck and lead to latency spikes as they are on the critical path of every packet.

I helped design CRAB, a technique that eliminates these bottlenecks by observing that load-balancing decisions are made *only once per connection* during setup. After the initial SYN packet, the load balancer can be removed from the critical path, preventing latency spikes for all subsequent packets.

We realized CRAB by introducing a new TCP option to bypass the load balancer and implementing the option using OS kernel extensions. CRAB's design is pragmatic and takes advantage of the fact that cloud providers control both the physical infrastructure and the OS images used by tenants in their datacenters. This pragmatic design has ensured that CRAB's techniques have been deployed as part of Alibaba's Next-Generation Load Balancer since 2021.

3 Future Research

The computing landscape is becoming increasingly heterogeneous with the widespread use of accelerators, new memory technologies, and diverse networking fabrics, each optimized for a specific class of applications. In parallel, we are witnessing the rise of incredibly compute- and cost-intensive workloads, such as large-scale AI training and inference. Together, these trends make it imperative that the abstractions and interfaces used to build the systems of tomorrow treat performance as a first-class citizen. Failing to do so will result in the software underutilizing the capabilities of heterogeneous hardware and lead to prohibitive costs.

My work on performance interfaces makes me optimistic that we can build such systems to be efficient and cost-effective, just like semantic interfaces enabled building systems that were more complex yet safer than any that came before. I plan to leverage my experience in building systems with well-understood performance to tackle this challenge and expand my work into other areas of computer systems. Some specific directions I am interested in include:

Performance interfaces for large-scale distributed systems. I am particularly keen to expand my vision of performance interfaces to large-scale distributed AI training and inference. Given the current price and scarcity of GPUs, as well as the fact that bottlenecks in such systems can shift with each generation of

GPUs [25, 31], we have been repeatedly approached by practitioners to develop techniques that estimate the end-to-end performance gains offered by newer GPU models without requiring large-scale purchases.

I am also keen to expand my work to large-scale distributed applications. Such applications are increasingly broken down into thousands of component microservices, which, while enabling faster innovation and better fault tolerance, are harder to reason about due to complex interactions between individual components. I believe that an LPN-like performance abstraction that accurately captures interactions between microservices could enable precise reasoning about end-to-end performance for such applications.

Hardware-software co-design for predictable performance. In this work, I seek to realize performance predictability from the ground up by redesigning the interface between hardware and software (i.e., the ISA). The ISA is a *semantic interface*, and has largely remained static even as the underlying microarchitecture—which determines performance—has evolved dramatically (e.g., with the introduction of out-of-order execution, multiple layers of caching, etc). This raises the question: Can we augment the ISA to enable predictable performance? For example, exposing events in the memory hierarchy could allow software to control object placement and eviction, and exposing scheduling priorities in memory controllers could enable software to leverage policies like those used in network switches to mitigate the increased memory latency caused by CXL and tiered memory.

Programming framework for safe cyber-physical systems. I am also keen to expand my work to emerging cyber-physical systems (CPS), such as self-driving cars and surgical robots, where meeting performance requirements is necessary for functional correctness. Despite significant research on the topic, developing correct CPS software remains challenging, even for experts [16]. My goal is to answer questions such as: Can we build a programming framework that enables *correct-by-construction* CPS software by leveraging programming language techniques such as resource-aware types [9]? I believe this domain is amenable to disruption with new programming frameworks much like how programming frameworks such as MapReduce [6] and Spark [28] revolutionized cluster computing.

Energy-efficient datacenter systems. Beyond performance interfaces, I am also keen on redesigning datacenter systems to improve energy efficiency and reduce carbon emissions. Datacenters are estimated to consume 1-3% of the world's electricity today, and this figure is only expected to increase with the emergence of large-scale AI [8]. I believe that addressing this challenge requires designing new system abstractions that treat energy and carbon emissions as first-class citizens, enabling engineers and tools such as datacenter schedulers to make informed design and deployment decisions. I aim to leverage my experience in making performance a first-class citizen in systems design to do the same for energy efficiency.

Systems abstractions for emerging hardware technologies. Finally, in the longer term, I am excited about designing systems for emerging hardware technologies, particularly optical networks and optical compute. Such hardware has the potential to provide manifold improvements in both performance and energy efficiency, but harnessing this potential will require rethinking the system stack from the ground up. For example, optical interconnects today can enable direct chip-to-chip communication using light [20], which raises questions such as: What should network interfaces and network stacks for such systems look like? Similarly, optical compute today offers a faster and more energy-efficient alternative for computations such as matrix multiplications [30], which raises questions such as: How should we redesign our systems and programming abstractions to best make use of such hardware?

In summary, I am enthusiastic about the challenges and opportunities awaiting systems researchers in the coming years. I believe my experience in building systems with well-understood performance behavior will serve me well given that such systems are becoming increasingly necessary. I am also interested in broader topics in systems and am excited to address emerging technological challenges, such as new hardware substrates, and societal challenges, such as energy efficiency.

References

[1] Cleaning up after BPF exceptions. https://lwn.net/SubscriberLink/969185/47c78a0491515046/.

[2] Cruise Self-Driving Car Hits Pedestrian In San Francisco. https://www.sfchronicle.com/projects/2024/cruise-sf-collision-timeline/.

[3] eBPF equivalence checker. https://github.com/sebymiano/ebpf-equivalence-check.

[4] Exceptions in eBPF - Linux Plumbers Conference 2023. https://lpc.events/event/17/contributions/1578/attachments/1240/2521/Exceptions%20in%20BPF.pdf.

[5] Stack unwinding with exceptions in eBPF. https://lwn.net/Articles/938435/.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX Symposium on Operating System Design and Implementation*, 2004.

[7] Kumar Kartikeya Dwivedi, Rishabh R. Iyer, and Sanidhya Kashyap. Fast, Flexible, and Practical Kernel Extensions. In *ACM Symposium on Operating Systems Principles*, 2024.

[8] Goldman Sachs. AI Poisted to Drive Significant Increase in Datacenter Power Demand. https://www.goldmansachs.com/insights/articles/AI-poised-to-drive-160-increase-in-power-demand. [Last accessed on 2024-10-23].

[9] Jan Hoffmann and Steffen Jost. Two Decades of Automatic Amortized Resource Analysis. In *Mathematical Structures in Computer Science*, 2022.

[10] Rishabh R. Iyer, Katerina Argyraki, and George Candea. Performance Interfaces for Network Functions. In *Symposium on Networked Systems Design and Implementation*, 2022.

[11] Rishabh R. Iyer, Katerina J. Argyraki, and George Candea. Automatically Reasoning About How Systems Code Uses the CPU Cache. In *USENIX Symposium on Operating Systems Design and Implementation*, 2024.

[12] Rishabh R. Iyer, Jiacheng Ma, Katerina J. Argyraki, George Candea, and Sylvia Ratnasamy. The Case for Performance Interfaces for Hardware Accelerators. In *Workshop on Hot Topics in Operating Systems*, 2023.

[13] Rishabh R. Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina J. Argyraki, and George Candea. Performance Contracts for Software Network Functions. In *Symposium on Networked Systems Design and Implementation*, 2019.

[14] Rishabh R. Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In *Symposium on Operating Systems Principles*, 2023.

[15] Marios Kogias, Rishabh R. Iyer, and Edouard Bugnion. Bypassing the Load Balancer without Regrets. In *Symposium on Cloud Computing*, 2020.

[16] Ao Li and Ning Zhang. Data-flow Availability: Achieving Timing Assurance in Autonomous Systems. In *USENIX Symposium on Operating Systems Design and Implementation*, 2024.

[17] Dana Lu, Boxuan Tang, Michael Paper, and Marios Kogias. Towards Functional Verification of eBPF Programs. In *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, 2024.

[18] Jiacheng Ma, Rishabh R. Iyer, Sahand Kashani, Mahyar Emami, Thomas Bourgeat, and George Candea. Performance Interfaces for Hardware Accelerators. In *USENIX Symposium on Operating Systems Design and Implementation*, 2024.

[19] Survey of NF Operators and Results. Taken from Microscope [SIGCOMM'20]. https://www.dropbox.com/s/66cp4k3wl8zm0q5/survey.pdf?dl=0. [Last accessed on 2024-03-15].

[20] The Future of Chip Connectivity. https://ayarlabs.com/blog/thefuture-of-chip-connectivity-ucie-and-optical-i-o-faqs-explained. [Last accessed on 2024-10-23].

[21] Wolfgang Reisig. *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer Publishing Company, 2013.

[22] Simbricks: Fast Full-System Virtual Prototyping for Heterogeneous Computer Hardware. https://www.simbricks.io. [Last accessed on 2024-10-23].

[23] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Symposium on Operating Systems Design and Implementation*, 2010.

[24] The Cost of Latency. https://perspectives.mvdirona.com/2009/10/the-cost-of-latency. [Last accessed on 2024-03-15].

[25] vLLM Performance Update and Roadmap. https://blog.vllm.ai/2024/09/05/perf-update.html. [Last accessed on 2024-10-23].

[26] Why Brands are Fighting over Milliseconds. https://www.forbes.com/sites/steveolenski/2016/11/10/why-brands-are-fighting-over-milliseconds. [Last accessed on 2024-03-15].

[27] x86 Support for User Interrupts. https://lwn.net/Articles/869140/. [Last accessed on 2024-03-15].

[28] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[29] Arash Pourhabibi Zarandi, Mark Sutherland, Alexandros Daglis, and Babak Falsafi. Cerebros: Evading the RPC Tax in Datacenters. In *International Symposium on Microarchitecture*, 2021.

[30] Hailong Zhou, Jianji Dong, Junwei Cheng, Wenchan Dong, Chaoran Huang, Yichen Shen, Qiming Zhang, Min Gu, Chao Qian, Hongsheng Chen, et al. Photonic Matrix Multiplication Lights up Photonic Accelerator and Beyond. *Light: Science & Applications*.

[31] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. NanoFlow: Towards Optimal Large Language Model Serving Throughput, 2024. On Arxiv https://arxiv.org/abs/2408.12757.