Agile Hardware/Software Co-Design for Hyperscale Cloud Systems

Sagar Karandikar

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-222
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-222.html

December 19, 2024

Agile Hardware/Software Co-Design for Hyperscale Cloud Systems

by

Sagar Prashant Karandikar

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Emeritus and Professor of the Graduate School Krste Asanović, Chair
Professor Borivoje Nikolić
Associate Professor Yakun Sophia Shao
Dr. Parthasarathy Ranganathan, Vice President and Engineering Fellow, Google

Fall 2024

Agile Hardware/Software Co-Design for Hyperscale Cloud Systems

Abstract

Agile Hardware/Software Co-Design for Hyperscale Cloud Systems

by

Sagar Prashant Karandikar

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Emeritus and Professor of the Graduate School Krste Asanović, Chair

Global reliance on cloud services, powered by transformative technologies such as generative AI, machine learning, and big-data analytics, is driving exponential growth in demand for hyperscale cloud compute infrastructure. Meanwhile, the breakdown of classical hardware scaling (e.g., Moore's Law) is hampering growth in compute supply. Building domain-specific hardware can address this supply-demand gap, but catching up with exponential demand requires developing new hardware rapidly and with confidence that performance/efficiency gains will compound in the context of a complete system. These are challenging tasks given the status quo in hardware design, even before accounting for the immense scale of the cloud.

This dissertation focuses on two themes: (1) Developing agile, end-to-end HW/SW co-design tools that challenge the status quo in hardware design across system scale. (2) Leveraging these tools and hyperscale datacenter fleet profiling insights to architect/implement state-of-the-art domain-specific hardware to address key inefficiencies in hyperscale systems.

We first cover the FireSim FPGA-accelerated hardware simulation platform, which automatically constructs high-performance, cycle-exact, scale-out simulations of novel hardware designs derived from tapeout-friendly RTL, empowering hardware designers and domain experts alike to rapidly co-design systems. FireSim unlocks innovation in datacenter hardware with the ability to scale to massive, distributed simulations of specialized datacenter clusters. Next, we cover Chipyard, a platform for agile construction, evaluation, and tape-out of specialized RISC-V System-on-Chip (SoC) designs using an RTL-generator-driven approach.

We then cover Hyperscale SoC, a cloud-optimized server chip built, evaluated, and taped-out with FireSim/Chipyard. Hyperscale SoC includes several new domain-specific accelerators for expensive but foundational overheads in hyperscale servers, including (de)serialization, (de)compression, and more. This SoC demonstrates a new paradigm of data-driven, end-to-end HW/SW co-design, combining key insights from profiling Google's global datacenter fleet with the ability to rapidly build/evaluate novel HW/SW systems in FireSim/Chipyard.

Contents

List of Figures

List of Tables

Acknowledgments

I would first like to thank Krste Asanović, my Ph.D. advisor. Krste has been my advisor from the very beginning of my journey in computer architecture research, including when I was an undergraduate. My taste for building real systems and driving real impact is inspired by his example and high standards. Krste always gave me the freedom to focus on what I wanted and approach research the way I wanted, while always having the perfect advice at the right time.

Next, I would like to thank Parthasarathy Ranganathan. Partha has been my advisor/champion at Google since 2020 and is another member of my "advisory panel". Partha's insights and advice on how to make sure research stays grounded in the ability to drive impact at scale has been key to this work. Partha's relentless enthusiasm and support for big ideas and unconventional approaches to solving problems has also been an important guiding star.

I would also like to thank Borivoje Nikolić. Bora is the third member of my "advisory panel" and his go-big-or-go-home approach to projects that bridge the gap between academic research, industrial impact, and educational impact has been an important guiding vision. Bora can also always be relied on to resolve the "unknown unknowns"; he has given me lots of immensely useful advice that I didn't even know I needed at the time.

I've had the privilege of collaborating with several other faculty via research and teaching and I would like to thank them for graciously donating their time and wisdom: Yakun Sophia Shao (with a special thanks for agreeing to serve on my dissertation and quals committees), Dan Garcia, David Culler, David Patterson, John Wawrzynek, Jonathan Bachrach, Justin Hsia, Lisa Wu Wills, Miki Lustig, Rachit Agarwal, Randy Katz, Scott Shenker, Sylvia Ratnasamy, and Vladimir Stojanović.

The sense of community among students in architecture (UCB-BAR) and adjacent groups at Berkeley is one of its greatest strengths. The work in this dissertation simply would not have been possible without the brilliant group of student collaborators I had the privilege of working with. I would especially like to thank my "partners-in-crime" and closest collaborators in grad school: Alon Amid, David Biancolin, Abraham Gonzalez, Albert Ou, Howard Mao, Nathan Pemberton, and Jerry Zhao. I would also like to thank many other current/former Berkeley students, who helped make Berkeley a great place to do architecture research, including: Adam Izraelevitz, Aditya Chopra, Albert Magyar, Ameer Haj Ali, Andrew Lin, Andrew Waterman, Arya Reais-Parsi, Ben Keller, Ben Korpan, Brendan Sweeney, Brian Zimmer, Charles Hong, Christopher Celio, Coleman Hooper, Colin Schmidt, Daniel Grubb, David Bruns-Smith, David Kohlbrenner, Dayeol Lee, Deborah Soung, Dima Nikiforov, Dinesh Parimi, Donggyu Kim, Edwin Lim, Ella Schwarz, Emmanuel Amaro, Eric Love, Hansung Kim, Harrison Liew, Hasan Genc, Henry Cook, Jack Koenig, James Dunn, Jennifer Zhou, John Wright, Joonho Whangbo, Junsun Choi, Ken Ho, Kevin Anderson, Kevin Laeufer, Kris Dong, Kyle Kovacs, Lux Zhang, Martin Maas, Minwoo (Josh) Kang, Nayiri Krysztofowicz, Palmer Dabbelt, Paul Rigge, Prashanth Ganesh, Qijing (Jenny) Huang, Quan Nguyen, Raghav Gupta, Rimas Avižienis, Roger Hsiao, Sarah Zhou, Scott Beamer, Seah Kim, Shweta Shinde, Stephen Twigg, Stevo Bailey, Tianrui Wei, Tushar Sondhi, Vighnesh Iyer, Vikram Jain, Vrishab Madduri, Xingyu Li, Yufeng Chi, Yunsup Lee, and Zhangxi Tan.

I would also like to thank my collaborators and co-authors at (or formerly at) Google, including Ani Udipi, Chris Leary, Chris Kennelly, Svilen Kanev, Niket Agarwal, Victor Lee, and Jyrki

Alakuijala. The work in later parts of this dissertation also builds on profiling infrastructure work done by several engineering teams at Google (e.g., GWP, protobufz, and protodb) and I would like to thank current and former colleagues in those teams, including Darryl Gove, Martijn Vels, Chris Cummins, Todd Jackson, Garrett Wang, and Alexey Alexandrov. I would also like to thank Aamer Mahmood, Daniel Berlin, Don Stark, Jichuan Chang, Liqun Cheng, and Tipp Moseley for their feedback on drafts of the paper versions of later parts of this dissertation.

I would also like to thank David Culler, Stephen Dawson-Haggerty, and Andrew Krioukov for taking a chance on a first-semester undergraduate who wanted to do research. Working in the LoCal/Software-Defined Buildings group gave me my first taste of real, impactful systems research. Many of my "firsts" were part of this group, e.g., my first contribution to a research paper, my first time contributing to an open-source project that people use, and my first Berkeley retreats.

I also thank Yunsup Lee and Andrew Waterman (and Krste, again) for taking a risk on an undergraduate who at the time had only taken CS61C and wanted to work on computer architecture research. I learned important lessons about hands-on construction of hardware/software systems from Yunsup and Andrew; their specific advice lives on as part of UCB-BAR lore. As an undergraduate researcher in BAR, I had the opportunity to contribute to fundamental RISC-V infrastructure and be a part of the public "launch" of RISC-V at HotChips 2014, setting a high standard in my mind for what impact means and what it takes to achieve it.

I would also like to thank the ASPIRE/ADEPT/SLICE/BWRC/RISE/Sky staff, including Ria Briggs, Tamille Chouteau, Kostadin Ilov, Roxana Infante, Jim Lawson, Chick Markley, Brian Richards, Kattt Atchley, Shane Knapp, Jon Kuroda, and Ivan Ortega. Their friendliness and tireless work to keep the labs, events, compute infrastructure, and more functioning smoothly is another key piece of the fantastic research environment at Berkeley.

I also thank my friends outside of BAR/SLICE, including Akshay, Deepti, Michael, Michael, Paroma, Rohan, Shoumik, Stephanie, and Tyler. Thanks for all the fun times hanging out, eating, go-karting, arguing, playing various sports, and general shenanigans.

I would like to especially thank my family, including my parents, Prashant and Manjushree, and my brother Saarang. From when I was young, my parents always emphasized the importance of education and let me make my own decisions (such as going to college 2,500 miles from home no-questions-asked). My love of engineering and curiosity about how things work is thanks to them and the opportunities they gave me. I would also like to thank my in-laws, Prakash and Trupti, my sister-in-law Krishna, and my extended family and extended family-in-law for their support.

Finally, I would like to thank my wife, Sheevangi. Sheevangi has been along for the journey during my entire time at Berkeley, first as my best friend, then girlfriend, then fiancée, and now wife. She has been an amazing source of support through the uncertain times and the great times, and I couldn't ask for anyone better to have by my side. I can't put into words how much this wouldn't have been possible without her support.

* * *

The information, data, or work presented herein was funded in part by DARPA Award Number HR0011-12-2-0016, ARPA-E Award Number DE-AR0000849, NSF CCRI ENS Chipyard Award #2016662, NSF Award CCF-1955450, and SLICE/ADEPT/ASPIRE/RISE Lab industrial sponsors and affiliates Amazon Web Services, AMD/Xilinx, Apple, Futurewei, Google, HP, Intel, Qualcomm, Seagate, NVIDIA, and SK Hynix. Any opinions, findings, conclusions, or recommendations in this work are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

Chapter 1

Introduction

While generative artificial intelligence, machine learning, and big-data analytics are producing groundbreaking advances across disciplines, behind these advances is an *exponential increase in demand* for compute infrastructure, especially in *hyperscale datacenters* (or WSCs, *Warehouse-Scale Computers*). These hyperscale datacenters, operated by companies like Google, Amazon, Microsoft, Meta, Baidu, Alibaba, and Tencent, are the compute backbone of society. They host massive cloud services relied upon by billions of users and must do so in a *cost-efficient* and *environmentally sustainable* way. The exponential demand trends for these systems are visible across metrics: We see exponential growth in demand for *compute* driven by machine learning [55]. Global *internet traffic*, much of which is to or from these hyperscale datacenters, continues to grow exponentially [96]. The *volume of data* created, captured, copied, and consumed worldwide also follows a similar trend [207]. From a business perspective, we continue to see increased demand for cloud systems, with Gartner projecting a 4.5× increase in cloud-user enterprises from 2023 to 2027 [77]. At the same time, the computing industry is at a *once-in-a-generation inflection point*: Gone are the "rising tide lifts all boats" days of device scaling producing trends like Moore's law and Dennard scaling [87] that enabled exponential growth in the *supply* of compute to meet this demand.

Further compounding this shift and in stark contrast to the stagnation of individual server performance, datacenter network performance has continued to scale. Datacenter operators are currently deploying networks with 100s of Gbit/s of bandwidth at the leaves and latencies below 10s of microseconds. On the horizon is the potential for silicon photonic networks to push 100s of Terabits-per-second bandwidths straight to server processor dies [200]. New memory technologies such as HBM also have the potential to fill interesting gaps in the datacenter memory hierarchy, but also further deepen and complicate it, requiring detailed evaluation at scale. A large number of academic and industry groups have also pushed towards *disaggregated* datacenter architectures that combine all of these trends by splitting resources, including compute, high-performance storage, and memory across a high-bandwidth, low-latency datacenter fabric [140, 76, 20, 121, 90, 92, 98, 66]. Following these hardware trends and the expectations of modern web-scale service users, application and systems framework developers are also beginning to expect the ability to deploy fine-grained tasks, where task runtime and latency expectations are measured in microseconds [29].

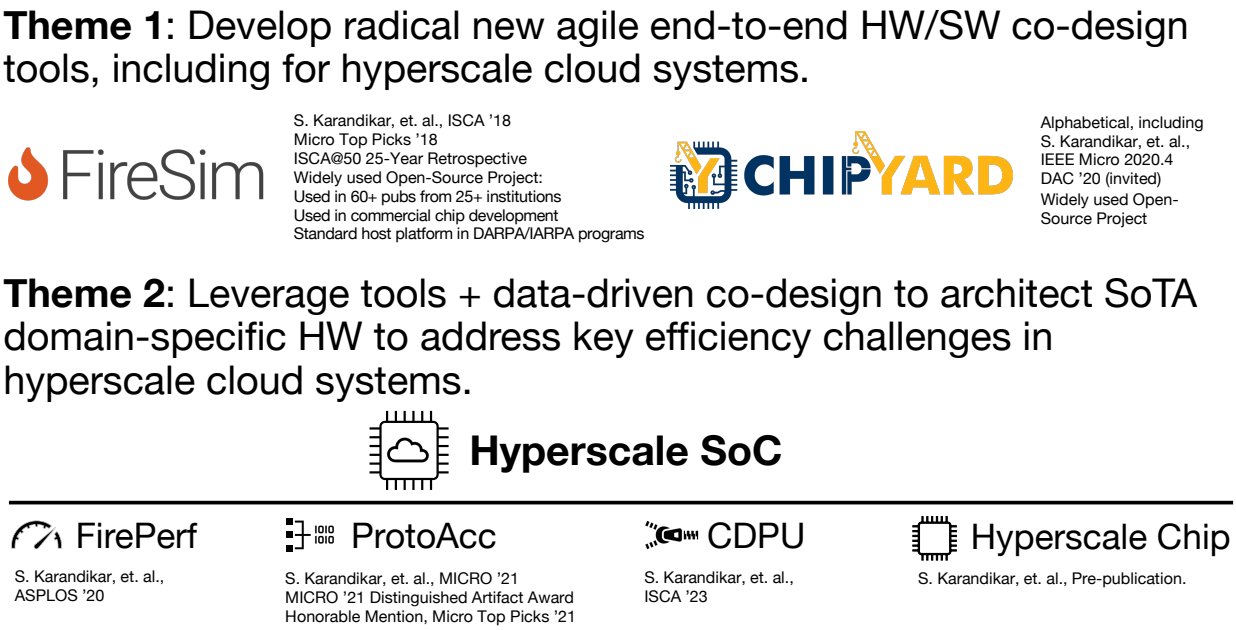


Figure 1.1: Overarching themes and outline of this dissertation.

Focusing more specifically on compute, while the first few generations of WSCs used off-the-shelf CPUs, the aforementioned compute supply-demand gap has pushed datacenter architects towards building compute engines that are increasingly specialized and vertically integrated [30, 131]. As a case in point, the increased efficiency provided by domain-specific hardware such as GPUs and TPUs has enabled the wide deployment of machine-learning pipelines in hyperscale contexts. This is neatly distilled in a quote from Google's original TPU paper [109], projecting datacenter capacity requirements if a TPU-like device did not exist:

> *"The conversation changed in 2013 when a projection where people use voice search for 3 minutes a day using speech recognition DNNs would require [Google's] data-centers to double to meet computation demands..."*
>
> —N. Jouppi, et. al., 2017 [109]

In fact, domain-specific hardware for machine learning has become so successful that despite the widespread integration of machine learning into Google's services, machine learning does not currently dominate Google's end-to-end energy consumption [164].

In modern WSCs, there is a clear need for hardware-software co-design to expand beyond the confines of a single system (one GPU, TPU, server, etc.). Machine-learning model sizes have rapidly outpaced the training and inference capabilities of individual GPUs, TPUs, and other ASICs for ML, requiring scale-out co-design. The general-purpose compute platforms that feed

vast amounts of data to these machine learning pipelines face similar problems; for example, datacenter server CPUs are plagued by system-level overheads that arise from the distributed nature of WSCs, such as the *datacenter taxes* [110, 198]. Further complicating this co-design effort, WSCs run large, layered software stacks consisting of diverse and rapidly evolving microservices, making specialization difficult.

In this dissertation, we address two critical roadblocks to eliminating the hyperscale datacenter capacity supply-demand gap: (1) How can we rapidly co-design hardware and software in scale-out contexts such as hyperscale datacenters? (2) How can we address key system-level overheads in WSCs (such as the datacenter taxes) to help close the WSC capacity supply-demand gap? Our approach to addressing these roadblocks and our key contributions are shown in Figure 1.1, which serves as a roadmap to the rest of this dissertation, while highlighting some of the wider impacts of this work.

1.1 Chasing exponentials in the post-Moore era with agile hardware design at-scale

Irrespective of domain, knowing what to specialize *for* can be challenging, as exemplified by the *hardware lottery* effect, which shows that historically *certain research ideas (e.g., new algorithms/ML models) succeed only because they are well-suited to available hardware and not because they are a global optimum* [89]. In the new world of vertically optimized domain-specific hardware, computer architects must collaborate across the stack to ensure the *best ideas succeed*, but doing this requires radical new ways to rapidly develop new hardware and assess the hardware implications of new algorithmic ideas.

To help us understand the scope of this methodological problem, Figure 1.2 shows an example of aggregate system-level performance improvement over time when trends like Moore's law and Dennard scaling were alive and well. In this world, aggregate performance improvement over time was a matter of compounding an achieved speedup of $N\times$ (e.g., $N = 2$) every p years (e.g., $p = 1.5$). Figure 1.2 emphasizes the importance of both of these terms. If, for example, our timeline slips from an improvement cadence of 1.5 years to 2 years, we lose over 65% of our aggregate performance improvement potential over the course of 10 years. Even worse, if achieved speedup in each generation drops from $2\times$ to $1.5\times$, over 85% of our aggregate performance improvement potential is lost over 10 years.

What do these terms mean in the new world of vertically optimized domain-specific hardware? First, achieved improvements must compound over time. This means that new optimizations and new domain-specific hardware blocks must demonstrate large wins *in the context of a complete system*, rather than in isolation. The second term implies that we must also produce these improvements at a rapid pace. Only the combination of these two will result in new hardware scaling trends that can keep pace with global demand for compute.

To achieve this, agile, end-to-end hardware/software co-design methodologies aim to revolutionize the process of architecting computer systems by emphasizing two goals: (1) minimizing

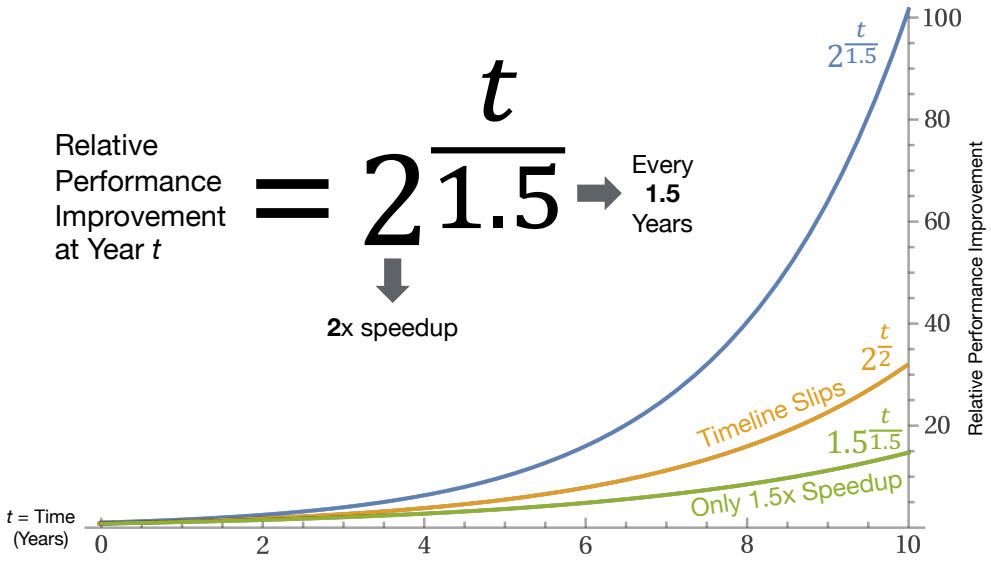


Figure 1.2: Performance improvement over time in the world of classical hardware scaling (e.g., Moore's law, Dennard scaling).

hardware development iteration time/cost, especially when working in small groups of researchers/engineers (the *agile*) and (2) developing/evaluating hardware and software in parallel, with a focus on their interplay in a complete system (the *end-to-end co-design*).

Effective end-to-end co-design requires pre-silicon collaboration between researchers and engineers working across the computing stack, including computer architecture, systems, compilers, very-large-scale integration (VLSI), and application domains. However, working with the single-source-of-truth describing a novel digital hardware design—the tapeout-friendly register transfer-level (RTL) implementation—was incompatible with the development and evaluation velocity needed to enable end-to-end full-system co-design, even for individual systems. Co-design for *hyperscale* systems was not possible.

In the first half of this dissertation, we address this methodological challenge by building FireSim and Chipyard, which together enable an agile hardware/software co-design flow that can scale all the way to hyperscale cloud systems.

FireSim: Fast simulation of novel datacenter designs with RTL-level fidelity

FireSim is an open-source (`https://fires.im`) FPGA-accelerated hardware simulation platform that automatically instruments and transforms tapeout-friendly RTL designs written in languages like Chisel or SystemVerilog into fast (10s to 100s MHz), deterministic, and cycle-exact FPGA-hosted simulators for productive pre-silicon verification and performance validation. Unlike FPGA prototypes, FireSim includes validated, performance-accurate models for common I/O interfaces like DRAM [34] and Ethernet [118] to enable modeling *complete systems*. To enable *agility* when

working with RTL designs, FireSim includes many debugging and profiling features not usually available on FPGAs.

Chapter 2 demonstrates FireSim's ability to scale to modeling *massive datacenter clusters* with RTL-level detail. There, we show FireSim modeling a networked 1024-node cluster hosted on 256 cloud FPGAs, where each node is a RISC-V server SoC written in tapeout-friendly RTL with a 200 Gbit/s Ethernet NIC and a complete memory system, including DRAM. Simulated servers are interconnected with a scalable, flexible Ethernet network model that provides both synchronization between the 1024 simulations and data exchange for faithful Ethernet modeling. This 1024-node simulation is sufficiently fast (10+ MHz) to, for example, boot Linux on each node and run WSC services (e.g., *memcached*) to evaluate end-to-end performance in the modeled datacenter and enable full-system end-to-end co-design at scale.

FireSim has been *used* (not only cited) in over 60 peer-reviewed publications from first authors at over 25 academic and industrial institutions. FireSim has also been used as a standard host platform for DARPA and IARPA programs and has been used in the development of commercially available chips. FireSim was also selected as an IEEE Micro Top Pick [117], selected for the ISCA@50 25-year Retrospective 1996-2020 [119], and nominated for CACM Research Highlights. Chapter 2.9 provides a detailed retrospective on FireSim and discusses key lessons learned.

Chipyard: Agile generation of RISC-V systems-on-chip (SoCs)

To facilitate an RTL-first approach to constructing new SoCs in agile teams, we discuss Chipyard (Chapter 3), an open-source design, evaluation, and implementation framework for specialized RISC-V SoCs. Chipyard enables researchers and engineers to productively compose complex systems from a large collection of parameterized, generator-based IP blocks such as in- and out-of-order RISC-V cores, uncore components, peripherals, accelerators, and more. Chipyard also includes software that runs on the generated SoCs, ranging from firmware, bootloaders, and bare-metal testing infrastructure to several compatible Linux distributions (e.g., Ubuntu). Users can customize any component of a Chipyard-generated system and push it through high-performance FPGA-accelerated simulation (FireSim), software simulation (e.g. Verilator and VCS), and automated ASIC flows (e.g. Hammer [214]) to productively iterate on specialized designs.

Like FireSim, Chipyard has become a widely used open-source project: the Chipyard repository on GitHub has over 650 public forks and more than 700 users and developers have joined Chipyard's mailing list to track development milestones and to ask and answer questions about Chipyard usage. To democratize access to agile co-design tools, we have run *nine* hands-on FireSim/Chipyard tutorials at academic conferences since 2019 and organized the first FireSim/Chipyard *Workshop* in 2023, a full-day event with *ten* talks from external FireSim/Chipyard users and developers. Chapter 3.5 provides a retrospective on Chipyard and highlights key lessons learned.

1.2 Addressing hyperscale datacenter inefficiencies with data-driven co-design of a cloud-optimized server system-on-chip (SoC)

Prior work has highlighted critical overheads that drastically reduce the efficiency of hyperscale cloud servers, such as the "datacenter taxes" [110, 198]. These datacenter taxes include fundamental primitives that are needed to enable scalable and distributed computation, such as serialization, compression, remote-procedure call, memory-movement operations (copies, moves, etc.), hashing, and memory allocation. Together, these have been shown to account for over 25% of CPU cycles in hyperscale server fleets [110], with several additional externalities that we will bring to light in later parts of this dissertation.

Addressing these overheads has previously been a challenge because they are deeply intertwined with the systems software stack. Accelerating these operations requires much finer-grained specialization as compared to other domains. This motivated the design of Hyperscale SoC (Chapter 4), a custom server system-on-chip optimized for the cloud.

As part of a multi-year industry-academic collaboration between Google and UC Berkeley, we architect, implement, evaluate, and ultimately tape-out Hyperscale SoC. Hyperscale SoC is based on Chipyard, configured with high-performance RISC-V MegaBOOM OoO application cores, a wide on-chip network to match existing hyperscale designs, and the 200 Gbit/s Ethernet NIC from FireSim [118]. Rather than relying on the coarse-grained acceleration opportunities enabled by traditional server integration (e.g., PCIe cards), Hyperscale SoC takes advantage of several forms of near-CPU accelerator integration to enable fine-grained offloading of key datacenter taxes.

A core thread tying together the Hyperscale SoC project is *rigorous, data-driven methodology*. Given the goal of building a highly specialized hardware/software system for the hyperscale cloud, naturally we must answer the questions alluded to earlier: (1) How do we know that we are building the *right* specialized hardware? (2) How do we ensure we are building a *complete solution* that allows us to understand the *end-to-end impact* of our novel architectural ideas?

To answer the first question, we must understand what specialized hardware should be deployed in next-generation datacenters. Given the ability to profile Google's worldwide hyperscale datacenter fleet[1], we are able to identify critical insights about the motivation and design of custom accelerators for the datacenter taxes and build and open-source hyperscaler-representative benchmarks for each domain.

To address the second question, we harness the agile hardware/software co-design tools discussed earlier to design and implement (in RTL) specialized hardware for the aforementioned domains and integrate it into Hyperscale SoC. We then run complete software stacks (including Google-representative benchmarks) on this design using FireSim. We also obtain ASIC quality-of-result data for our designs for a commercial 16nm-class FinFET process and eventually tape-out the design in this process. The following subsections will provide a brief overview of three key pieces of Hyperscale SoC.

[1]The work in this dissertation was partially done while being affiliated with both Google and UC Berkeley.

FirePerf: Agile cross-stack profiling and co-design for end-host networking

Bringing up a new server design like Hyperscale SoC and achieving performance comparable to modern commercial designs is a daunting task, given the need to improve not only core compute kernels, but also intricate and elusive system-level bottlenecks. Profiling these bottlenecks requires both high-fidelity introspection and the ability to run sufficiently many cycles to execute complex software stacks, a challenging combination. FirePerf (Chapter 5) addresses this systematically by developing and open-sourcing high-fidelity, out-of-band performance profiling features for FireSim that allow non-domain-experts to rapidly understand how to improve end-to-end system performance for a domain (in this case, networking) by avoiding many of the pitfalls of classical profiling tools. Having been upstreamed to FireSim, several FireSim users have written peer-reviewed publications that use the FirePerf tools and extend FirePerf's capabilities.

FirePerf enables us to quickly bring Hyperscale SoC's Linux userspace networking performance in line with commercial servers via HW/SW co-design, without consulting domain experts. With FirePerf, we identify/implement broadly applicable fixes to the Linux kernel for RISC-V (which also help existing commercial chips), specific improvements to the HW/SW interface between Linux and Hyperscale SoC's Ethernet NIC implementation, and demonstrate the effectiveness of hardware acceleration for *memory-to-memory copies and moves*, a key overhead in commercial hyperscale servers.

A hardware accelerator for protocol buffers

Serialization libraries like Google's Protocol Buffers (protobuf) define the *foundational interface* for flexible and robust data exchange in hyperscale systems, both between multiple services (via RPC) and between services and persistent storage. However, this flexibility and robustness, a must for highly available systems at scale, introduces significant compute overheads.

In Chapter 6, we present the first in-depth study of serialization library usage at scale by profiling protobuf usage in Google's hyperscale datacenter fleet. We show that nearly 10% of fleet CPU cycles are spent in protobufs and challenge several prevailing assumptions about serialization, including showing that PCIe-attached protobuf accelerators are likely infeasible due to memory access patterns and in-memory object sparsity. To support community research, we build and open-source a hyperscaler-representative (de)serialization benchmark, Google HyperProtoBench.

We then design an open-source hardware accelerator for protobuf, implemented in RTL and integrated into Hyperscale SoC near-CPU. Applications can easily harness the accelerator, as it sits behind a modified version of the open-source protobuf software library and is wire-compatible with standard protobufs. Closing the loop, we present an end-to-end evaluation of our entire RTL-based accelerated system running HyperProtoBench on Linux using FireSim. We demonstrate significant speedups compared to Xeon servers while consuming a small fraction of the silicon area. Our experiences have influenced industry, including various silicon vendors. This work also received the Distinguished Artifact Award at MICRO '21 and was selected as an IEEE Micro Top Picks 2021 Honorable Mention.

CDPU: Accelerating general-purpose lossless (de)compression at scale

General-purpose lossless data compression and decompression ("(de)compression") is unique amongst the datacenter taxes in that its purpose is not to add functionality, but to trade-off datacenter resources like CPU cycles, network bandwidth, and memory/storage capacity. Efficient hardware for (de)compression can radically change these trade-offs, but designing optimal hardware compression and decompression processing units ("CDPUs") is challenging due to the variety of algorithms deployed, input data characteristics, and evolving system-balance tradeoffs.

To navigate this vast design space, in Chapter 7 we present the first multi-year data-driven analysis of (de)compression usage at a major cloud provider by profiling Google's hyperscale datacenter fleet. We present several surprising insights, covering hardware implementation timelines, accelerator chaining challenges, and accelerator placement considerations. Most notably, while (de)compression already consumes 2.9% of fleet CPU cycles and 10-50% of cycles in key services, this demand is artificially limited. We identify that hyperscale services tend to *avoid using the best software compression techniques due to their high CPU cost*, a significant opportunity for hardware accelerators to save more than just CPU cycles.

While prior work has improved the microarchitectural state-of-the-art for CDPUs, we find that higher-level design parameters like CDPU placement, hash table sizing, and history window sizes have a more critical impact on the viability of CDPU integration but are not well-studied. To address this, we present the first end-to-end design/evaluation framework for CDPUs, including: 1. An open-source RTL-based CDPU generator that supports many run-time and compile-time parameters. 2. Integration into Hyperscale SoC for rapid performance and silicon area evaluation across CDPU placements and parameters. 3. An open-source (de)compression benchmark, HyperCompressBench, that is representative of (de)compression usage in Google's fleet.

We perform an extensive CDPU design space exploration, spanning a 46× range in CDPU speedup, 3× range in silicon area (for a single pipeline), and evaluate a variety of CDPU integration techniques to optimize CDPU designs for hyperscale contexts. Our final hyperscale-optimized CDPU instances are up to 10× to 16× faster than a single Xeon core, while consuming a small fraction (as little as 2.4% to 4.7%) of the area. Our experiences exploring CDPU design have influenced product design at Google and various silicon vendors.

Hyperscale chip

To demonstrate the *realizability* of Hyperscale SoC, we tape-out a 16 mm^2 Hyperscale chip in the Intel 16 process. The die contains *two* Hyperscale systems-on-chip networked together, each with an out-of-order RISC-V MegaBOOM core, protobuf (de)serialization accelerators, CDPUs, and 200 Gbit/s Ethernet NIC. While the chip has returned from fabrication, it is still being tested, so further details about the chip are not included in this dissertation.

1.3 Incorporation of previously published work

This dissertation incorporates content from the following previously published work:

1. ISCA 2018: "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud"
Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolić, Randy Katz, Jonathan Bachrach, and Krste Asanović
45th ACM/IEEE International Symposium on Computer Architecture (ISCA 2018)
https://doi.org/10.1109/ISCA.2018.00014

2. IEEE Micro 2019.3: Top Picks Issue: "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud"
Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolić, Randy Katz, Jonathan Bachrach, and Krste Asanović
IEEE Micro, vol. 39, no. 3, pp. 56-65, May-June 2019 (Micro Top Picks 2018 Issue)
https://doi.org/10.1109/MM.2019.2910175

3. ASPLOS 2020: "FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design"
Sagar Karandikar, Albert Ou, Alon Amid, Howard Mao, Randy Katz, Borivoje Nikolić, and Krste Asanović
25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020)
https://doi.org/10.1145/3373376.3378455

4. MICRO 2021: "A Hardware Accelerator for Protocol Buffers"
Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolić, Krste Asanović, and Parthasarathy Ranganathan
54th IEEE/ACM International Symposium on Microarchitecture (MICRO 2021)
https://doi.org/10.1145/3466752.3480051

5. ISCA@50 Retrospective: "RETROSPECTIVE: FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud"
Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolić, Randy Katz, Jonathan Bachrach, and Krste Asanović
ISCA@50 25-Year Retrospective: 1996-2020, Ed. José F. Martínez and Lizy K. John
https://sites.coecis.cornell.edu/isca50retrospective/files/2023/06/Karandikar_2018_FireSim.pdf

6. ISCA 2023: "CDPU: Co-designing Compression and Decompression Processing Units for Hyperscale Systems"
Sagar Karandikar, Aniruddha N. Udipi, Junsun Choi, Joonho Whangbo, Jerry Zhao, Svilen Kanev, Edwin Lim, Jyrki Alakuijala, Vrishab Madduri, Yakun Sophia Shao, Borivoje Nikolić, Krste Asanović, and Parthasarathy Ranganathan
50th ACM/IEEE International Symposium on Computer Architecture (ISCA 2023)
https://doi.org/10.1145/3579371.3589074

Chapter 2

FireSim: Fast simulation of novel datacenter designs with RTL-level fidelity

In this chapter, we discuss the first component of our agile hardware/software co-design flow for hyperscale systems research, FireSim. FireSim is an open-source FPGA-accelerated hardware simulation platform that can scale to modeling novel datacenter designs, using customizable server RTL designs as ground truth. After discussing FireSim's design, we conclude the chapter by discussing FireSim's impact as an open-source project and highlight key lessons learned.

2.1 Introduction

The trends discussed in Chapter 1 push the boundaries of hardware-software co-design at-scale. Architects can no longer simply simulate individual nodes and leave the issues of scale to post-silicon measurement. Additionally, the introduction of custom silicon in the cloud means that architects must model emerging hardware, not only incremental enhancements to well-understood processor microarchitectures. Hardware-software co-design studies targeting next-generation WSCs are hampered by a lack of a scalable and performant simulation environment. Using microarchitectural software simulators modified to scale out [160, 154] is fundamentally bottlenecked by the low simulation speeds (5–100 KIPS) of the underlying single-server software simulator. Fast custom-built simulation hardware has been proposed [203], but is difficult to modify and expensive ($100k+) to acquire, which limits access by most academic and industrial research teams.

In this chapter, we present FireSim[1,2], an open-source, cycle-exact, FPGA-accelerated simulation framework that can simulate large clusters, including high-bandwidth, low-latency networks, on a public-cloud host platform. Individual nodes in a FireSim simulation are automatically derived from synthesizable RTL and run realistic software stacks, including booting Linux, at 10s to 100s of MHz. High-performance C++ switch models coordinate simulation globally and provide clean abstractions that hide the host system transport to allow users to define and experiment with their

[1] `https://fires.im`

[2] `https://github.com/firesim`

own switching paradigms and link-layer protocols. FireSim also automates the construction of a scale-out simulation—users define the network topology and number and types of server blades in the system being simulated, and FireSim builds and deploys high-performance simulations on Amazon EC2 F1 instances. Users can then treat the simulated nodes as if they were part of a real cluster—simulated datacenter nodes are visible on the network and can be accessed via SSH to run software while collecting performance data that is cycle-exact. Thus, a FireSim user is isolated from the complications of running FPGA-accelerated simulations. They write RTL (which can later be synthesized with CAD tools and possibly taped out) to customize their datacenter blades, C++ code to customize switch designs, and specify the topology and link characteristics of their network simulation to the simulation manager, which then builds and deploys a simulation. Only RTL changes require re-running FPGA synthesis—network latency, bandwidth, network topology, and blade selection can all be configured at runtime.

Chapter 2.2 describes recent hardware trends that allow us to build a fast, usable, and cost-effective hardware simulation environment. Chapter 2.3 describes the FireSim simulation environment, including the target designs FireSim is capable of simulating, and our high-performance network simulation infrastructure. Chapter 2.4 validates data collected by software in FireSim simulations against parameters set in the FireSim configuration, as well as reproducing scale-out system phenomena from recent literature. Chapter 2.5 discusses simulation performance and scale, including a 1024-node simulation. Chapter 2.6 describes some early work in warehouse-scale hardware-software co-design that uses the FireSim simulation platform and discusses preliminary results. Finally, Chapter 2.7 discusses prior work in the area of scale-out system simulation and contrasts it with FireSim.

2.2 Harnessing FPGAs in the public cloud

Architects experience many challenges when building and using FPGA-accelerated simulation platforms. FPGA platforms are unwieldy, especially when compared to software simulators that run on commodity compute platforms. Traditional FPGA platforms are constrained by high prices, individual platform quirks that make reproducibility difficult, the need to provision for maximum utilization at time of initial purchase, and long build times, where parallelism is limited by the number of build licenses and build servers available. Even when FPGA pricing is insignificant to a project, building a custom rack of large FPGAs requires significant systems management and operations experience and makes it extremely difficult to share and reproduce research prototypes.

But recently, many cloud providers have begun integrating FPGAs into their cloud services, including Amazon [11], Microsoft [44, 175], Huawei [93], and Alibaba [10]. In particular, Amazon makes FPGAs available as part of its *public* cloud offering, allowing developers to directly design FPGA-based applications that run in the cloud. Using an FPGA-enabled public cloud platform such as EC2 F1 addresses many of the traditional issues with FPGA-based hardware simulation platforms and provides many of the same benefits to computer architects that the cloud brought to distributed systems builders. At the dawn of the cloud era, systems researchers identified several changes to the economics of obtaining compute: (1) the new illusion of infinite computing

resources, (2) the elimination of up-front commitment towards hardware resources, and (3) the ability to scale resources on-demand [18]. Given that prices of large FPGAs are much higher than even the most expensive general-purpose-compute servers, these advantages are magnified for developers and users of FPGA-based simulation platforms.

Since EC2 F1 is a relatively recent offering, we summarize some of the key features of the service to explain why it forms a natural platform on which to build the scalable FireSim environment. Amazon's EC2 F1 offering provides two new EC2 instance types, `f1.2xlarge` and `f1.16xlarge`, which consist of a powerful host instance (8 or 64 vCPUs, 122 or 976 GB of DRAM, 10 or 25 Gbit/s networking) attached to 1 or 8 Xilinx Virtex UltraScale+ FPGAs over PCIe. Furthermore, each FPGA contains 64 GB of DRAM onboard across 4 channels, making it an ideal platform for prototyping servers. The ability to provision any number of these instances on-demand and distribute work to them provides scalability. Chapter 2.3 describes FireSim's ability to automate the mapping and deployment of a simulation across a large number of `f1.2xlarge` and `f1.16xlarge` instances.

Amazon also provides an "FPGA Developer AMI", an OS image that contains all of the tooling and licenses necessary to produce FPGA images for F1 instances pre-installed. As with FPGAs themselves, users can now scale to an essentially unlimited number of FPGA synthesis/P&R machines, making it possible to parallelize builds for design-space exploration and for constructing heterogeneous cluster simulations. Chapter 2.3 describes the FireSim infrastructure that can automatically distribute FPGA image builds based on a set of supplied node configurations.

In addition to F1 instances, FireSim also uses `m4.16xlarge` instances, which are "standard" EC2 instances that support high-performance (25 Gbit/s) networking. FireSim uses these instances to run aggregation and root switch models. Altogether, by taking advantage of the scalability of a cloud FPGA platform, we demonstrate the ability to automatically generate, deploy, and simulate a cluster of 1024 quad-core server nodes (for a total of 4096 cores) interconnected by a 200 Gbit/s network with 2 μs latency at 3.4 MHz. In aggregate, this simulation runs ˜14 billion instructions per second and harnesses 12.8 million dollars' worth of FPGAs, at a total cost of only $100 per simulation hour[3] to the user with no upfront capital expenditure. Chapter 2.5 details this example FireSim instantiation.

2.3 FireSim design and internals

FireSim models a target system containing a collection of server blades connected by some form of network. The target server blades are modeled using FAME-1 models [202] automatically derived from the RTL of the server SoCs and mapped onto FPGA instances, while the target network is modeled with high-performance, cycle-by-cycle C++ switch models running on host server instances. These two target components are interconnected by a high-performance simulation-token transport that models target link characteristics and abstracts away host platform details, such as PCIe communication and host-to-host Ethernet communication. Figures 2.1 and 2.2 show the tar-

[3] Dollar amounts cited in this chapter are 2018 numbers.

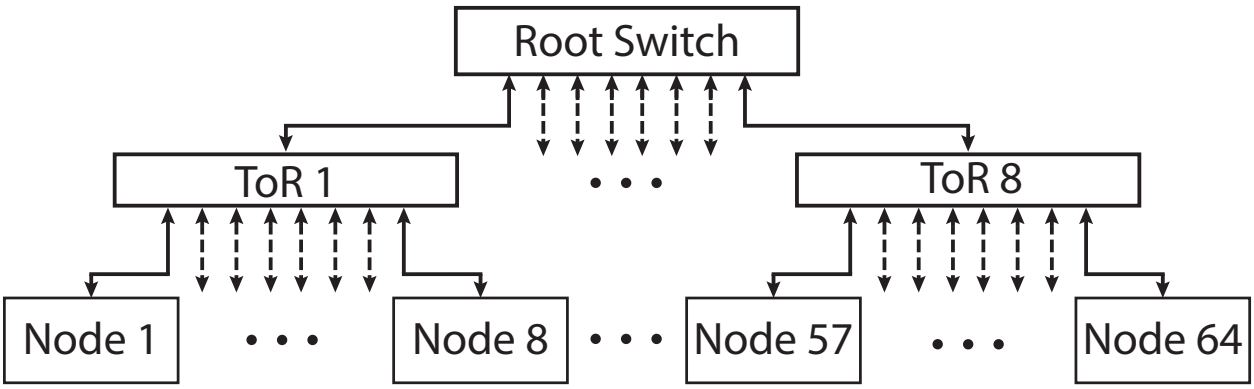


Figure 2.1: Target view of the 64-node topology simulated in Figure 2.2.

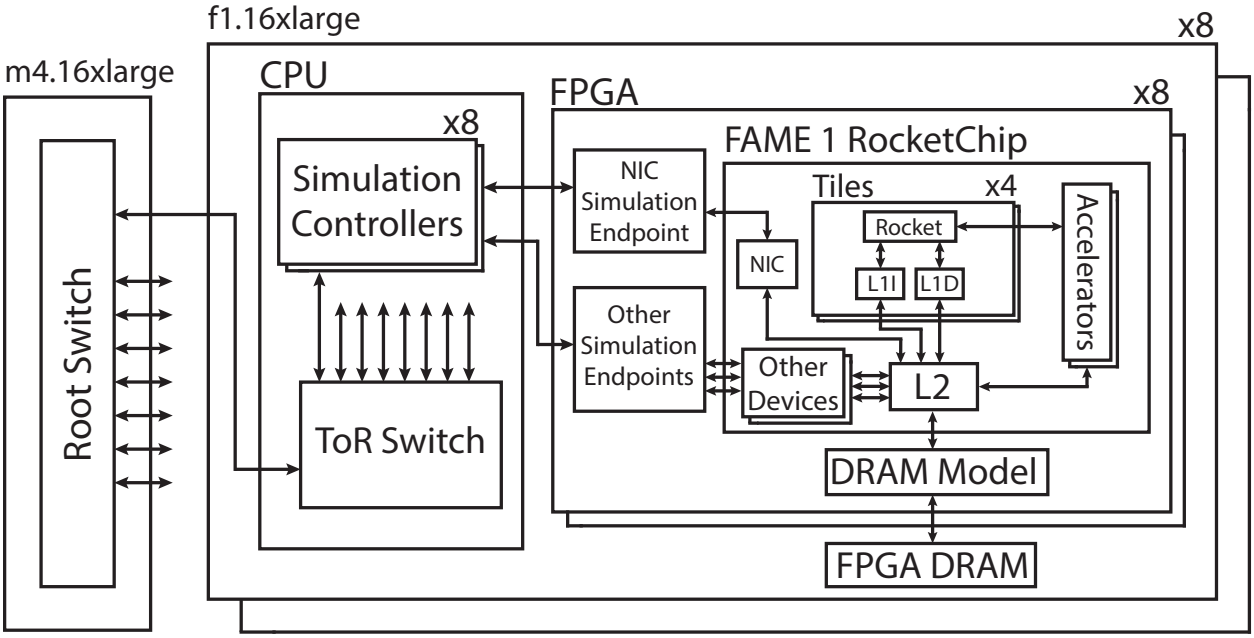


Figure 2.2: Example mapping of a 64-node simulation to EC2 F1 in FireSim.

get topology and target-to-host mapping respectively for a 64-node simulation with 8 top-of-rack (ToR) switches and one root switch, which we use as an example throughout this section.

Server Blade Simulation

Target Server Design

FireSim compute servers are derived from the Rocket Chip SoC generator [22], which is an SoC generation library written in Chisel [25]. Rocket Chip can produce Verilog RTL for a complete

Table 2.1: Server blade configuration.

Blade Component	RTL or Model
1 to 4 RISC-V Rocket Cores @ 3.2 GHz	RTL
Optional RoCC Accel. (Table 2.2)	RTL
16 KiB L1I$, 16 KiB L1D$, 256 KiB L2$	RTL
16 GiB DDR3	FPGA Timing Model
200 Gbit/s Ethernet NIC	RTL
Disk	Software Model

Table 2.2: Example accelerators for custom blades.

Accelerator	Purpose
Page Fault Accel.	Remote memory fast-path (Chapter 2.6)
Hwacha [137, 136, 135]	Vector-accelerated compute
HLS-Generated	Rapid custom scale-out accels.

processor system, including the RISC-V Rocket CPU, L1 and L2 caches, custom accelerators (Table 2.2), and I/O peripherals. Table 2.1 shows the Rocket Chip configurations we use throughout this work. When we refer to a particular frequency f for Rocket Chip, for example 3.2 GHz in Table 2.1, this implies that all models that require a notion of target time in the simulation (e.g., the network) assume that 1 cycle is equivalent to $1/f$ seconds. The "FAME-1 Rocket Chip" box in Figure 2.2 provides a sample block diagram of a Rocket Chip server node.

Target Server Network Interface Controller

To build complete server nodes, we add two new peripherals to the Rocket Chip SoC. The first is a network interface controller (NIC) written in Chisel that exposes a top-level network interface on the SoC. The design of the NIC is shown in Figure 2.3. The NIC sends and receives Ethernet packets to/from the network switch. Recent works have called for CPU and NIC to be integrated on the same die in order to decrease communication latency [184]. Our NIC implements this approach and connects directly to the on-chip network of the Rocket Chip SoC through the TileLink2 interconnect [191]. This allows the NIC to directly read/write data in/out of the shared L2 on the server SoC (Figure 2.2).

The NIC is split into three main blocks: the controller, the send path, and the receive path (Figure 2.3). The controller in the NIC is responsible for communicating with the CPU. It holds four queues: send request queue, receive request queue, send completion queue, and receive completion queue. The queues are exposed to the processor as memory-mapped IO registers. To send a packet out through the NIC, the CPU writes the memory address and length of the packet to the send request queue. To receive a packet from the NIC, the CPU writes the address of the receive buffer to the receive request queue. When the send/receive paths finish processing a request, they add an

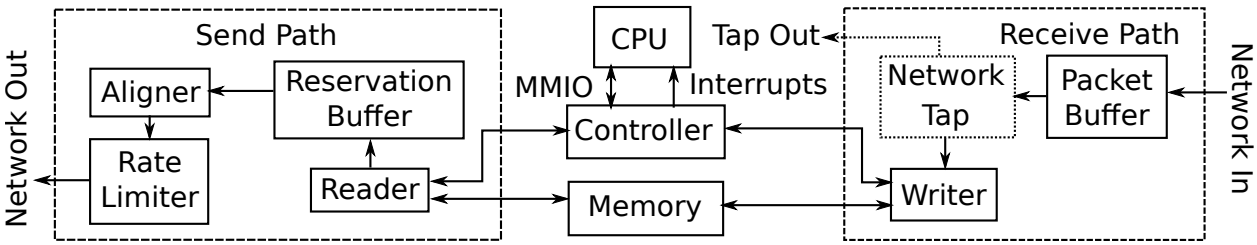


Figure 2.3: Network Interface Controller (NIC) design.

entry to their respective completion queues. The NIC also has an interrupt line to the CPU, which it asserts when a completion queue is occupied. The interrupt handler then reads the completion entries off of the queue, clearing the interrupt.

The send path in the NIC begins with the reader module, which takes a send request from the controller and issues read requests for the packet data from memory. Responses from memory are forwarded to the next stage, the reservation buffer. The reader sends a completion signal to the controller once all the reads for the packet have been issued.

The reservation-buffer module stores data read from memory while it waits to be transmitted through the network interface. Responses from the memory bus can arrive out-of-order, so the reservation buffer also performs some reordering so that the data is sent to the next stage in-order.

After the reservation buffer comes the aligner, which allows the send path to handle unaligned packets. The interface to the memory system is 64 bits wide, so the reader can only read data at an eight-byte alignment. If the starting address or ending address of the packet is not a multiple of eight, some extra data before or after the packet will be read. The aligner shifts data coming from the buffer so that the extra data is omitted, and the first byte of the packet will be the first byte delivered to the destination.

The final module in the send path is the rate limiter, which allows NIC bandwidth to be limited at runtime using a token-bucket algorithm. Essentially, the limiter holds a counter that is decremented every time a network flit is sent and incremented by k every p cycles. Flits can be forwarded from input to output so long as the count is greater than zero. This makes the effective bandwidth $\frac{k}{p}$ times the unlimited rate. The values k and p can be set at runtime, allowing simulation of different bandwidths without resynthesizing the RTL. Unlike external throttling of requests, this internal throttling appropriately backpressures the NIC, so it behaves as if it actually operated at the set bandwidth.

The receive path begins at the network input with a packet buffer. Since we cannot backpressure the Ethernet network, the buffer must drop packets when there is insufficient space. Packets are only dropped at full-packet granularity so that the operating system never sees incomplete packets.

The writer module takes packet data from the buffer and writes it to memory at the addresses provided by the controller. The writer sends a completion to the controller only after all writes for the packet have retired.

To interface between user-space software and the NIC, we wrote a custom Linux driver to allow any Linux-compatible networking software to run on FireSim.

The top-level interface of the NIC connects to the outside world through a FAME-1 decoupled interface—each cycle, the NIC must receive a token and produce a token for the simulation to advance in target time. Chapter 2.3 details cycle-accurate packet transport outside of the NIC.

Target Server Block-Device Controller

We also add a block-device controller to the server blades simulated in FireSim to allow booting of custom Linux distributions with large root filesystems. The block-device controller contains a frontend module that interfaces with the CPU and one or more trackers that move data between memory and the block device. The frontend exposes memory-mapped I/O (MMIO) registers to the CPU, through which it can set the fields needed for a block-device request. To start a block device transfer, the CPU reads from the allocation register, which sends a request to one of the trackers and returns the ID of the tracker. When the transfer is complete, the tracker sends a completion to the frontend, which records the ID of the tracker in the completion queue and sends an interrupt to the CPU. The CPU then reads from the completion queue and matches the ID with the one it received during allocation. The block device is organized into 512-byte sectors, so the controller can only transfer data in multiples of 512 bytes. The data does not need to be aligned at a 512-byte boundary in memory, but it does need to be aligned on the block device.

Cycle-Exact Server Simulations from RTL

We use the FAME-1 [202] transforms provided by the MIDAS/Strober frameworks [125, 124] to translate the server designs written in Chisel into RTL with decoupled I/O interfaces for use in simulation. Each target cycle, the transformed RTL on the FPGA expects a token on each input interface to supply input data for that target cycle and produces a token on each output interface to feed to the rest of the simulated environment. If any input of the SoC does not have an input token for that target cycle, simulation stalls until a token arrives. This allows for timing-accurate modeling of I/O attached to custom RTL on the FPGA. To provide a cycle-accurate DRAM model for our target servers, we use an existing synthesizable DRAM timing model provided with MIDAS, attached directly to each host FPGA's on-board DRAM, with parameters that model DDR3. Other I/O interfaces (UART, Block Device, NIC) communicate with a software driver ("simulation controller" in Figure 2.2) on the host CPU core, which implements both timing and functional request handling (e.g. fetching disk blocks). Since in this work we are primarily interested in scaling to large clusters and network modeling, we focus on the implementation of the network token-transport mechanism used to globally coordinate simulation target time between the FAME-1-transformed server nodes.

Improving Scalability and Utilization

In addition to the previously described configuration, FireSim includes an additional "supernode" configuration, which simulates multiple complete target designs on each FPGA to provide im-

proved utilization and scalability.

The basic target design described above utilizes only 32.6% of the FPGA LUT resources and one of the four memory channels. Only 14.4% of this utilization was for our custom server-blade RTL. As a result, the supernode configuration packs four simulated nodes on each FPGA, increasing FPGA LUT utilization for server blades to approximately 57.7% of the FPGA logic resources, and total FPGA LUT utilization to 76%. This optimization reduces the cost of simulating large clusters, at the cost of multiplexing network token transfer for four nodes over a single PCIe link. Chapter 2.5 describes how supernodes support the simulation of a large cluster with 1024 nodes.

This type of scalability is particularly important when attempting to identify datacenter-level phenomena and reduces the dependency on the cloud-provider's instantaneous FPGA instance capacity. Furthermore, this capability allows for the simulation of smaller target network topologies, such as connecting a ToR switch to 32 simulated nodes, without an expensive host-Ethernet crossing for token transport.

Network Simulation

Target Switch Modeling

Switches in the target design in FireSim are modeled in software using a high-performance C++ switching model that processes network flits cycle-by-cycle. The switch models have a parametrizable number of ports, each of which interact with either a port on another switch or a simulated server NIC on the FPGA. Port bandwidth, link latency, amount of buffering, and switching latency are all parameterized and runtime configurable.

The simulated switches implement store-and-forward switching of Ethernet frames. At ingress into the switch, individual simulation tokens that contain valid data are buffered into full packets, timestamped based on the arrival cycle of their last token, and placed into input packet queues. This step is done in parallel using OpenMP threads, with one thread per-port. The timestamps are also incremented by a configurable minimum switching latency to model the minimum port-to-port latency of datacenter switches. These timestamps are later used to determine when a packet can be released to an output buffer. A global switching step then takes all input packets available during the switching round, pushes them through a priority queue that sorts them on timestamp, and then drains the priority queue into the appropriate output port buffers based on a static MAC address table (since datacenter topologies are relatively fixed). In this step, packets are duplicated as necessary to handle broadcasts. Finally, in-parallel and per-port, output ports "release" packets to be sent on the link in simulation-token form, based on the switch's notion of simulation time, the packet timestamp, and the amount of available space in the output token buffer. In essence, packets can be released when their release timestamp is less than or equal to global simulation time. Since the output token buffers are of a fixed size during each iteration, congestion is automatically modeled by packets not being able to be released due to full output buffers. Dropping due to buffer sizing and congestion is also modeled by placing an upper bound on the allowable delay between a packet's release timestamp and the global time, after which a packet is dropped. The switching algorithm described above and assumption of Ethernet as the link-layer is not fundamental to

FireSim—a user can easily plug in their own switching algorithm or their own link-layer protocol parsing code in C++ to model new switch designs.

High-performance Token Transport

Unlike simulating "request-response" style channels (e.g. AXI channels) in cycle-accurate simulation platforms, the decoupled nature of datacenter networks introduces new challenges and prevents many optimizations traditionally used to improve simulation performance, especially when simulated nodes are distributed as in FireSim. In this section, we describe our network simulation and how we map links in simulation to the EC2 F1 host platform.

From the target's view, endpoints on the network (either NICs or ports on switches) should communicate with one another through a link of a particular latency and bandwidth. On a simulated link, the fundamental unit of data transferred is a token that represents one target cycle's worth of data. Each target cycle, every NIC expects one input token and produces one output token in response. Each port on every switch also behaves in the same way, expecting one input token and generating one output token every cycle. For a link with link latency of N cycles, N tokens are always "in-flight" on the link at any given time. That is, if a particular network endpoint issues a token at cycle M, the token arrives at the other side of the link for consumption at cycle $M+N$.

Network tokens in FireSim consist of two components: the target payload field and a "last" metadata field that indicates to the transport that a particular token is the end of a packet, without having to understand a particular link-layer protocol. In the case of our Ethernet model, the target payload field contains two fields: the actual data being transmitted by the target design during that cycle and a valid signal to indicate that the input data for this cycle is legitimate data (as opposed to an empty token, which corresponds to a cycle where the endpoint received nothing from the network). To simulate the 200 Gbit/s links we use throughout this work, the width of the data field is set to 64 bits, since we assume that our simulated processor frequency is 3.2 GHz. In a distributed simulation as in FireSim, different host nodes are decoupled and can be executing different target cycles at the same time, but the exchange of these tokens ensures that each server simulation computes each target cycle deterministically, since all NICs and switch ports are connected to the same network and do not advance unless they have input tokens to consume.

In a datacenter topology, there are two types of links that need to be modeled: links between a NIC and a switch port and links between two switch ports on different switches. Since we model switches in software and NICs (and servers) on FPGAs, these two types of links map to two different types of token transport. Transport between NICs and switch models requires two hops: a token must first cross the PCIe interface to an individual node's simulation driver, then be sent to a local switch model through shared memory or a remote switch model over a socket.

As discussed in prior work on distributed software-based cluster simulation [154], batching the exchange of these tokens improves host bandwidth utilization and hides the data movement latency of the host platform—both PCIe and Ethernet in the case of EC2 F1. Tokens can be batched up to the target's link latency, without any compromise in cycle accuracy. Given that the movement of network tokens is the fundamental bottleneck of simulation performance in a FireSim simulation, we always set our batch size to the target link latency being modeled.

We utilize three types of physical transports to move tokens. Communication between FPGAs and host CPUs on EC2 F1 happens over PCIe. For high-performance token transport, we use the Amazon Elastic DMA (EDMA) interface to move batches of tokens (one link latency's worth) between token queues on the FPGA and the simulation controller on the host. Transport between the simulation controller and a ToR switch or between two switches can be done either through a shared memory port (to effectively provide zero-copy transfer between endpoints) or a TCP socket when endpoints are on separate nodes. Since we are focused on simulating low-latency target networks, our primary bottleneck in simulation is the host latency of these transport interfaces. Since latency is the dominant factor, we also do not employ any form of token compression. This also means that simulation performance is stable (workload independent), apart from the cost of the switching phase inside a switch. We explore the trade-off between target-link latency and simulation performance in Chapter 2.5.

To provide a concrete example to tie together the physical and logical layers in link modeling and token transport, let us trace the progress of a packet moving between two servers (labeled A and B) connected by a single ToR switch in simulation. For simplicity, we assume that the packet consists of only one token, however this description naturally extends to longer packets. We assume that links have a latency of l cycles and that we batch token movement by always moving l tokens at a time across host PCIe/network links. We also assume that the network is completely unloaded and that switches have a minimum port-to-port latency of n cycles. Within a server, cycle-accuracy is maintained by virtue of directly running FAME-1-transformed RTL on an FPGA, so we focus on the path between the top-level I/O of the two server NICs that are communicating:

1. Suppose that all simulated components (the switch and the two servers) begin at cycle $t = 0$, with each input token queue initialized with l tokens.

2. Each simulated component can independently advance to cycle $t = l$ by consuming the input tokens. Suppose that server A's NIC has a valid packet to send at cycle $t = m < l$.

3. This packet is written into the output token buffer in the NIC Simulation Endpoint (see Figure 2.2) on the FPGA at index m. When server A has completely filled the buffer, the buffer is copied first to the software simulation controller over PCIe and then to the ToR switch via shared memory.

4. In the meantime, since the switch was also initially seeded with l tokens per port, its simulation time has also advanced to cycle l, before it receives this buffer.

5. Now, when the switch "ticks" cycle-by-cycle through the newly received buffer and reaches simulation time $t = l + m$, it will "receive" the packet sent by server A.

6. Next, the switch will write the packet to an output buffer after a minimum switching delay, n, at cycle $t = l + m + n$. For the sake of argument, assume $m + n < l$. Then, this packet will be written to the next buffer sent out by the switch.

7. In the meantime, server B will have received two rounds of input buffers, so it will have advanced to cycle $2l$ when it receives the buffer containing the packet. Since the packet is at index $m + n$ in this buffer, it will arrive at the input of the server's NIC at cycle $2l + m + n$, or two times the link latency, plus the switching latency, plus the time at which server A sent the packet.

This decoupled simulation technique allows us to scale easily to large clusters. Furthermore, simulations generally map well onto our host platform, since we are in essence simulating large

```
m4_16xlargeIPs = [...]
f1_16xlargeIPs = [...]
root = SwitchNode()
level2switches = [SwitchNode() for x in range(8)]
servers = [[ServerNode("QuadCore")
                                for y in range(8)]
                                for x in range(8)]
root.add_downlinks(level2switches)

for l2switchNo in range(len(level2switches)):
    switch = level2switches[l2switchNo]
    servers = servers[l2switchNo]
    switch.add_downlinks(servers)
```

Figure 2.4: Example simulation configuration. This instantiates a simulation of the cluster topology shown in Figure 2.1 with quad-core servers.

target clusters on large host clusters. Unlike software simulators, the power of our host CPUs can be dedicated to fast network switching, while FPGAs handle the computationally expensive task of modeling the low-level details of the processors and the rest of the server blades.

Deploying/Mapping Simulation to EC2 F1

At this point, we have outlined each component necessary to build a large-scale cluster simulation in FireSim. However, without automation, the task of stitching together all of these components in a reliable and reproducible way is daunting. To overcome this challenge, the FireSim infrastructure includes a simulation manager that automatically builds and deploys simulations given a programmatically defined datacenter topology. That is, a user can write a configuration in Python that describes a particular datacenter topology and server types for each server blade as shown in Figure 2.4. The FireSim cluster manager takes this configuration and automatically runs the desired RTL through the FPGA build flow and generates the high-performance switch models and simulation controllers with the appropriate port implementations (shared memory, socket, PCIe transport). In particular, based on the given topology, simulated servers are automatically assigned MAC and IP addresses and the MAC switching tables in the switch models are automatically populated for each switch in the simulation. Once all component builds are complete, the manager flashes FPGAs on each F1 instance with the desired server configurations, deploys simulations and switch models as described by the user, and finally boots Linux (or other software) on the simulated nodes. At the root switch, a special port can be added to the network that allows for direct ingress into the simulated datacenter network over SSH. That is, a user can *directly ssh* into the simulated system from the host machine and treat it as if it were a real cluster to deploy programs

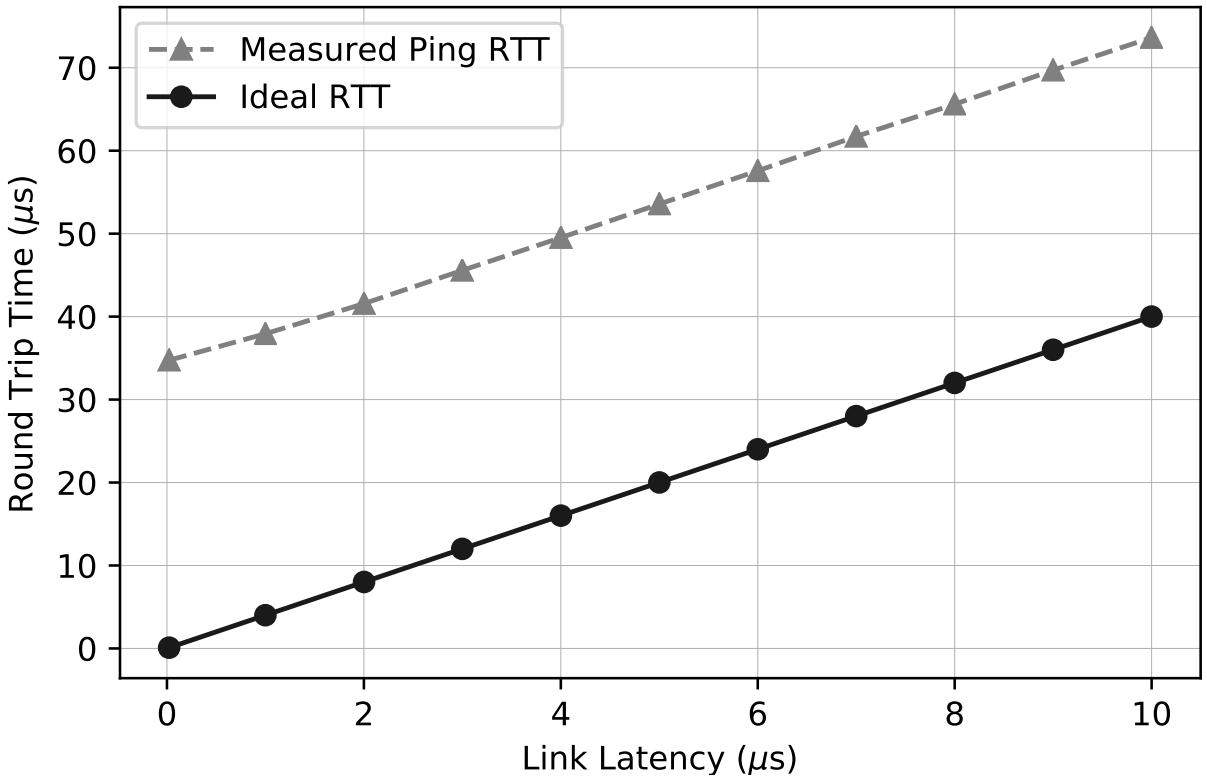


Figure 2.5: Ping latency vs. configured link latency.

and collect results. Alternatively, a second layer of the cluster manager allows users to describe jobs that *automatically* run on the simulated cluster nodes and *automatically* collect result files and host/target-level measurements for analysis outside of the simulation. For example, the open release of FireSim includes reusable workload descriptions used by the manager to automatically run various versions of SPECint, boot other Linux distributions such as Fedora, or reproduce the experiments described later in this chapter, among others.

2.4 Validating FireSim's modeling fidelity

In this section, we validate the FireSim simulation environment, in particular our high-performance, cycle-accurate network simulation, using several benchmarks.

Network Latency Benchmarking

We benchmark end-to-end latency between simulated nodes by collecting samples from the ping utility in Linux, while varying the target link latency set in simulation. This experiment boots Linux

on a simulated 8-node cluster connected by a single ToR switch and collects the results of 100 pings between two nodes. We ignore the first ping result on each boot, since this includes additional latency for an ARP request for the node to find the MAC address of the node being pinged. We then vary the configured link latency for the simulation and record the average RTT reported by ping. Figure 2.5 shows the results of this benchmark. The bottom line represents the "ideal" round trip time (link latency times four, plus a fixed port-to-port switching latency of 10 cycles times two). As expected for a correct network simulation, the measured results parallel the ideal line, with a fixed offset that represents overhead in the Linux networking stack and other server latency. The $\approx$34 μs overhead we observe matches results widely reported in the OS literature [184].

Network Bandwidth Benchmarking: `iperf3` on Linux

Next, we run `iperf3`, a standard network benchmarking suite, on top of Linux on the simulated nodes and measure the bandwidth between two nodes connected by a ToR switch. In this benchmark, we measure an average bandwidth over TCP of 1.4 Gbit/s. While this is much lower than our nominal link bandwidth of 200 Gbit/s, we suspect that the bulk of this mismatch is due to the relatively slow single-issue in-order Rocket processor running the network stack in software on an immature RISC-V Linux port with our Linux network driver.

Bare-metal node-to-node bandwidth test

To separate out the limits of the software stack from our NIC hardware and simulation environment, we implemented a bare-metal bandwidth benchmarking test that directly interfaces with the NIC hardware. The test constructs a sequence of Ethernet packets on one node and sends them at maximum rate to another node. On the other node, we verify that the received data is correct and then send an acknowledgement to the sender to indicate test completion. With this test, a single NIC is able to drive 100 Gbit/s of traffic onto the network, confirming that our current Linux networking software stack is a bottleneck.

Saturating Network Bandwidth

Because our current target server-blade designs are not capable of saturating the 200 Gbit/s network links that we are modeling even using bare-metal programs, we demonstrate the ability to saturate our network simulation by running concurrent streams across a cluster. We simulate a 16-node cluster with two ToR switches and one root switch. Each server attached to the first ToR switch sends data to the corresponding server on the other ToR switch (through the root switch). We then measure the aggregate bandwidth over time at the root switch. Figure 2.6 shows the results of this benchmark. We performed this test with different rate limits set on the sending nodes to simulate the standard Ethernet bandwidths of 1, 10, 40, and 100 Gbit/s. Each sender node starts a fixed unit of time after the previous sender began, so that traffic at the ToR switch ramps up over time and eventually saturates in the cases of 40 and 100 Gbit/s senders.

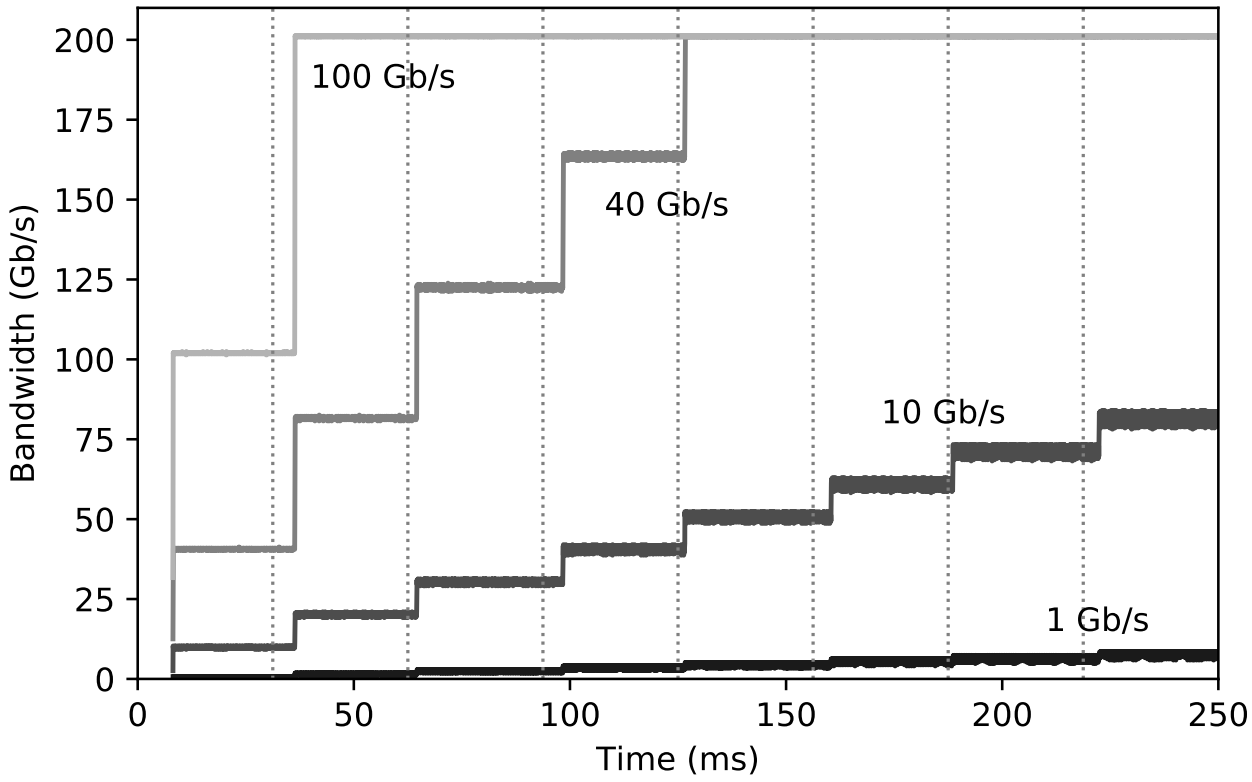


Figure 2.6: Multi-node bandwidth test. Dotted grey lines mark the entry points of individual senders.

In the 100 Gbit/s test, each sender is able to saturate the full bandwidth of its link, so the total bandwidth at the switch jumps up by 100 Gbit/s for each sender until it saturates at 200 Gbit/s after the second sender enters. In the 40 Gbit/s test, the total bandwidth increases by 40 Gbit/s for each additional sender until it saturates after five senders enter. In the 1 and 10 Gbit/s tests, the root switch's bandwidth is not saturated and instead maxes out at 8 and 80 Gbit/s, respectively.

Reproducing `memcached` QoS Phenomena from Deployed Commodity Clusters in FireSim

As an end-to-end validation of FireSim running a realistic datacenter workload, we run the memcached key-value store and use the mutilate distributed memcached load-generator from Leverich and Kozyrakis [139] to benchmark our simulated system. While much simulation work has focused on reproducing well-known phenomena like the long-latency tail, we go further to validate a finer-grained phenomenon: thread imbalance in memcached servers when memcached is instructed to use more threads than the number of cores in the system. Reproducing this result involves interaction between the core microarchitecture, operating system, and network. Under thread imbalance, a sharp increase in tail latency was shown while median latency was relatively unchanged [139].

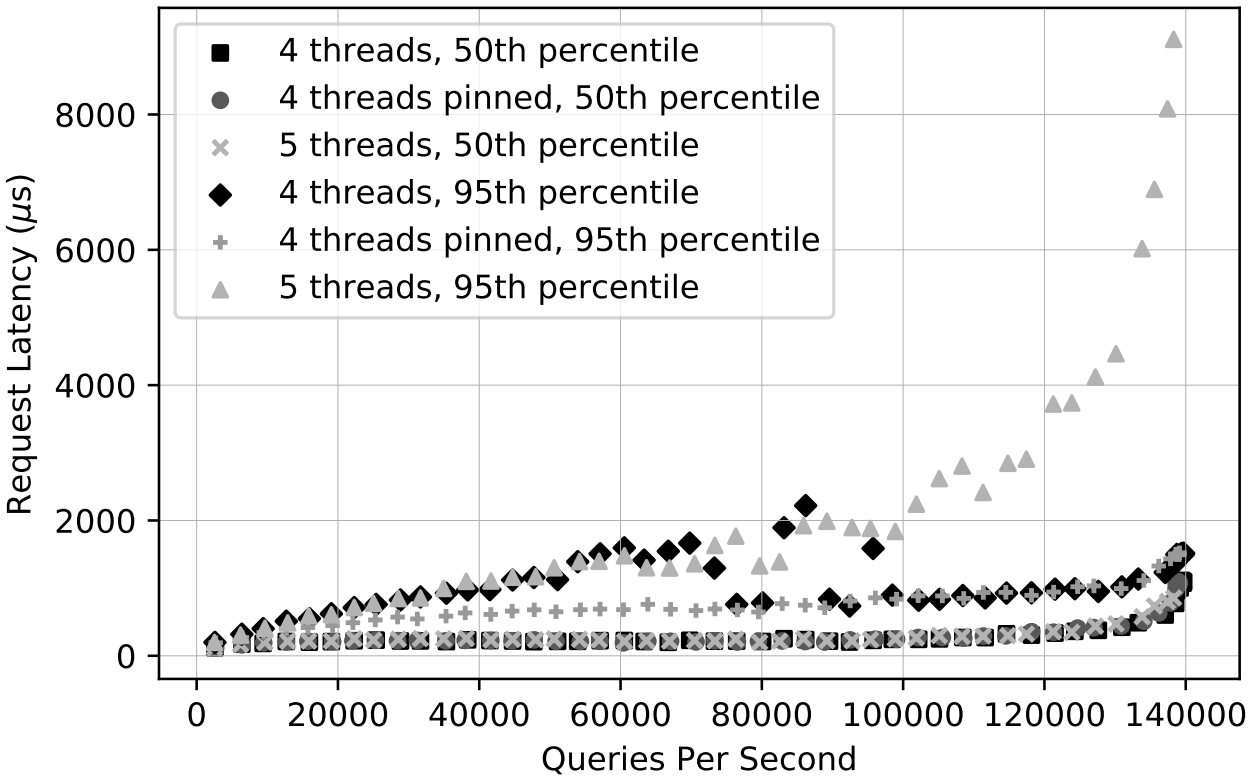


Figure 2.7: Reproducing the effect of thread imbalance in memcached on tail latency.

To replicate this result, we simulate an 8-node cluster in FireSim interconnected by a 200 Gbit/s, 2 μs latency network, where each simulated server has 4 cores. We provision one server in the simulation as a memcached host. We also cross-compile mutilate and its dependencies to run on RISC-V servers and run it on the remaining seven simulated blades to generate a specified aggregate load on the memcached server. On the serving node, we configure memcached to run with either 4 or 5 threads and report median and tail (95th-percentile) latencies based on achieved queries per second. Figure 2.7 shows the results of this experiment. As expected from the earlier work [139], we observe thread imbalance when running with 5 threads—the tail latency is significantly worsened by the presence of the extra thread, while median latency is essentially unaffected. In the 4-thread case, we also notice an interesting phenomenon at low to medium load—the 95th-percentile line for 4-threads tracks the 5-thread case, until a certain load is reached. We suspect that this phenomenon is due to poor thread placement in that region of QPS, even when the number of threads equals the number of cores in the system [139]. To confirm our suspicion, we run an additional experiment where we drive the same load, but run a memcached server with 4 threads and pin one thread to each of the four cores in the processor ("4 threads pinned" in Figure 2.7). In this case, we again see the same curve for 50th-percentile latency. However, the 95th-percentile curve is smoothed-out relative to the no-pinning 4-thread 95th-percentile curve, but overlaps it at high load, where we suspect that the Linux scheduler automatically places threads as if they were

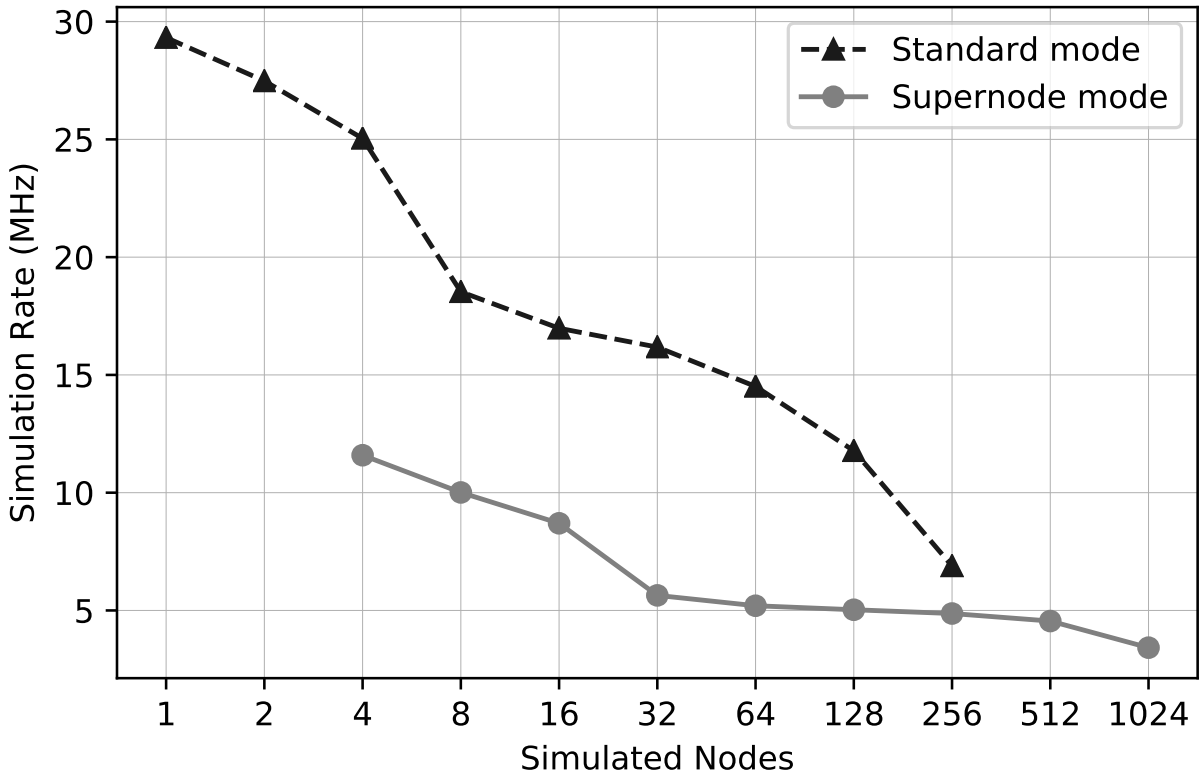


Figure 2.8: Simulation rate vs. # of simulated target nodes.

pinned one-to-a-core, even in the no-pinning case.

2.5 Characterizing FireSim simulation rates across system scale and configuration

In this section, we analyze the performance of FireSim simulations, using a variety of target configurations.

Performance vs. target scale

To show the overhead of token-based synchronization of all simulated nodes in clusters interconnected by a simulated 2 μs, 200 Gbit/s network as a function of cluster size, we run a benchmark that boots Linux to userspace, then immediately powers down the nodes in the cluster and reports simulation rate. This process does not exercise the network from the target perspective. However, as we do not yet perform any form of token compression, the number of tokens exchanged on the host platform is exactly the same as if there were network traffic (empty tokens are being moved between the target network endpoints). The only component of simulation overhead not included

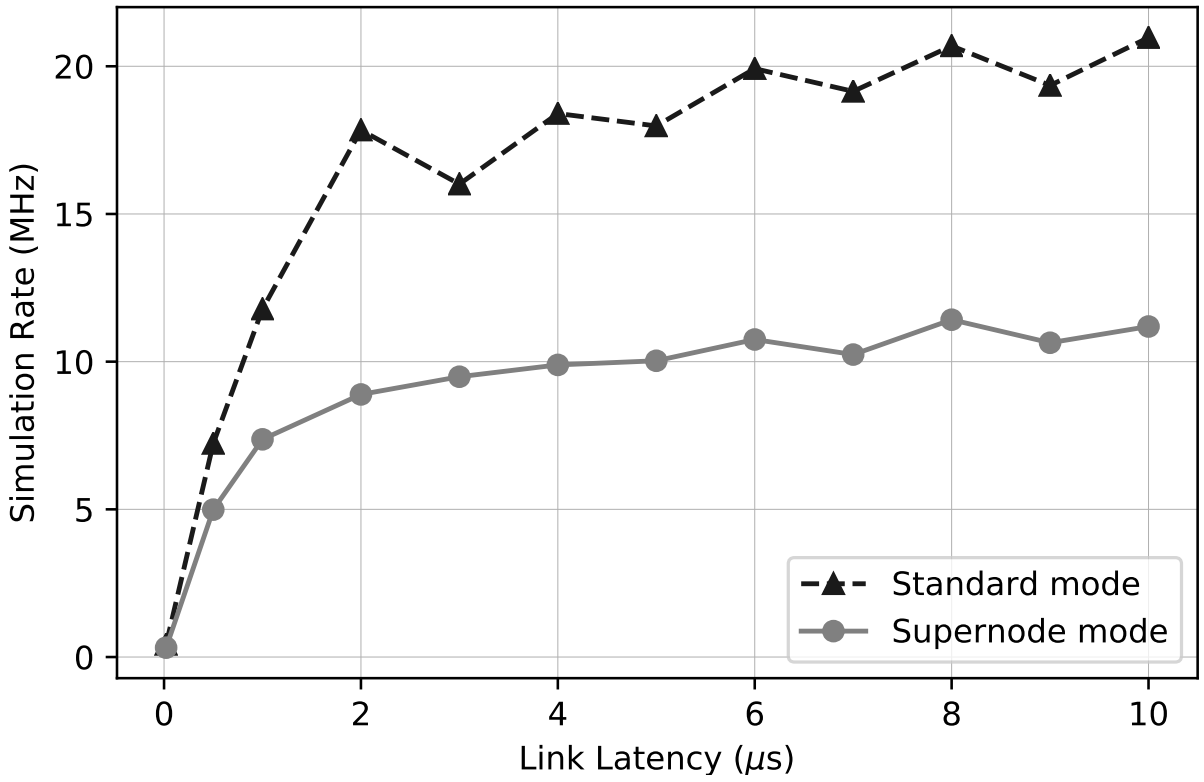


Figure 2.9: Simulation rate vs. simulated network link latency.

in this measurement is the overhead of packet switching inside the switch when there is traffic on the network. However, this is primarily a function of the load on a single switch, rather than simulation scale. This benchmark shows the overhead of distributing simulations, first between FPGAs on one instance, then between FPGAs in different instances. Figure 2.8 shows the results of this benchmark, both for "standard" and "supernode" FPGA configurations.

Performance vs. target network latency

As prior work has shown, moving tokens between distributed simulations is a significant bottleneck to scaling simulations [154]. Furthermore, as target link latency is decreased, simulation performance also decreases proportionally due to the loss of benefits of request batching. Throughout this work, we focus on a 2 μs link latency when benchmarking, since we consider this to be similar to latencies desired in experiments. Figure 2.9 shows simulation performance as a function of varying target link latency. As expected, simulation performance improves as the batch size of tokens increases.

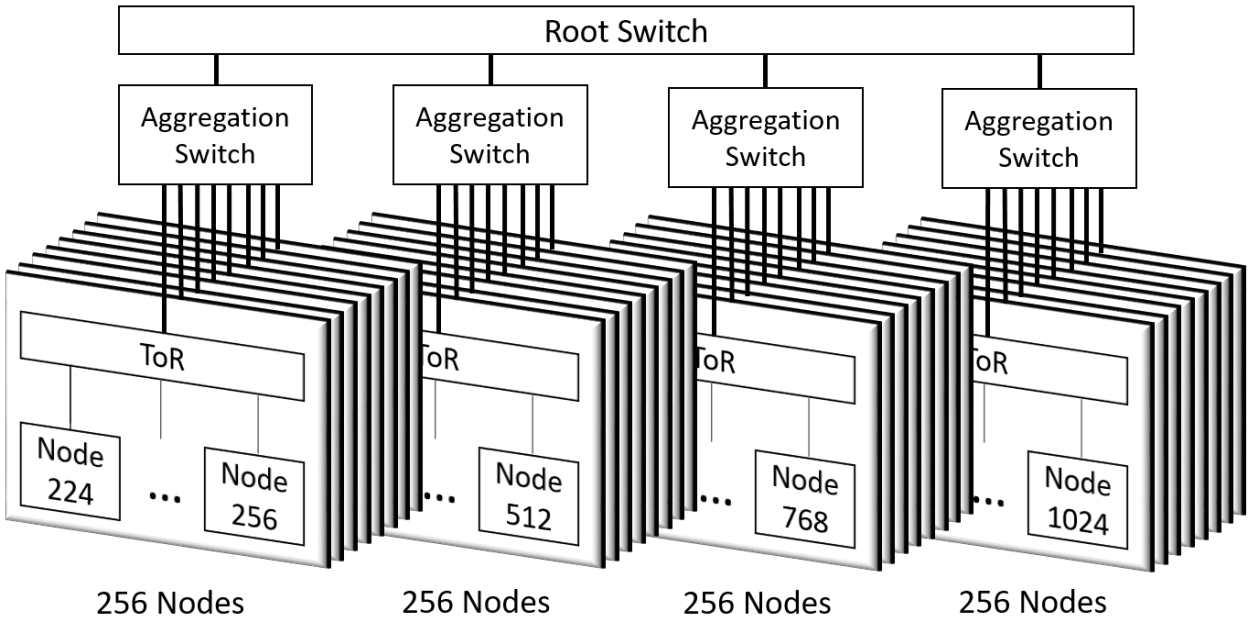


Figure 2.10: Topology of 1024-node datacenter simulation.

Table 2.3: 1024-node `memcached` experiment latencies and QPS.

	50th Percentile (μs)	95th Percentile (μs)	Aggregate Queries-Per-Second
Cross-ToR	79.26	128.15	4,691,888.0
Cross-aggregation	87.10	111.25	4,492,745.6
Cross-datacenter	93.82	119.50	4,077,369.6

Thousand-Node Datacenter Simulation

To demonstrate the scale achievable with FireSim, we run a simulation that models 1024 $\times$ 3.2 GHz quad-core nodes, with 32 top-of-rack switches, 4 aggregation switches, and one root switch, all interconnected by a 2 μs, 200 Gbit/s network and arranged in a tree-topology, at a simulation rate of 3.42 MHz. This design represents a more realistic target design point than the simpler design used as an example in Chapter 2.3, since we make use of FireSim's "supernode" feature to pack four simulated nodes per FPGA, giving a total of 32 simulated nodes attached to each ToR switch. Figure 2.10 shows this topology in detail. Each ToR switch has 32 downlinks to nodes and one uplink to an aggregation switch. Each aggregation switch has eight downlinks, each to one ToR switch and one uplink to the root switch. Finally, the root switch has 4 downlinks to the 4 aggregation switches in the target topology. This topology is specified to the FireSim simulation manager with around 10 lines of configuration code. More complicated topologies, such as a fat-

tree, can similarly be described in the manager configuration, but we do not explore them in this work. Compared to existing software simulators, this instantiation of FireSim simulates an order of magnitude more nodes, with several orders of magnitude improved performance.

To map this simulation to EC2, we run 32 `f1.16xlarge` instances, which host ToR switch models and simulated server blades, and 5 `m4.16xlarge` instances to serve as aggregation and root-switch model hosts. The cost of this simulation can be calculated for 2 different EC2 pricing models: spot instances (bidding on unused capacity) and on-demand (guaranteed instances). To calculate the spot price of this simulation, we use the longest stable prices in recent history, ignoring downward and upward spikes. This results in a total cost of ≈$100 per simulation hour. Using on-demand instances, which have fixed instance prices, this simulation costs ≈$440 per simulation hour. Using publicly listed retail prices of the FPGAs on EC2 (≈$50K each), this simulation harnesses ≈$12.8M worth of FPGAs. We expect that users will use cluster-scale experiments to prototype systems, with datacenter-scale experiments to analyze behavior at-scale once a system is already stable at cluster-scale.

As an example experiment that runs at this scale, we use the mutilate memcached load generator (as in Chapter 2.4) to generate load on memcached servers throughout the datacenter. In each experiment, there are 512 memcached servers and 512 mutilate load generator nodes, with each load generator targeting one server. We run these in 3 different configurations: one where all requests remain intra-rack, one where all requests cross an aggregation switch (but not the root switch), and one where all requests cross the root switch, by changing which servers and load generators are paired together. Table 2.3 shows average 50th-percentile latency, average 95th-percentile latency, and total QPS handled across all server-client pairs for these experiments. As expected, when we jump from crossing ToR switches only to crossing aggregation switches for each request, the 50th-percentile latency increases by 4 times the link latency, plus switching latency, or approximately 8 μs. A similar increase is seen in the 50th-percentile latency when moving from crossing aggregation switches to crossing root switches. On the other hand, in both cases, there is no predictable change in 95th-percentile latency, since it is usually dominated by other variability that masks changes in number of hops for a request. Finally, we see that number of queries per second decreases. This decrease is not as sharp as one would expect, because we limit the generated load to ≈10,000 requests per server, since we are interested primarily in showing the effects of latency rather than OS-effects or congestion.

2.6 A preliminary case-study of scale-out acceleration: Building a page-fault accelerator

In this section, as a case study to exemplify the cross-cutting architecture projects enabled by FireSim, we show preliminary performance results for a "Page-Fault Accelerator" [167], developed by Nathan Pemberton, Emmanuel Amaro, Howard Mao, and Deborah Soung, that removes software from the critical path of paging-based remote memory.

There have been several recent proposals to disaggregate memory in warehouse-scale comput-

ers, motivated by the increasing performance of networks, and a proliferation of novel memory technologies (e.g., HBM, NVM) [20, 90]. In a system with memory disaggregation, each compute node contains a modest amount of fast memory (e.g., high-bandwidth DRAM integrated on-package), while large capacity memory or NVM is made available across the network through dedicated memory nodes.

One common proposal to harness the fast local memory is to use it as a large cache for the remote bulk memory. This cache could be implemented purely in hardware (e.g., [213, 132]), which could minimize latency but may involve complicated architectural changes and would lack OS insights into memory usage. An alternative is to manage the cache purely in software with traditional paging mechanisms (e.g., [76, 84]). This approach requires no additional hardware, can use sophisticated algorithms, and has insight into memory usage patterns. However, our experiments show that even when paging to local memory, applications can be slowed significantly due to the overhead of handling page faults, which can take several microseconds and pollute the caches. In this case study, we propose a hybrid HW/SW cache using a new hardware device called the "page fault accelerator" (PFA).

The PFA works by handling the latency-critical page faults (cache-miss) in hardware, while allowing the OS to manage latency-insensitive (but algorithmically complex) evictions asynchronously. We achieve this decoupling with a queue of free page frames (*freeQ*) to be used by the PFA for fetched pages, and a queue of new page descriptors (*newQ*) that the OS can use to manage new page metadata. Execution then proceeds as follows:

- The OS allocates several page frames and pushes their addresses onto the *freeQ*. The OS experiences memory pressure and selects pages to evict to remote memory. It marks them as "remote" in the page tables and then provides them to the PFA for eviction.
- The application attempts to access a remote page, triggering the PFA to request the page from remote memory and place it in the next available free frame. The application is then resumed.
- Some time later (either through a background daemon, or through an interrupt due to full queues) the OS pops all new page descriptors off the *newQ* and records the (now local) pages in its metadata. The OS typically provides more free frames at this time.

We implemented the PFA in Rocket Chip and modified Linux to use it for all paging activity. For our baseline, we modified Linux to use the memory blade directly through its normal paging mechanisms (similar to Infiniswap [84]). The memory blade itself is implemented as another Rocket core running a bare-metal memory server accessed through a custom network protocol. Figure 2.11 plots PFA performance on two benchmarks: Genome, a de-novo genome assembly benchmark that involves random accesses into a large hash table, and Qsort, a simple quicksort benchmark. Both benchmarks were tuned to have a peak memory usage of 64 MiB. Quicksort is known to have good cache behavior and does not experience significant slowdowns when swapping. Genome assembly, however, has unpredictable access patterns, leading to significant cache thrashing in low local memory configurations. In both cases, the PFA significantly reduces the

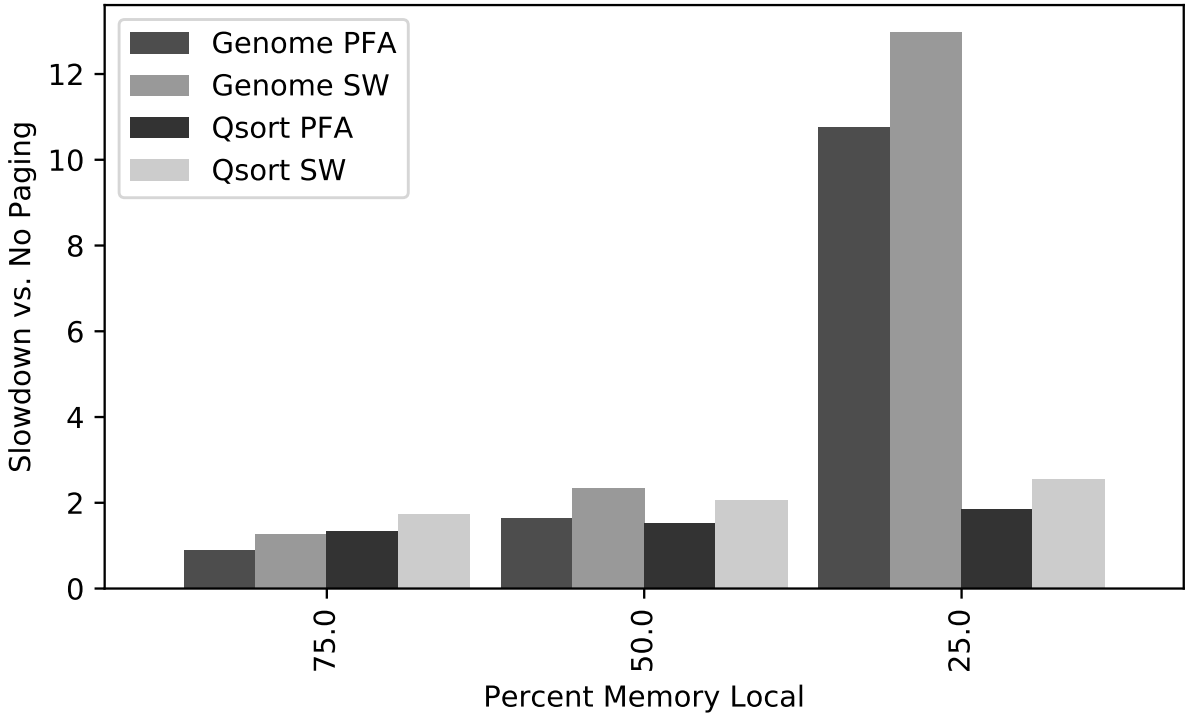


Figure 2.11: Hardware-accelerated vs. software paging.

overhead (by up to 1.4×). A more detailed analysis shows that while the number of evicted pages is the same in both cases, using the PFA leads to a 2.5× reduction in metadata management time on average. While the same code path is executed for each new page, the PFA batches these events, leading to improved cache locality for the OS, and fewer cache-polluting page-faults for the application. Future implementations could use a background daemon for new-page processing, further decoupling the application from new page management.

2.7 Related work

Computer architects have long used cycle-accurate simulation techniques to model machines and workloads, including servers in warehouse-scale machines. Simulation of machines at the scale of individual nodes was largely sufficient for 2010-era datacenters; the commodity Ethernet network used in most datacenters provided a natural decoupling point to draw a boundary on the effects of microarchitectural changes. Since most systems were built using commodity components, issues of scale could be measured in actual deployments and used to improve future designs. For example, in cases where scale-out workloads differed from traditional workloads in terms of fine-grained performance, performance-counter based mechanisms could be used to analyze microarchitectural

performance issues on commodity servers at scale [67, 110].

Other tools for simulating rack-scale systems can broadly be divided into three categories—software-based simulators that provide flexibility but low performance, hardware-accelerated simulators that are expensive to deploy and difficult to use but provide high-performance, and statistical models that analyze the big-picture of datacenter performance but are not targeted towards modeling fine-grained system interactions. Below, we review this prior work.

Software Simulators

One approach to simulating warehouse-scale computers is to take existing cycle-accurate full-system software simulators and scale them up to support multiple nodes. One simulator that uses this approach is dist-gem5 [154], a distributed version of the popular architectural simulator gem5. This simulator networks together instances of gem5 running on different nodes by moving synchronization messages and target packets. The primary advantage of these software-based approaches is that they are extremely flexible. However, this flexibility comes at the expense of performance—these software-based solutions are several orders of magnitude slower than FPGA-accelerated simulation platforms like FireSim. Also, software models of processors are notoriously difficult to validate and calibrate against a real design [86], and do not directly provide reliable power and area numbers. By utilizing FPGAs in a cloud service, FireSim matches many of the traditional flexibility advantages of software simulators, excluding short build times. Software simulators have also traditionally had more sophisticated hardware models than FPGA-based simulators, however the recent explosion in open-source hardware is providing realistic designs [21, 85, 26] that have the advantage of being truly cycle-exact and synthesizable to obtain realistic physical measurements.

Another prior software-based approach is reflected in the Wisconsin Wind Tunnel (WWT) [179] project, which used the technique of direct execution to achieve high simulation performance for individual nodes. In turn, WWT encountered similar performance bottlenecks as FireSim—network simulation has a significant impact on overall simulation performance. Follow-on work in the WWT project [41] explored the impact of several different network simulation models on the performance and accuracy of WWT. Similarly, we plan to explore more optimized network simulation models in FireSim in the future, which would trade-off accuracy vs. performance to support different user needs, using our current cycle-exact network model as the baseline. FireSim already supports the other extreme of the performance-accuracy curve—purely functional network simulation—which allows individual simulated nodes to run at 150+ MHz, while still supporting the transport of Ethernet frames between simulated nodes.

The Graphite simulator [153] takes a different software-based approach to simulating datacenters. Graphite can simulate thousands of cores in a shared-memory system at high simulation rate (as low as 41× slowdown), but only by dropping cycle accuracy and relaxing synchronization between simulated cores. Moreover, unlike full-system software simulators, Graphite only supports user applications and does not boot an OS.

A final software-based approach to simulating datacenters is to abstractly model the datacenter as a set of statistical events. This reduces simulation runtime, but sacrifices the fidelity provided by detailed microarchitectural models. This approach is used by datacenter-scale simulators like

BigHouse [150] and MDCSim [141]. BigHouse models a datacenter using a stochastic queuing simulation, representing datacenter workloads as a set of tasks with a distribution of arrival and service times. The distribution is determined empirically by instrumenting deployed datacenter systems. These distributions are then used to generate a synthetic event trace that is fed into a discrete-event simulation. The simulation is distributed across multiple nodes by running a separate instance of the simulator with a different random seed on each node, then collecting the individual results into a final merged result. MDCSim separates the simulation into three layers: user, kernel, and communication. It then models specific services and systems in each layer, such as web and database servers, schedulers, and NIC, as M/M/1 queues. While these simulation tools are useful for providing the "big picture" of datacenter performance and can do so considerably faster than FireSim, FireSim instead focuses on modeling fine-grained interactions at the level of detailed microarchitecture changes between systems at-scale.

Hardware-accelerated Simulators

Several proprietary tools exist for hardware-accelerated system simulation, such as Cadence Palladium, Mentor Veloce, and Synopsys Zebu [108]. These systems are generally prohibitively expensive (≈millions of dollars) and thus unattainable for all but the largest industrial design groups.

Some existing projects use FPGAs to accelerate simulation of computer systems, including large multicore systems [16]. Most recently, projects in the RAMP collaboration [215] pushed towards fast, productive FPGA-based evaluation for multi-core systems [49, 165, 50, 204]. However, most of these simulation platforms do not support simulation of scale-out systems. To our knowledge, the closest simulator to FireSim in this regard is DIABLO [203, 201] and FireSim builds on key lessons learned from the RAMP and DIABLO projects. Although DIABLO also uses FPGAs to simulate large scale-out systems, there are several significant differences between DIABLO and FireSim:

Automatically transformed RTL vs. Abstract Models. In DIABLO, servers are modeled using handwritten SystemVerilog abstract RTL models of SPARC cores. Authoring abstract RTL models is far more difficult than developing an actual design in RTL, and abstract RTL cannot be run through an ASIC flow to gain realistic power and area numbers. FireSim's simulated servers are built by directly applying FAME-1 transforms to the original RTL for a server blade to yield a simulator that has the exact cycle-by-cycle bit-by-bit behavior of the user-written RTL. Simulated switches in DIABLO are also abstract RTL models. In FireSim, users still write abstract switch models, but since switches are not the simulation bottleneck, switch models can be written in C++ making them considerably easier to modify. For detailed switch simulations, FireSim could be extended to also support FAME-1 transformations of real RTL switch designs.

Specialized vs. Commodity Host Platform. Given the lack of FPGA-cloud platforms at the time, DIABLO uses a custom-built FPGA platform that cost ≈$100K at time of publication, excluding operation and maintenance costs and the requirement for sysops expertise. This choice of platform makes it very difficult for other researchers to use DIABLO and reproduce results. In contrast, FireSim is deployed in a public cloud environment where Amazon has made all platform-specific FPGA scripts open-source [13]. Similarly, the entire FireSim code base is open-source,

which allows any user to easily deploy simulations on EC2 without the high CapEx of a DIABLO-like system. Furthermore, Amazon frequently gives substantial EC2 credit grants to academics for research purposes through their AWS Cloud Credits for Research program [12].

2.8 Conclusion

The open-source FireSim simulation platform represents a new approach to warehouse-scale architectural research, simultaneously supporting an unprecedented combination of *fidelity* (cycle-exact microarchitectural models derived from synthesizable RTL), *target scale* (4,096 processor cores connected by network switches), *flexibility* (modifiable to include arbitrary RTL and/or abstract models), *reproducibility*, target *software support*, and *performance* (less than 1,000× slowdown over real time), while using a *public FPGA-cloud platform* to remove upfront costs and provide large cluster simulations on-demand.

2.9 A retrospective on six years of FireSim

Introduction

"Why is it called *FireSim*?" is a question we receive often, posed by users who today employ FireSim to simulate a variety of systems beyond its initial goal: as a "from-the-RTL-up" simulator for a specialized Warehouse-Scale Computer (WSC) architecture called *FireBox* [20]. When we set out to build FireSim, the published mandate [20] was to:

> *"...simulate an entire FireBox, including the fiber-optic network, the switch, the NIC, and 1000 SoCs, with every core running the full BDAS stack (from the AMP Lab) and the Linux OS, as well as interactive services and batch applications, with only a factor of 1000x slowdown from realtime."*

The FireSim ISCA'18 paper (Chapter 2 and [118]) describes how we exceeded these objectives, with one caveat: our demo applications were not JVM-based as no usable RISC-V JVM existed in 2017. While the achieved scale was exciting, the true promise and broader adoption of FireSim has been driven by *how* this scale was realized.

Hardware Trends in 2017

Around 2017, when building FireSim, we identified several technology trends that helped us overcome these issues:

FPGAs in the public cloud became broadly available [27, 112, 116]. Cloud computing, established for years in systems research [18], could now benefit architects. Academics/startups could elastically scale high-fidelity simulations to 1000s of nodes without buying millions of dollars of FPGAs. In large organizations, architects/systems SW developers, who rarely get access to costly "big metal" HW-accelerated simulators, could now co-design HW/SW directly using real RTL.

FPGA capacity grew sufficiently to host interesting research targets without *immediately* requiring "tricks" (multi-threading, abstract modeling, partitioning, etc.) from the FPGA simulator literature. Many were later added to FireSim, but critically, were not initially *required* to ship a useful simulator.

Open-source, industry-verified hardware implementations became available. These were sufficiently capable to serve as a base for architecture research and included microprocessors, caches, on-chip networks, and peripherals [22, 1, 2].

RISC-V brought broad SW support for open HW designs, allowing them to run entire operating systems and applications.

Intermediate representations of RTL enabled compiler passes that automatically transformed HW designs [103, 25].

FireSim's Design Philosophy

Several guiding principles enabled FireSim to scale from high-fidelity simulations of single SoCs to entire WSCs:

Treat FPGAs as a "simulation appliance." Users should focus on their design task, not FPGA platform specifics. FireSim took a cloud-first approach, with heavy automation to hide the complexities of using FPGAs and ample documentation (`https://docs.fires.im`). This automation was *required* to enable the simulation scale presented in the original paper, but all users today, including those simulating single systems or using on-premises FPGAs, reap the benefits.

Parameterized tapeout-friendly RTL should be the single source of truth. The vast majority of users should not have to modify and re-calibrate abstracted models.

FPGA simulation "tricks" should be applied by a compiler, e.g., FireSim's multi-ported RAM [147] and multi-threading optimizations [33]. Abstract models should be used only when necessary and must be heavily validated and re-used (e.g., FireSim's FASED DRAM model [34]). Extending upon this basis, FireSim now supports automatic designer-guided partitioning of large, monolithic RTL designs across multiple FPGAs via FireAxe [218].

One flow should scale from high-level architectural modeling and software implementation to pre-silicon verification/validation for tape-out. This required coherently packaging everything from RTL designs and VLSI tools to compatible software [120, 168]. In 2019, as the flow grew, parts were moved to a more logical home in Chipyard [14].

Easy-to-use instrumentation and deterministic simulation are essential. FirePerf [115] and DESSERT [123] added high-fidelity and deterministic debugging and performance instrumentation to FireSim, enabling SW-simulator-like flexibility and introspection, but at significantly higher speeds.

Open-source and the FireSim community

FireSim was open-sourced (`https://github.com/firesim/firesim`) in May 2018 alongside the publication of the ISCA paper. With the wide variety of features added to FireSim over time (many of which are highlighted in the previous sub-section), it has matured into the de

facto open-source high-performance FPGA-accelerated simulation platform for designs at various scales. While we cannot cover all of FireSim's features here, hundreds of pages of documentation covering many more features are available at `https://docs.fires.im`. Today, FireSim enables domain experts across the computing stack (e.g., algorithms and ML model experts) to work directly with computer architects on fast simulations of *real hardware implementations* running their *actual software stacks*, allowing domain experts to directly influence hardware designs in hours rather than years, *before* a hardware design is ossified in a fabricated chip.

As of the time of writing, FireSim has been *used* (not only cited) in over 60 peer-reviewed publications from first authors at over 25 academic and industrial institutions. These publications have spanned many EE/CS-focused research domains, including computer architecture, systems, networking, security, scientific computing, algorithms, HPC, circuits, design automation, and more. This validates our vision that an easy-to-use, high-performance simulator of realistic RTL hardware implementations would provide a shared platform to enable HW/SW co-design and collaboration between researchers at varying levels of the computing stack. The complete list of FireSim user publications and user institutions is too long to list here; see the FireSim website: `https://fires.im/publications/#userpapers`.

Several users have deployed FireSim in surprising ways. For example, FireSim has been used as a standard host platform for DARPA and IARPA programs. In particular, FireSim was used as a host platform for DARPA's first ever bug bounty program, FETT, to make several novel security-augmented hardware designs available to hundreds of white-hat hackers for security evaluation over the internet [128]. FireSim has also been used in the development of commercially available chips and industrial users have also published comparisons of FireSim simulations of their designs against their taped-out silicon, providing an end-to-end validation of FireSim's performance modeling capabilities [133]. Other users have even deployed FireSim for its *original* purpose of modeling novel, *scale-out* systems [95, 107]. In addition to its original Chisel HDL support, FireSim now also works with SystemVerilog designs, such as those generated by HLS toolchains [91].

To help build community, many tutorials at ISCA, MICRO, ASPLOS, and HPCA have given hundreds of attendees hands-on experience running FireSim simulations on cloud FPGAs. The first FireSim *Workshop*, co-located with ASPLOS 2023, brought together the FireSim community with a day of talks from external FireSim users (`https://fires.im/workshop-2023`). Workshops and tutorials are discussed further in Chapter 3.5.

FireSim was designed from the ground-up to support reproducibility, which has been a challenge in architecture research. The initial FireSim release included a script that reproduced the experiments from the ISCA paper, including the largest scale-out simulations. For users, FireSim provides an automated and user-friendly interface for managing simulations, enabling easy scaling from a few simulations hosted by on-premises FPGAs to massive parallel simulations using hundreds of FPGAs on cloud platforms like AWS EC2 F1. FireSim removes the high capital expenses traditionally involved in FPGA-based simulation, democratizing access to realistic pre-silicon modeling of new hardware designs, including at scale. Accessing individual simulated systems in FireSim looks like accessing any virtual machine, presenting a familiar interface to SW developers and enabling collaborative HW/SW co-design from a single-source-of-truth: the RTL for a digital hardware design. Exemplifying this democratization and corresponding ease of

reproducibility, many FireSim user papers have since undergone artifact evaluation processes now part of conferences, including receiving multiple distinguished artifact awards at conferences like ISCA and MICRO [113, 159, 218].

Today, FireSim is actively developed by a global group of contributors. Notably, a recent FireSim release, coinciding with our tutorial at ISCA-50, supports several *on-premises* FPGA boards, including desktop/server-class boards requested by users (Xilinx U250, U280, and VCU-118). Also added is the RHS Research Nitefury II, an exciting low-cost, portable board that works with a laptop via Thunderbolt or M.2. FireSim's on-premises FPGA support was added with the aforementioned design principles in-mind, maintaining the automation and abstraction that have made cloud-hosted FireSim a powerful tool. Users can also easily transition between on-premises and cloud FPGAs, enabling a *hybrid-cloud* approach where early development occurs on a small cluster of on-premises FPGAs, with the ability to burst to cloud FPGAs during deadlines.

In addition to this direct impact, FireSim was also selected as an IEEE Micro Top Pick [117], selected for the ISCA@50 25-year Retrospective 1996-2020 [119], and nominated for CACM Research Highlights.

Looking forward, we are excited to see how FireSim and the broader open-hardware community evolve.

Chapter 3

Chipyard: Agile generation of RISC-V systems-on-chip

In this chapter, we discuss the second component of our agile hardware/software co-design flow for hyperscale systems research, Chipyard. Chipyard (`https://chipyard.readthedocs.io`) is an open-source RISC-V system-on-chip design, implementation, and evaluation framework with a focus on agility, generator-driven-design, and automation. Chipyard grew out of the difficulties we experienced in building complete, working systems from the vast new collections of open-source hardware IP being developed at Berkeley and other institutions around 2018. A later section in this chapter will detail this history and Chipyard's impact (Chapter 3.5). But first, we will focus on Chipyard's capabilities, especially from the lens of serving as the basis for the development of specialized systems-on-chip, such as Hyperscale SoC (Chapter 4).

3.1 Use cases and philosophy

While Chipyard has been used in a variety of domains (Chapter 3.5), it was designed to address one key question common in modern architecture research:

> *Having identified a new specialization opportunity and devised a paper design, how can one identify its true impact on power, performance, and area (PPA)* ***rapidly*** *and in the context of a* ***complete system****?*

While there are numerous interpretations of the above question, Chipyard takes the following opinionated approach:

1. A complete hardware system is an RTL implementation of an entire system-on-chip, containing cores, accelerators, a memory hierarchy, peripherals, and more.

2. Designs should be expressed in a generator-driven way, rather than building single instances. Designers should use HDLs (e.g., Chisel) that employ modern software engineering tech-

niques, including powerful parameterization and metaprogramming mechanisms to enable design-space exploration and early verification.

3. A complete system includes software; not only microbenchmarks, but substantial end-user workloads that require real operating systems, compilers, and more.

4. Feedback on a design (e.g., PPA) comes from rigorous tools, not only modeling and intuition. These include industry ECAD tools and validated open-source tools.

5. All of the above must be achievable rapidly by users in small, agile teams.

Chipyard allows exceptions to these methodological-defaults (e.g., SystemVerilog-based IP can be easily integrated), but these tenets form the basis of the design decisions in Chipyard and are what makes Chipyard so powerful.

Figure 3.1 shows an overview of the Chipyard flow, starting with optional user inputs at the top. Users can supply their own configuration of the generated system-on-chip design or rely on included default example designs for systems of various scales. Users can supply their own Chisel or SystemVerilog IP blocks and integrate them in various ways, whether as new cores or tiles, new accelerators, peripherals, etc. Users can also supply custom software stacks to run on their designs or customize the included FireMarshal flows for generating packaged workloads running on RISC-V Linux distributions. The middle of the diagram highlights the various hardware and software components built into Chipyard, which we will cover individually in the following sections. Finally, the bottom of the diagram highlights various output flows in Chipyard and how they contribute to obtaining high-quality power, performance, and area (PPA) results for a novel system-on-chip design.

3.2 Chipyard's curated library of hardware components

We will first explore the set of hardware components included in Chipyard and shown in the middle of Figure 3.1. Chipyard provides a *curated* library of components that users can productively combine into complex systems-on-chip. The vast majority of hardware IP included in Chipyard is also silicon-proven in academic testchips and in some cases has even shipped in commercial products.

Put together, these components will be able to generate various types of specialized systems-on-chip, with Figure 3.2 serving as an example.

Tiles and Cores

Tiles in Chipyard consist of a RISC-V core and its private (usually L1) caches. Chipyard supports several different cores written in a variety of HDLs, while providing a standardized interface for integrating new cores.

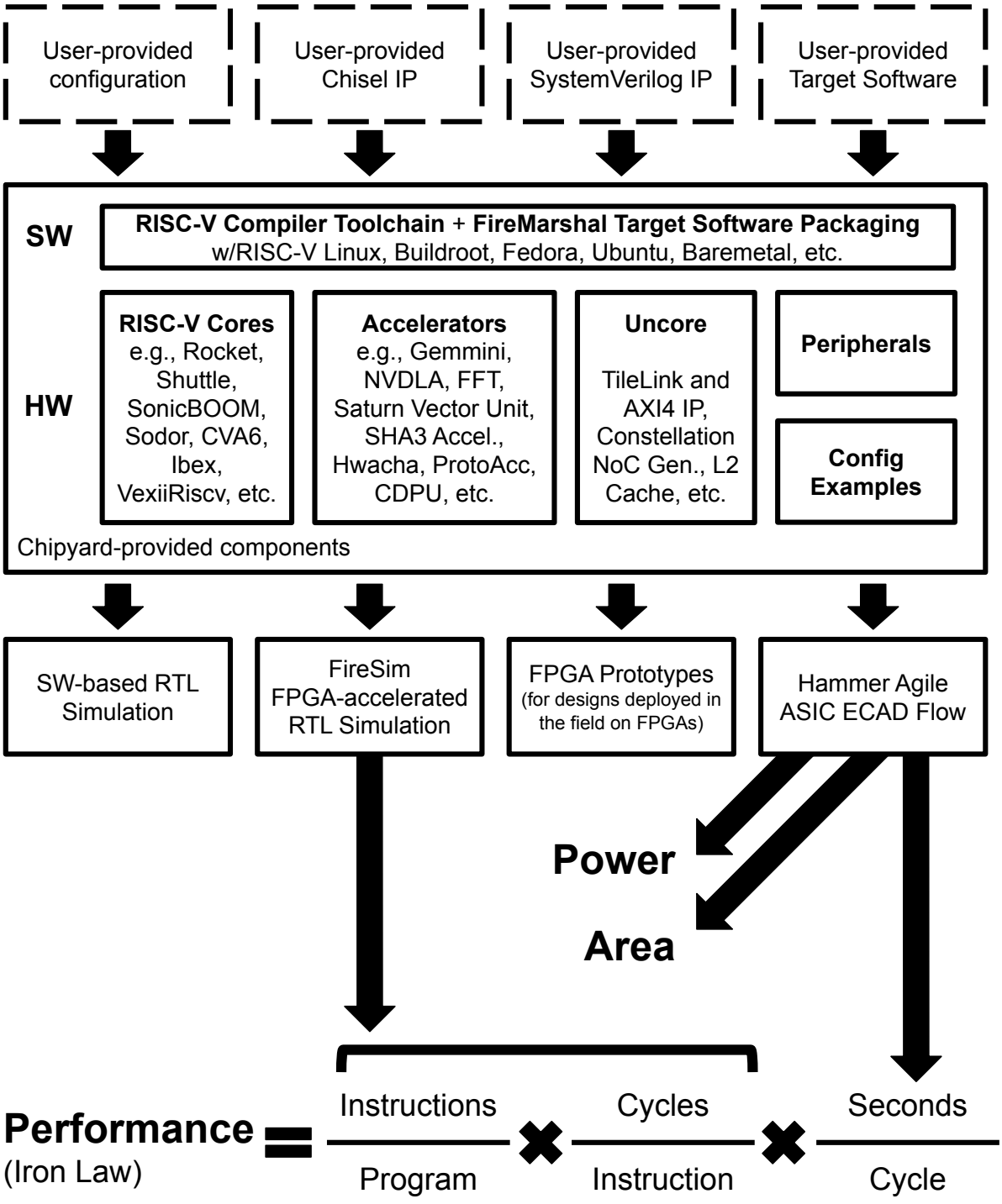


Figure 3.1: Overview of Chipyard inputs, flow, components, and outputs.

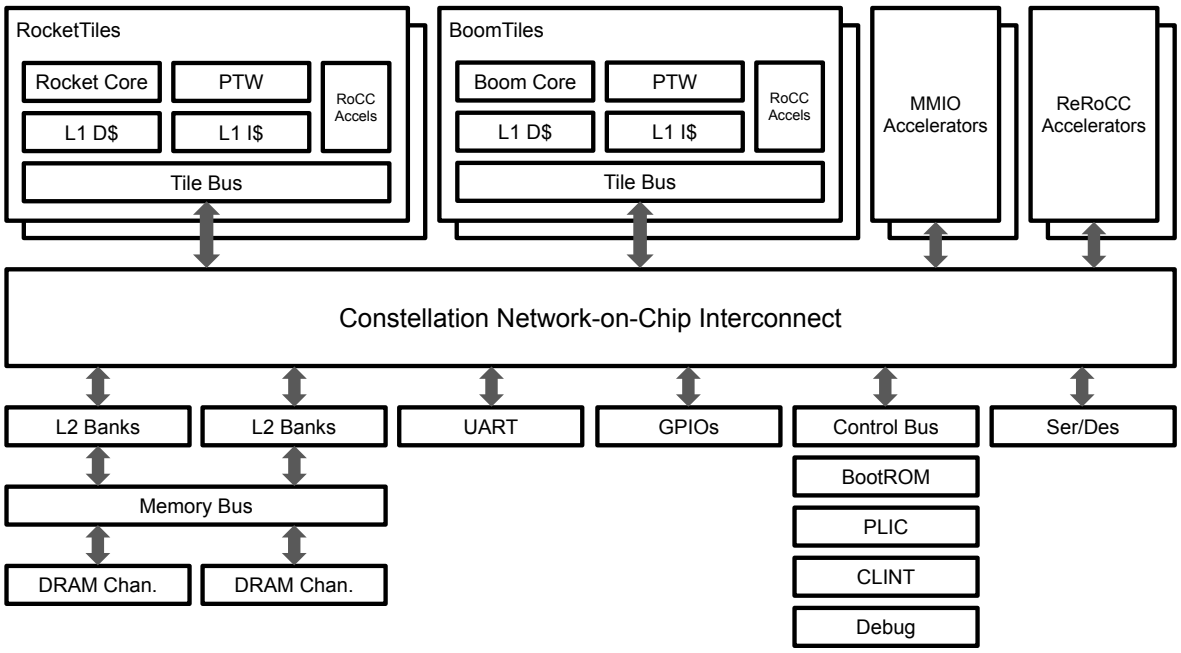


Figure 3.2: Example Chipyard-generated SoC design.

The most commonly used cores included in Chipyard are Rocket [22] and SonicBOOM [230, 228]. Rocket is a 5-stage single-issue in-order core designed as an efficient design point for low-power applications and was the first open-source RISC-V CPU. It supports the RV64GC profile, boots off-the-shelf RISC-V Linux distributions, and has been proven in numerous academic testchips and commercial products.

SonicBOOM [230, 228] is a 12-stage superscalar out-of-order RISC-V core that serves as a high-performance design point for general-purpose systems. Like Rocket, it supports the RV64GC profile and boots off-the-shelf RISC-V Linux distributions like Buildroot, Fedora, Ubuntu, and more. SonicBOOM has also been tapeout proven in several academic testchips. Both Rocket and SonicBOOM support attaching RoCC-based accelerators, which we will discuss further in Chapter 3.2.

Also included in Chipyard are several other cores:

- The **Sodor Educational Cores** [45] are a set of textbook-style RV32IM cores used in computer architecture classes at Berkeley, including 1, 2, 3, and 5-stage cores as well as a microcoded bus-based implementation.
- The **CVA6** [226] and **Ibex** [143] cores are SystemVerilog RISC-V implementations from the PULP project [182].

- **VexiiRiscv** [212] is a 6-stage dual-issue in-order core written in SpinalHDL.
- **SpikeTile** allows users to plug in the Spike golden reference RISC-V ISA simulator as a tile in Chipyard, replacing detailed core models with a fast functional ISA simulator to enable rapid testing of RTL-implementations of peripherals, accelerators, and more.

This large collection of integrated cores means that Chipyard can build SoCs designed for a wide-variety of applications and targeting a wide variety of power, performance, and area envelopes.

Accelerators

Chipyard supports two primary "sockets" for integrating accelerators: RoCC and MMIO.

The RoCC interface (originally, the *Rocket Custom Coprocessor* interface) enables integrating custom decoupled co-processors with Rocket and BOOM cores in Chipyard. Commands for accelerators are encoded in custom RISC-V instructions executed by the application processor. When the application processor encounters such an instruction, it dispatches a command to the accelerator that can contain optional register data and relevant metadata. The accelerator performs computations based on these commands and can write back into core registers. RoCC-attached accelerators also have access to the page-table walker and TLB to support virtual addressing. These accelerators can read/write data to/from both the L1 data cache or the outer memory (e.g., via the L2). A vast number of accelerators have been implemented within this abstraction in the literature. Chipyard includes several as composable examples, including the Gemmini ML accelerator [78], the Hwacha vector accelerator [136, 137, 135, 54], a SHA3 accelerator, and accelerators for (de)serialization and (de)compression that we will explore in later chapters. Recently, AuRORA [127] added support in Chipyard for ReRoCC, "Remote RoCC" accelerators that are disaggregated SoC-level accelerators rather than co-processors attached directly to cores. These accelerators can be shared with other cores while preserving much of the RoCC infrastructure.

Chipyard also supports more classical "MMIO-attached" accelerators that are connected directly to the system bus (i.e., on-chip network), controlled via memory-mapped registers, and have the ability to DMA to the memory system. Examples packaged in Chipyard include the NVIDIA NVDLA accelerator [161] and an FFT accelerator generator [208].

Uncore, SoC Interconnect, and Peripherals

Chipyard is built around the free and open TileLink [192] chip-scale coherent interconnect standard. TileLink is similar to AXI/ACE and supports common functionality required to build complex SoCs, including support for multi-core systems, accelerators, peripherals, direct-memory access, etc. Some TileLink interconnect IP comes from Rocket Chip, which provides RTL generators for crossbar-based buses, width-adapters, clock-crossings, and adapters to other protocols like AXI4 and APB. The Constellation Network-on-Chip (NoC) generator [229] is used in Chipyard to produce more complex TileLink interconnects, such as meshes. Constellation supports advanced

NoC features like multi-NoC support, irregular topologies, and deadlock-free wormhole routing with virtual channels.

Chipyard uses an open-source L2 from SiFive [189], which supports directory-based coherence and is highly parameterized to support various design tradeoffs. Backing memory in Chipyard is provided in one of three ways: RTL simulation uses DRAMSim2 [181], FireSim FPGA-accelerated simulations use the FASED DRAM Model [34], and taped-out chips either talk to memory controller IP or proxy requests to a testchip host over serial TileLink.

Chipyard also provides numerous standard peripherals, including interrupt controllers, JTAG, UART, GPIOs, SPI, I2C, PWM, clock-management devices, ser/des, scratchpads, and a 200 Gbit/s Ethernet NIC.

Configuration Layer

Chipyard configurations are written in Chisel/Scala and enable users to productively and rapidly express complex system-on-chip designs in a concise fashion. These configurations are broadly broken into three groups. The digital system configuration components allow users to manage IP within the SoC and IP parameters. The chip I/O configuration allows the user to dictate how the SoC will communicate with the outside world (e.g., I/O cells, FireSim Bridges, etc.). Finally, the harness configuration supplies information about what the I/Os will talk to, such as a software model, FireSim bridge, or tethered FPGA. In a few tens of lines, users can build complex Chipyard SoCs with dozens of cores, accelerators, a complex NoC, and more.

3.3 Managing software for the SoC design

Building complex software stacks is a strict necessity for effectively evaluating novel hardware-software co-design ideas. Building these software stacks and managing their evolution as a design evolves in maturity can be a daunting task in a small, agile team. As a start, Chipyard provides a RISC-V toolchain with associated compilers, libraries, simulators like Spike and QEMU [111], and debugging tools like GDB. The FireMarshal tool [168], co-developed with Chipyard and FireSim, bridges the gap between these toolchain components and reproducibly building and shipping complete workloads that run on Chipyard-generated designs (e.g., SPEC on Linux). Given a workload's YAML configuration, FireMarshal automates the process of building binaries and filesystems for the target design (whether baremetal or OS-hosted) as well as managing end-to-end experiments. This experiment management functionality includes incorporating input datasets, binaries, and other collateral as well as extraction of workload outputs. FireMarshal can also bridge various levels of simulation, for example pre-running complicated workload installation tasks in a QEMU-based simulation before actually running the workload on FireSim. Most critically, FireMarshal supports numerous RISC-V Linux distributions, including Ubuntu, Fedora, and Buildroot as well as supplying drivers for the various peripherals included in Chipyard.

3.4 Chipyard output flows

Elaboration flow

Chipyard's top-level elaboration flow is driven by Chisel [25] and FIRRTL [103]. Essentially, the "top" of Chipyard is a Chisel generator program that, when run, generates the FIRRTL intermediate representation for the design. As part of the execution of this program, various high-level pre-verification ("correct-by-construction") steps occur, including parameter negotiation and validation with Diplomacy [52].

Once the FIRRTL IR is generated for the design, various automated passes can run on the design to transform the target netlist. For example, FireSim simulators are automatically constructed using such passes and VLSI flows can also use FIRRTL passes to adjust the module hierarchy to satisfy downstream tools. Once the desired transformations are completed, CIRCT [142] is used to generate synthesizable Verilog and collateral for SystemVerilog blackboxes is also packaged in.

RTL-simulation flows

As a first step in the co-design loop, Chipyard supports many of the standard Verilog/SystemVerilog simulation tools, such as Verilator, VCS, and Xcelium. This enables running microbenchmarks and collecting waveforms. Various software shims are provided for peripherals, e.g., bridging UART to a user-readable file and bridging off-chip memory requests to DRAMSim2 [181].

FPGA-accelerated simulation flows

As a design develops and workloads become increasingly complicated, designs can automatically be deployed as FPGA-based *simulators* using FireSim, as described in Chapter 2.

Tape-out flows

Chipyard supports the Hammer [214] agile ASIC flow for both ASIC quality-of-results metric collection (power, frequency, area) and final tape-out. Hammer supports tools from several popular ECAD tool vendors as well as open-source flows. Hammer also has downloadable plugins for several commercial and open-source PDKs. Lastly, Chipyard also supports generating FPGA *prototypes* for the express purpose of building a host/tether system for testchips.

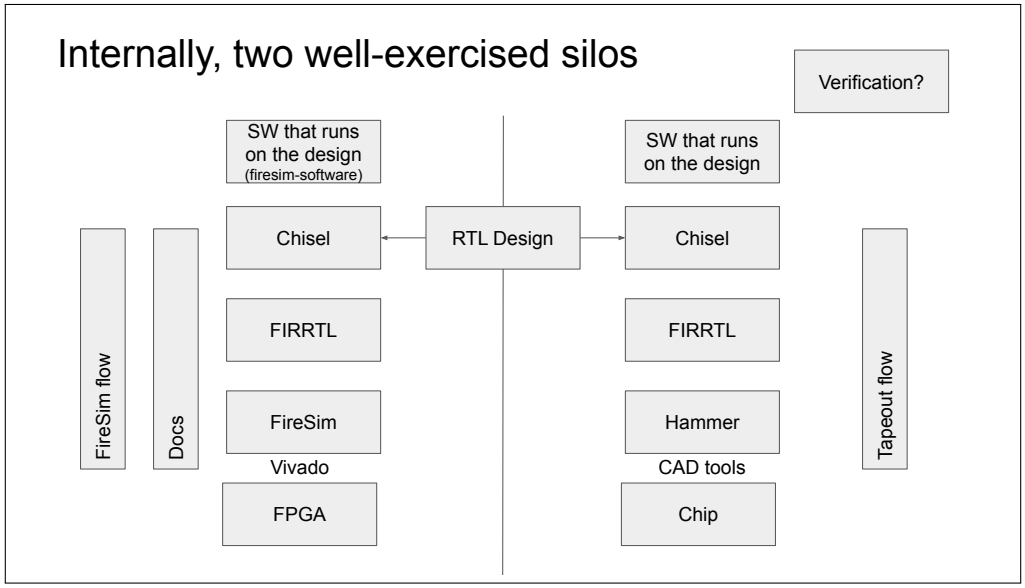


Figure 3.3: Internal hardware development flows at Berkeley in 2018. From "Building an (easy-to-use) ecosystem" (unpublished), a talk from this dissertation's author at a 2018 lab offsite.

3.5 A retrospective on five years of Chipyard

Today, Chipyard is *one of the leading open-source platforms* for specialized RISC-V SoC design and both the Chipyard and FireSim communities are growing rapidly. This chapter will outline some of the history behind Chipyard's development and some of the lessons learned along the way. One key reason for Chipyard's success is that it grew out of our own need for a productive and agile hardware development flow; Chipyard was used by its own developers from day one.

Battling productivity loss in the era of complex, heterogeneous systems-on-chip

Chipyard's development grew out of a session at a 2018 lab offsite at Berkeley where we discussed the state of productivity in our hardware development flows. At the time we had two key problems, first that internally we had multiple flows: the FireSim flow for architecture research (open-sourced in May 2018) and our tapeout flow (Figure 3.3). Researchers developing new hardware designs had to separately integrate their design into these two flows and manage/synchronize multiple "top" repositories. This was doable, especially with access to institutional knowledge, but was neither scalable nor productive.

Secondarily, it was extremely challenging for external users (or new internal users) to understand the interactions between components and put together a flow for their project that spanned from early design-space-exploration, pre-silicon validation and ASIC quality-of-results metric collection, and tape-out (Figure 3.4). While FireSim shipped with documentation from day-one about

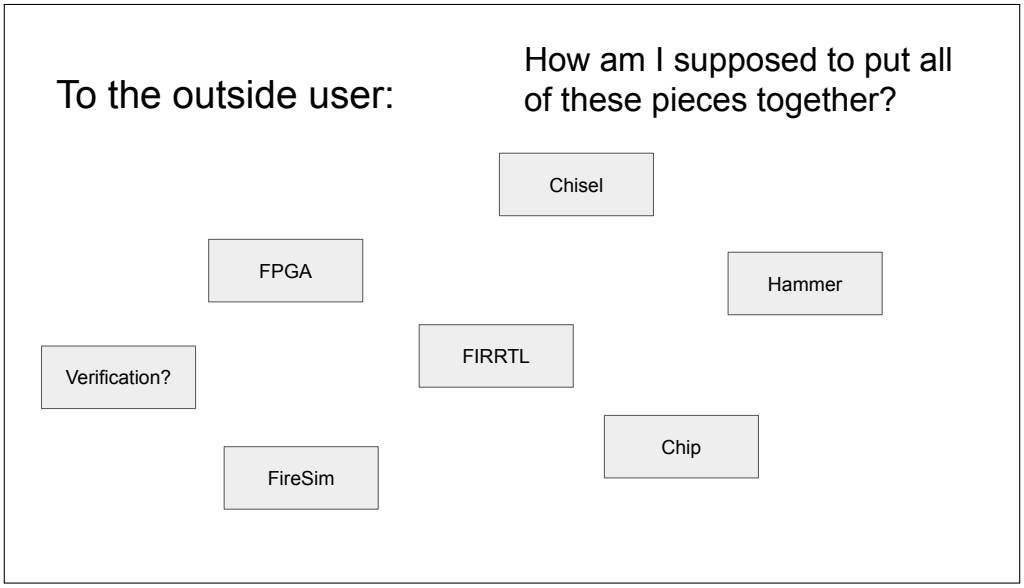


Figure 3.4: Externally visible hardware development flows from Berkeley in 2018. From "Building an (easy-to-use) ecosystem" (unpublished), a talk from this dissertation's author at a 2018 lab offsite.

how to use FireSim as a simulator, naturally this documentation did not dive in-depth into how to modify the design being simulated nor how to use any VLSI flow for ASIC metric collection.

The conclusion from this offsite session was that it would be worthwhile for us in the long run to spend time developing a new agile system-on-chip development framework with the explicit goal of enabling us (and the broader community) to build specialized systems more quickly. After a sequence of long meetings, deliberation, and technical discussion (and bike-shedding), this became Chipyard[1]. Technical details behind Chipyard have already been discussed in depth in Chapter 3. In the rest of this chapter, we will highlight some key aspects behind Chipyard's wide adoption.

[1]Originally called "REBAR", which was quickly ditched, but can still be found in old slides via a well-crafted Google search, along with a more comical depiction of the origin story described in this chapter.

Hands-on tutorials

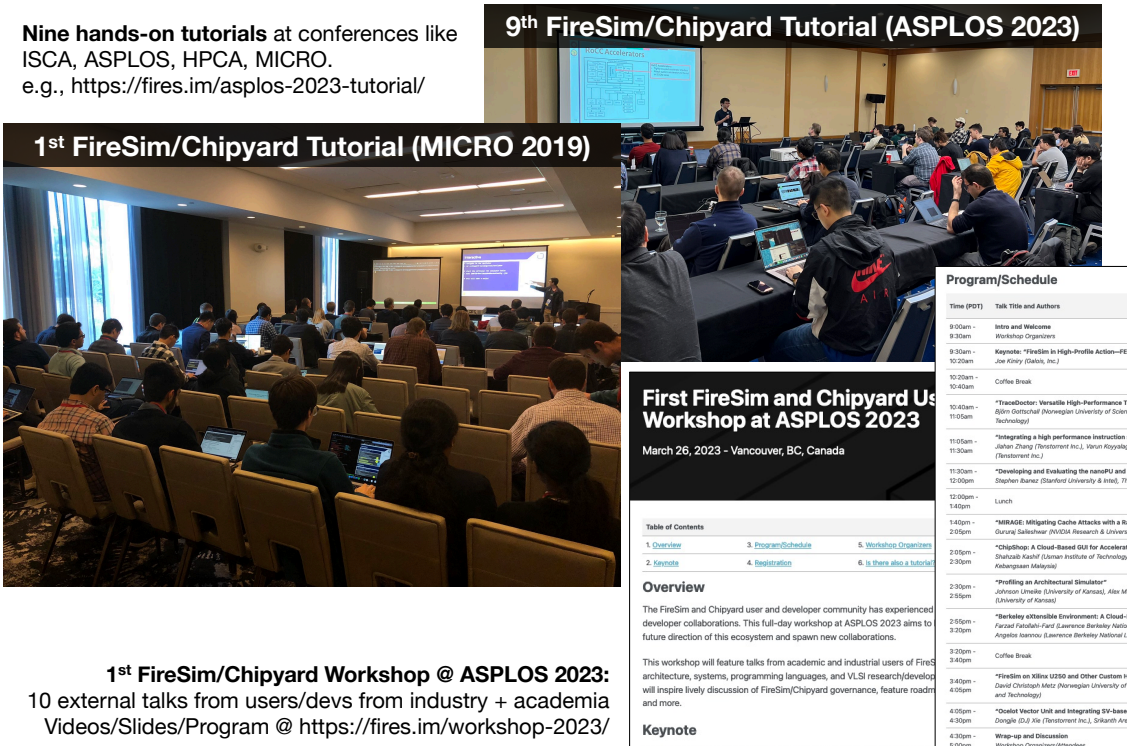


Figure 3.5: Chipyard/FireSim community building.

While both Chipyard and FireSim have large amounts of public documentation, we decided that the best way to reach users, refine our methodologies, and understand what truly works in an agile hardware design flow was to run *hands on* tutorials at various venues. This tutorial was designed to showcase a wide range of Chipyard and FireSim use-cases: attendees would get to customize a Chipyard-generated RISC-V system-on-chip with a hashing accelerator, run microbenchmarks on it in software RTL simulation, build a FireSim FPGA image, and then boot Linux on the SoC in FireSim (on cloud FPGAs) and run userspace applications that exercised the custom accelerator.

Having harnessed the benefits of a cloud-first approach when building FireSim, we decided that the flexibility of the cloud would serve as a natural platform for building scalable tutorials. Given the popularity of these tutorials, this was a prescient choice: purchasing only the FPGAs required to run our very first tutorial (with over 60 attendees) would have cost around 1.2 million dollars (not even counting host servers or OpEx). Instead, the all-in cloud compute cost of running our first tutorial was only $1254.44.

To enable this, we designed cloud infrastructure for managing isolated, controlled access to cloud instances for tutorial attendees that worked in cooperation with upstream FireSim's cloud manager, which dynamically launches/terminates instances as necessary to keep costs in control. Users sign up on a Google form, are automatically allocated an instance, and then are emailed an

IP address and keypair to `ssh` in. Another key advantage of this approach was that attendees could bring essentially any computer; once they were `ssh`-ed[2] into our cloud instances, they were in a sufficiently powerful and controlled environment to easily follow along with the tutorial.

Interestingly, this tutorial-oriented cloud management infrastructure also become extremely useful as artifact evaluation became popular in computer architecture conferences. It allowed us to easily give artifact evaluators access to specialized hardware used in our papers (e.g., FPGAs for FireSim) without putting the burden of obtaining that hardware on the evaluators.

To date, this tutorial has been run over nine times at numerous computer architecture conferences and other venues, with hundreds of researchers, students, engineers, and educators getting hands-on experience with our agile co-design tools. We have seen consistently high levels of interest (Figure 3.5) and plan to continue to run these tutorials in the future.

Community workshops

In addition to tutorials, we organized the First FireSim and Chipyard User and Developer Workshop at ASPLOS 2023 (`https://fires.im/workshop-2023/`). The goal of this workshop was to bring together the Chipyard/FireSim community to help drive the future direction of the ecosystem and spawn new collaborations. The first workshop was a full-day event with a total of ten talks (bottom-right, Figure 3.5) from external Chipyard/FireSim users and developers covering a wide variety of topics, including FireSim's usage in DARPA FETT, high-performance tracing, co-simulation, high-performance networking, new host platforms, security, and more. Like the tutorials, we plan to continue this workshop in the future.

Chipyard usage

Chipyard has become widely used in academic research, teaching, tapeouts, and more broadly in the open-source hardware community. For example, the Chipyard repository on GitHub has over 650 public forks. Over 45 of these public forks have seen active development in the last six months and many hold an entire *organization's* worth of Chipyard user projects. More than 700 people have joined Chipyard's mailing list to track development milestones and to ask and answer questions about Chipyard usage. Also, over 90 unique contributors have made commits to Chipyard since its inception. Chipyard has also been used extensively in courses at UC Berkeley [15] and other universities. Most notably, the "tapeout" class at Berkeley has harnessed Chipyard to enable students to go through the entire tapeout flow in a single semester, from nothing to a GDS for a bespoke, specialized SoC shipped to fabrication [205]. In the following semester, students take a "bringup" class, where they get the chip back from fabrication and in each semester have successfully demonstrated working chips.

[2]Perhaps surprisingly, this is not a perfect approach: once in a while we had issues with locale configuration on an attendee's laptop leaking into remote machines they `ssh` into; similarly, some attendees had local configurations that automatically applied various customization actions to any machine they `ssh`-ed into.

Lessons from running large academic hardware projects

Below is a summary of some of the key lessons we learned (or ideas that were validated) from the Chipyard/FireSim projects:

1. "Dogfooding" is key. Chipyard's developers are also Chipyard users (up to the time of writing, this is true without exception). As a result, many good design decisions are obvious because they solve problems we face on a day-to-day basis.

2. For projects oriented around improving productivity and scale-of-design, there is no better way to convince users than to show them hands-on and end-to-end. Hands-on tutorials are the single best tool for onboarding users. Of course, good documentation is essential too.

3. For academic projects in particular, continuous integration is critical, since it implicitly distributes the workload of maintaining the project, which grows as the complexity and amount of different IP in modern SoCs continues to increase. Maintaining an open-source project is rarely on anyone's critical path and it is commonly the case that there is only one person who actually has time to spend on releases, management, etc. Their job must be made as easy as possible and ensuring that the end-to-end system continues to work as developers make individual contributions is a key step towards this.

4. Spend disk space and CPU time in initial setup to save actual human-in-the-loop iteration time. There is always an urge to try to optimize these parameters in a large project, but it should rarely be done at the expense of latency when a user is actively working on their design. In practice, micro-optimizations like only initializing parts of the repo with option flags or only caching parts of initial builds end up slowing down users more when they suddenly decide to start using a feature. Perhaps more importantly, they create a complex matrix of repository states that make testing/CI extremely costly.

5. Tying project releases to tutorials is useful. Since everything *must* work for a live, hands-on tutorial, releasing at the same time forces a high level of release quality.

6. If you can, use the cloud. Supporting a single host platform eliminates the vast majority of user issues, especially for specialized hardware like FPGAs. Of course, this is not always feasible but can still be used for certain cases (e.g., the aforementioned tutorials).

Looking forward, we are excited to see how Chipyard, FireSim, and the broader open-hardware community evolve.

Chapter 4

Hyperscale SoC: A server system-on-chip optimized for cloud datacenters

In this chapter, we switch contexts from building agile hardware/software co-design *platforms* to designing new specialized hardware that directly addresses key system-level inefficiencies in hyperscale servers.

Prior work has highlighted critical overheads that drastically reduce the efficiency of hyperscale cloud servers, such as the "datacenter taxes" [110, 198]. These datacenter taxes include fundamental primitives that are needed to enable scalable and distributed computation, such as serialization, compression, remote-procedure call, memory-movement operations (copies, moves, etc.), hashing, and memory allocation. Together, these have been shown to account for over 25% of CPU cycles in hyperscale server fleets [110] with several additional externalities that we will show in later parts of this dissertation.

Addressing these overheads has previously been a challenge because they are deeply intertwined with the systems software stack. Accelerating these operations requires much finer-grained specialization as compared to other domains. This motivated the design of Hyperscale SoC (Chapter 4), a custom server system-on-chip optimized for the cloud.

As part of a multi-year industry-academic collaboration between Google and UC Berkeley, we architected, implemented, evaluated, and ultimately taped-out Hyperscale SoC. Hyperscale SoC is based on Chipyard, configured with high-performance RISC-V MegaBOOM OoO application cores, a wide on-chip network to match existing hyperscale designs, and the 200 Gbit/s Ethernet NIC from FireSim [118]. Rather than relying on the coarse-grained acceleration opportunities enabled by traditional server integration (e.g., PCIe cards), Hyperscale SoC can take advantage of several forms of near-CPU accelerator integration to enable fine-grained offloads.

To use this design as a hyperscale research platform, we first had to bring the SoC's networking performance in-line with modern cloud servers, leading to the FirePerf project. Once we had a reasonable baseline, we specialized the SoC design for costly but foundational operations in hyperscale servers such as (de)serialization, general-purpose lossless (de)compression, and memory copies/moves.

While we will return to these novel architectural ideas in later parts of this dissertation, a core

thread tying together the Hyperscale SoC project is *rigorous, data-driven methodology*. Given the goal of building a highly specialized HW/SW system for the Hyperscale Cloud, naturally we must answer the questions alluded to earlier:

1. How do we know that we are building the *right* specialized hardware?
2. How do we ensure we are building a *complete solution* that allows us to understand the *end-to-end impact* of our novel architectural ideas?

4.1 Profiling a hyperscale datacenter fleet to influence hardware designs and construct representative benchmarks

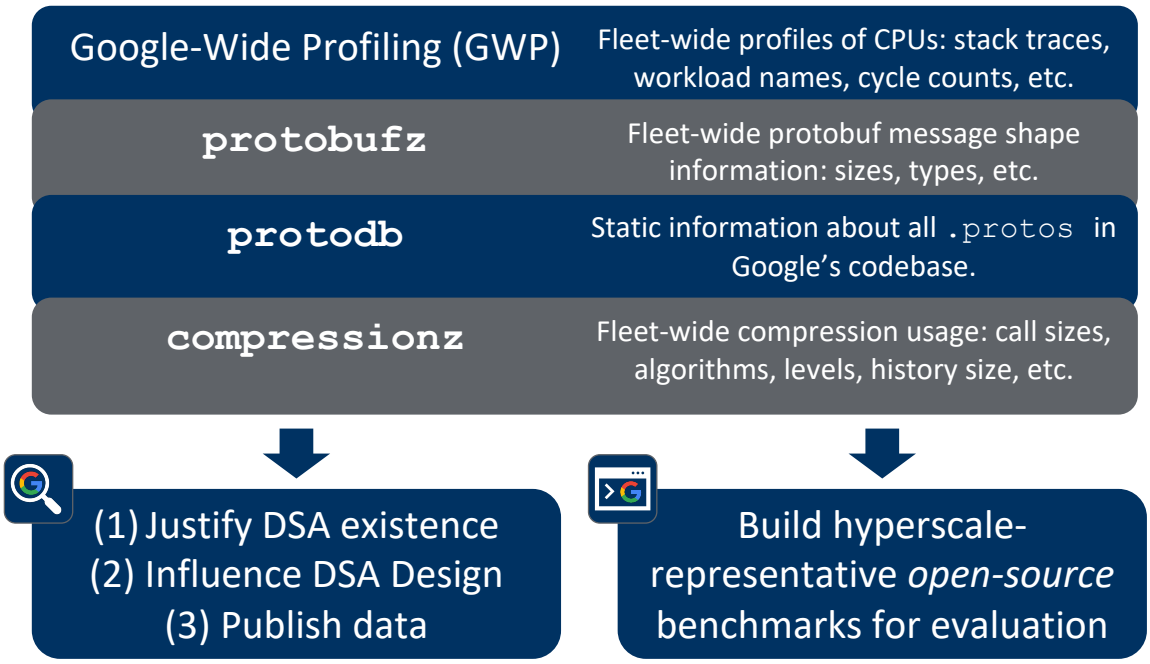


Figure 4.1: The *data-driven* co-design methodology for hyperscale systems established and used in this dissertation. Driven by profiling data collected at Google, we justify the existence of various domain-specific accelerators and derive requirements for these accelerators. We also open-source new Hyperscale-representative benchmarks for important domains, including Google HyperProtoBench and Google HyperCompressBench.

To answer the first question, we must understand what specialized hardware should be deployed in next-generation datacenters. Given the ability to profile Google's worldwide hyperscale dat-

acenter fleet[1], we employ the novel methodology shown in Figure 4.1. We use Google-Wide Profiling [180] to collect fleet-wide data about CPU cycle consumption across Google's datacenter fleet to understand which domains to focus on when designing specialized hardware. We also use (or develop and use) domain-specific, fleet-wide profilers at Google (`protobufz`, `protodb`, and `compressionz`) to find insights about how specific accelerators should be designed for the hyperscale environment. Using this data, we also build and open-source representative benchmarks for key domains on behalf of Google, HyperProtoBench [82] and HyperCompressBench [81], both of which have now been upstreamed into Google's Fleetbench suite [4].

4.2 An end-to-end data-driven co-design flow for hyperscale systems

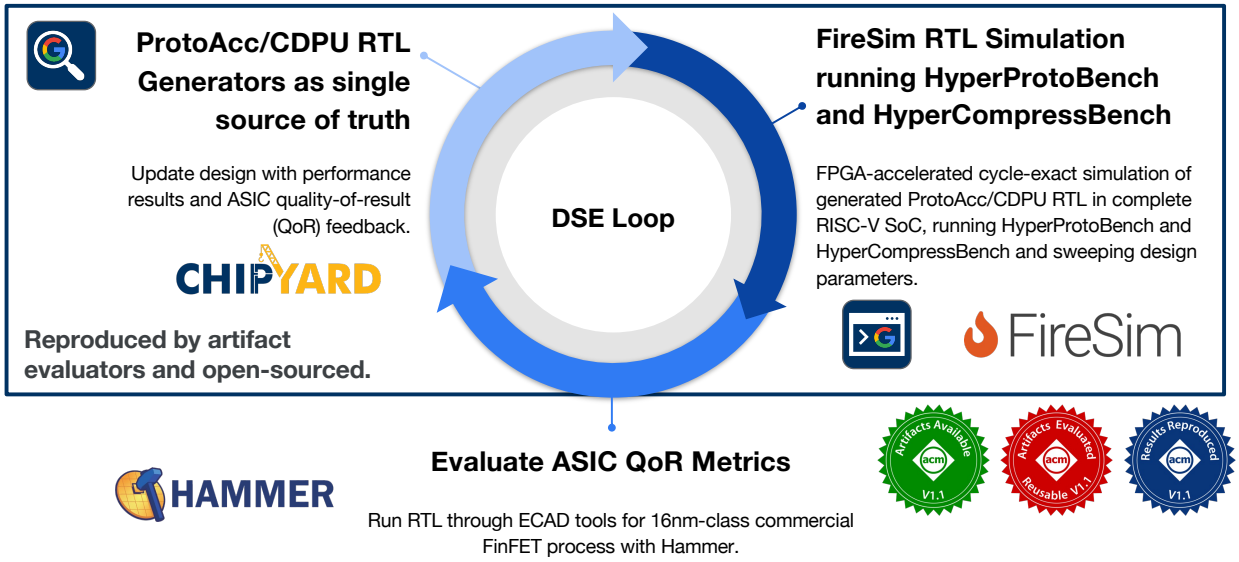


Figure 4.2: Using the *agile* HW/SW co-design methodologies developed in the first half of this dissertation, we demonstrate an implementation-driven HW/SW co-design flow for hyperscale systems, building RTL implementations of the complete Hyperscale SoC design, evaluating it using FireSim FPGA-accelerated simulation running hyperscaler-representative benchmarks, obtain ASIC quality-of-results data for a 16nm-class commercial FinFET process, and finally tape-out Hyperscale SoC in that same process.

Given the powerful hardware/software co-design tools we built earlier, we are then able to connect this data-driven design methodology to our rigorous, implementation-based hardware/software co-design flows, as shown in Figure 4.2. Given the collected hyperscaler insights, we design and

[1]The work in this dissertation was partially done while being affiliated with both Google and UC Berkeley.

implement (in RTL) specialized hardware for the aforementioned domains and integrate it into a customized Chipyard-generated RISC-V system-on-chip. We then run complete software stacks (including the Google-representative benchmarks) on this Hyperscale SoC design pre-silicon using FireSim. We also push our designs through ASIC ECAD tools to obtain ASIC quality-of-result data for our novel hardware designs for a commercial 16nm-class FinFET process and eventually tape-out the design in this process. The next several chapters will discuss the details behind building up Hyperscale SoC.

Chapter 5

FirePerf: Agile cross-stack profiling and co-design for end-host networking

In this chapter, we take the first step towards the broader goal of building Hyperscale SoC, by improving our "vanilla" system-on-chip design to achieve high performance on important networking tasks and introducing the first forms of hardware specialization into our server design[1]. In order to do so in an agile team, we build several new high-fidelity profiling capabilities into FireSim. In the case study, we demonstrate how we use these capabilities to rapidly co-design the hardware and software in our SoC for the networking domain.

5.1 Introduction

As hardware specialization improves the performance of core computational kernels, system-level effects that used to lurk in the shadows (e.g. getting input/output data to/from an accelerated system) come to dominate workload runtime. Similarly, as traditionally "slow" hardware components like networks and bulk storage continue to improve in performance relative to general purpose computation, it becomes easy to waste this improved hardware performance with poorly optimized systems software [29]. Due to scale and complexity, these system-level effects are considerably more difficult to identify and optimize than core compute kernels.

To understand systems assembled from complex components, hardware architects and systems software developers use a variety of profiling, simulation, and debugging tools. Software profiling tools like `perf` [56] and `strace` [199] are common tools of the trade for software systems developers, but have the potential to perturb the system being evaluated (as demonstrated in Chapter 5.4). Furthermore, because these tools are generally used post-silicon, they can only introspect on parts of the hardware design that the hardware developer exposed in advance, such as performance counters. In many cases, limited resources mean that a small set of these counters must be shared across many hardware events, requiring complex approaches to extrapolate information from them [144].

[1]For historical reasons (see Chapter 2), this chapter identifies the server design being developed as "*FireChip SoC*"; this is an early version of *Hyperscale SoC*.

Other post-silicon tools like Intel Processor Trace or ARM CoreSight trace debuggers can pull short sequences of instruction traces from running systems, but these sequences are often too short to observe and profile system behavior. While these tools can sometimes backpressure/single-step the cores when buffers fill up, they cannot backpressure off-chip I/O and thus are incapable of reproducing system-level performance phenomena such as network events.

Pre-silicon, hardware developers use software tools like abstract architectural simulators and software RTL simulation to understand the behavior of hardware at the architectural or waveform levels. While abstract architectural simulators are useful in exploring high-level microarchitectural tradeoffs in a design, new sets of optimization challenges and bottlenecks arise when a design is implemented as realizable RTL, since no high-level simulator can capture all details with full fidelity. In both cases, software simulators are too slow to observe end-to-end system-level behavior (e.g. a cluster of multiple nodes running Linux) when trying to rapidly iterate on a design. Furthermore, when debugging system integration issues, waveforms and architectural events are often the wrong abstraction level for conveniently diagnosing performance anomalies, as they provide far too much detail. FPGA-accelerated RTL simulation tools (e.g. [118]) and FPGA prototypes address the simulation performance bottleneck but offer poor introspection capabilities, especially at the abstraction level needed for system-level optimization. In essence, *there exists a gap* between the detailed hardware simulators and traces used by hardware architects and the high-level profiling tools used by systems software developers. But extracting the last bit of performance out of complete hardware-software systems requires understanding the interaction of hardware and software across this boundary. Without useful profiling tools or with noisy data from existing tools, developers must blindly make decisions about what to optimize. Mistakes in this process can be especially costly for small agile development teams.

To bridge this gap and enable agile system-level hardware-software optimization, we propose and implement FirePerf, a set of profiling tools designed to integrate with FPGA-accelerated simulation platforms (e.g. the open-source FireSim platform discussed in Chapter 2), and provide high-performance end-to-end system-level profiling capabilities without perturbing the system being analyzed (i.e. *out-of-band*). To demonstrate the power of FirePerf, we walk through an extensive case study that uses FirePerf to systematically identify and implement optimizations that yield an 8$\times$ speedup in Ethernet network performance on a commodity open-source RISC-V SoC design. Optimizing this stack requires comprehensive profiling of the operating system, application software, SoC and NIC microarchitectures and RTL implementations, and network link and switch characteristics. In addition to discovering and improving several components of this system in FPGA-accelerated simulation, we deploy one particular optimization in the Linux kernel on a commercially available RISC-V SoC. This optimization enables the SoC to saturate its onboard Gigabit Ethernet link, which it could not do with the default kernel. Overall, with the FirePerf profiling tools, a developer building a specialized system can improve not only the core compute kernel of their application, but also analyze the end-to-end behavior of the system, including running complicated software stacks like Linux with complete workloads. This allows developers to ensure that no new system-level bottlenecks arise during the integration process that prevent them from achieving an ideal speedup.

5.2 Background: Baseline system-on-chip design and modeling flow

In this section, we introduce the tools we use to demonstrate FirePerf as well as the networked RISC-V SoC-based system that we will optimize using FirePerf in our case study.

Target system design: FireChip SoC

In the case study in Chapter 5.4, we will use FirePerf to optimize network performance on a simulated networked cluster of nodes where each simulated node is an instantiation of the open-source FireChip SoC, the default SoC design included with FireSim. The FireChip SoC is derived from the open-source Rocket Chip generator [22], which is written in parameterized Chisel RTL [25] and provides infrastructure to generate Verilog RTL for a complete SoC design, including RISC-V cores, caches, and a TileLink on-chip interconnect. Designs produced with the Rocket Chip generator have been taped out over ten times in academia, and the generator has been used as the basis for shipping commercial silicon, like the SiFive HiFive Unleashed [190] and the Kendryte K210 SoC [122]. To this base SoC, FireChip adds a 200 Gbit/s Ethernet NIC and a block device controller, both implemented in Chisel RTL. The FireChip SoC is also capable of booting Linux and running full-featured networked applications.

Debugging networked system performance with FireSim

In our case study, we will use FireSim's (Chapter 2 and [118, 117, 69]) network simulation capabilities, which allow users to harness multiple cloud FPGAs to simulate clusters of SoCs interconnected by Ethernet links and switches, while maintaining global cycle-accuracy. FireSim also provides some hardware debugging capabilities, including hardware assertion checking, `printf` synthesis [123], and automatic Integrated Logic Analyzer (ILA) insertion. However, these introspection capabilities are generally targeted towards hardware-level waveform-style debugging or functional checks and produce large amounts of output that is not at a useful abstraction level for system-level profiling of hardware running complex software stacks.

Figure 5.1 shows an example FireSim simulation of two FireChip-based nodes running on two cloud FPGAs on Amazon EC2 F1. We will later instrument this simulation with FirePerf to use as the baseline for our case study. Because FireSim exactly simulates the cycle-by-cycle and bit-by-bit behavior of the transformed RTL designs with realistic I/O timing and is sufficiently fast to enable running complete software stacks (Linux + applications), the performance analyses and optimizations we make with FirePerf directly translate to real silicon (that is based on the FireSim-simulated RTL) as we demonstrate at the end of the case study.

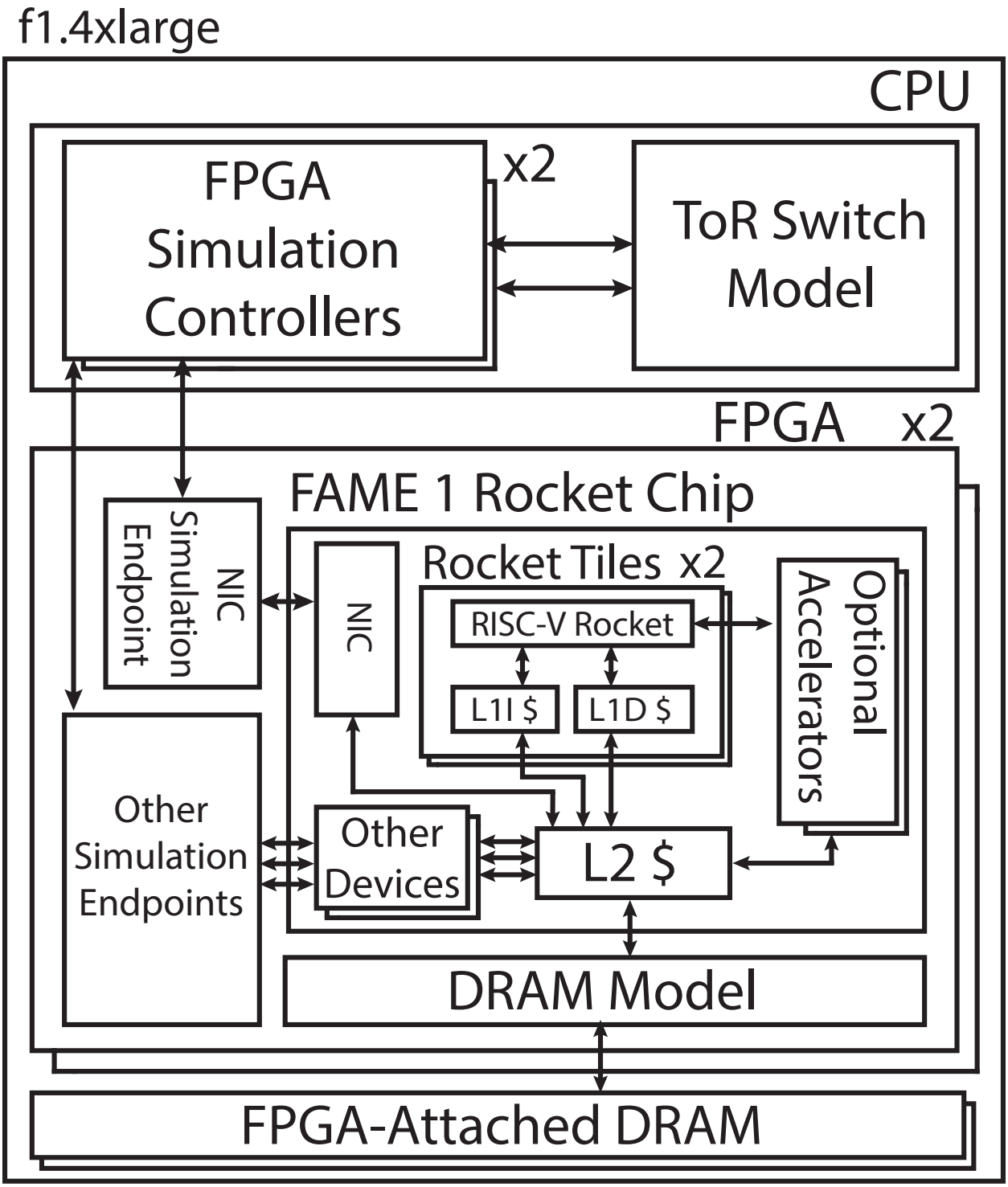


Figure 5.1: FireSim simulation of a networked 2-node, dual-core FireChip configuration on one AWS `f1.4xlarge` instance with two FPGAs, which will form the basis of the system we will instrument, analyze, and improve.

5.3 FirePerf design and internals: High-fidelity pre-silicon profiling tools

FirePerf makes two key contributions to the state-of-the-art in system-level hardware/software profiling by automatically instrumenting FPGA-accelerated simulations to improve performance analysis at the *systems software* and *hardware counter* levels. This section details these two contributions.

Software-level Performance Profiling

FirePerf enables *software-level* performance analysis by collecting cycle-exact instruction commit traces from FPGA simulations *out-of-band* (without perturbing the simulated system) and using these traces to re-construct stack traces to feed into a framework that produces Flame Graphs [83, 38, 37] (e.g. Figure 5.2). These cycle-exact flame graphs allow users to see complete end-to-end Linux kernel (or other software) behavior on simulated SoCs.

Instruction Commit Log Capture with TraceRV

As the first step in constructing flame graphs of software running on a simulated SoC, we implement *TraceRV* (pronounced "tracer-five"), a FireSim endpoint for extracting committed instruction traces from the FPGA onto the host system. Endpoints in FireSim [118] are custom RTL modules paired with a host-software driver that implement cycle-accurate models that interface with the top-level I/Os of the transformed design, like the NIC endpoint shown in Figure 5.1. As a FireSim endpoint, TraceRV is able to backpressure the FPGA simulation when the instruction trace is not being copied to the host machine quickly enough. In essence, when the trace transport is backed-up, simulated time stops advancing and resumes only when earlier trace entries have been drained from the FPGA, maintaining the cycle-accuracy of the simulation. This backpressuring mechanism is built into FireSim and uses its token-based simulation management features.

For the system we improve in the case study, TraceRV attaches to the standard `TracedInstruction` top-level port exposed by Rocket Chip and BOOM [46]. This port provides several signals that are piped out to the top-level of the design for post-silicon debugging, including instruction address, raw instruction bits, privilege level, exception/interrupt status and cause, and a valid signal. In the examples in this chapter, we configure TraceRV to only copy the committed instruction address trace for each core in the simulated system to the host and omit all other trace-port data, though this data remains visible to the TraceRV endpoint for triggering purposes. Since trace ports are standard features integrated in most SoCs, we expect that we will similarly be able to attach TraceRV to other RISC-V SoCs without any modifications to the SoCs.

Directly capturing and logging committed instruction traces has two significant drawbacks. Firstly, with high-speed FPGA-simulators like FireSim, it is easy to generate hundreds of gigabytes to terabytes of instruction traces even for small simulations, which become expensive to store and bottleneck simulation rate due to the performance overhead of transferring traces over PCIe and writing the trace to disk. Furthermore, architects are usually interested in traces for

a particular benchmark run, rather than profiling the entire simulation run, which frequently involves booting the OS, running code to prepare data for a benchmark, running the benchmark, post-processing data, and powering off cleanly. To address this problem, we provide *trigger* functionality in TraceRV, which allows trace logging to begin/end when certain user-defined external or internal conditions are satisfied. When trigger conditions are not satisfied, the TraceRV endpoint allows the simulation to advance freely without the performance overhead of copying traces from the FPGA to the host over PCIe, in addition to avoiding writing to disk. These trigger conditions can be set entirely at runtime (without re-building FPGA images) and include *cycle-count*-based triggers for time-based logging control, *program-counter*-based triggers, and *instruction-value*-based triggers. Instruction-value-based triggers are particularly useful, as some RISC-V instructions do not have side effects when the write destination is the `x0` register and can essentially be used as hints to insert triggers at specific points in the target software with single-instruction overhead. In this particular example using the RISC-V ISA, the 12-bit immediate field in the `addi` instruction can be used to signal 4096 different events to TraceRV or to scripts that are processing the trace data. By compiling simple one-line programs which consist of these instructions, the user can even manually trigger trace recording interactively from within the console of the simulated system. When instruction addresses are known, program-counter based triggers can be used to start and stop commit trace logging without any target-level software overhead. However, using this requires re-analyzing the object code after re-compilation.

The other significant drawback with capturing complete committed instruction traces is that, when initially profiling a system, instruction-level tracing is usually excessively detailed. Function-level profiling and higher-level visualization of hotspots is more useful.

On-the-fly Call-stack Reconstruction

To this end, the TraceRV host-software driver is capable of ingesting per-core committed instruction traces from the FPGA and tracking the functions being called using DWARF debugging information [61] generated by the compiler for the RISC-V executable running on the system (e.g., the Linux kernel). The driver automatically correlates the instruction address with function names and applies DWARF callsite information to construct cycle-exact stack traces of software running on the system. Functions in the stack trace are annotated with the cycle at which they are called and later return. Stack trace construction is done entirely on-the-fly and only function entry points, returns, and cycle-counts are logged to disk by the TraceRV host driver, drastically reducing the amount of data written and improving simulation performance.

Integration with Flame Graph

To visualize this stack trace data in a way that enables rapid identification of hotspots, we use the open-source Flame Graph tool [38, 37]. This produces a flame graph, a kind of histogram that shows the fraction of total time spent in different parts of the stack trace [83].

An example flame graph generated from FirePerf data is shown in Figure 5.2. The x-axis represents the portion of total runtime spent in a part of the stack trace, while the y-axis represents the

stack depth at that point in time. Entries in the flame graph are labeled with and sorted by function name (*not* time). Given this visualization, time-consuming routines can easily be identified: they are leaves (top-most horizontal bars) of the stacks in the flame graph and consume a significant proportion of overall runtime, represented by the width of the horizontal bars. While it cannot be shown in the format of this work, these flame graphs are interactive and allow zooming into interesting regions of the data, with each function entry labelled with a sample count. Traditionally, flame graphs are constructed using samples collected from software profiling tools like `perf`. In our case, the instruction traces collected with FirePerf allow us to construct *cycle-exact* flame graphs; in essence, there is a sample of the stack trace every cycle.

Putting all of these pieces together, FirePerf can produce cycle-exact flame graphs for each core in the simulated system that explain exactly where software executing on the core is spending its time. Because of the cycle-exact nature of the stack-traces available to us, once we identify a hotspot, we can immediately drill down to construct additional visualizations, like histograms of the number of cycles spent in individual calls to particular functions, which is not possible with only sampled data. In our case study, we will use flame graphs as well as custom visualizations generated from data collected with FirePerf instrumentation extensively to understand how software behaves and to identify and implement various optimizations in our system.

Hardware Profiling with *AutoCounter* Performance Counter Insertion

The second key contribution of FirePerf is *AutoCounter*, which enables automatic hardware performance counter insertion via cover points for productive hardware-level performance profiling. Like commit traces, these counters can be accessed *out-of-band*, meaning that reads do not affect the state or timing of the simulated system—counters can be added easily and read as often as necessary.

Cover points are existing boolean signals found throughout the Rocket Chip SoC generator RTL that mark particular hardware conditions intended to be of interest for a verification flow. Unlike assertions, which only trigger when something has gone wrong in the design, cover points are used to mark signals that may be asserted under normal operation like cache hits/misses, coherency protocol events, and decoupled interface handshakes. By default, Rocket Chip does not mandate an implementation of cover points; the particular flow being used on the RTL can decide what to "plug-in" behind a cover point. Unlike `printf`s, which print by default in most simulators, cover points can be inserted into designs without affecting other users of the same RTL codebase. This is especially important in open-source projects such as the Rocket Chip ecosystem. The cover API can also be expanded to allow the designer to provide more context for particular covers.

Performance counters are a common profiling tool embedded in designs for post-silicon performance introspection [155]. However, since these counters are included as part of the final silicon design's area, power, and other budgets, they are generally limited in number and frequently shared amongst many events, complicating the process of extracting meaningful information from them [144]. Pre-silicon use of performance counters in FPGA-simulation is not limited in this way. These counters do not need to be present in the final production silicon, and an unlimited number of counters can be read every cycle without perturbing the results of the simulated system (with the

only trade-off being reduced simulation speed). To enable adding out-of-band performance counters to a design in an agile manner, AutoCounter interprets signals fed to cover points as events of interest to which performance counters are automatically attached. AutoCounter also supports an extended cover point API that allows the user to supply multiple signals as well as a function that injects logic to decide when to increment the performance counter based on some combination of those signals. This allows for a clean separation between the design and instrumentation logic.

AutoCounter's automatic insertion of the performance counters is implemented by performing a transform over the FIRRTL [103] intermediate representation of the target SoC design. With a supplied configuration that indicates which cover points the user wishes to convert into performance counters, FirePerf finds the desired covered signals in the intermediate representation of the design and generates 64-bit counters that are incremented by the covered signals. The counters are then automatically wired to simulation host memory mapped registers or annotated with synthesizable `printf` statements [123] that export the value of the counters, the simulation execution cycle, and the counter label to the simulation host.

By reducing the process of instrumenting signals to passing them to a function and automating the rest of the plumbing necessary to pipe them off of the FPGA cycle-exactly, FirePerf reduces the potential for time-consuming mistakes that can happen when manually wiring performance counters. Unlike cases where mistakes manifest as functional incorrectness, improperly wired performance counters can simply give confusingly erroneous results, hampering the profiling process and worsening design iteration time. This is compounded by the fact that marking new counters to profile *does* require re-generating an FPGA bitstream.

AutoCounter provides users with additional control over simulation performance and visibility. The rate at which counter values are read and exported by the simulation host can be configured during simulation runtime. As exporting counter values requires communication between the FPGA and the simulation host, this runtime configuration enables users to trade off frequency of counter readings for simulation performance.

Also at runtime, collection of the performance counter data can be enabled and disabled outright by the same trigger functionality found in TraceRV. This enables designs to overcome the latency of re-building FPGA bitstreams to switch between different counters—many counters can be wired up at synthesis time, restricted only by FPGA routing resources, and can be enabled/disabled at runtime. Altogether, triggers eliminate extraneous data and enable higher simulation speeds during less relevant parts of the simulation, while enabling detailed collection during regions of interest in the simulation.

Unlike conventional debugging techniques used in FPGA prototypes, such as Integrated Logic Analyzers (ILAs), the FirePerf AutoCounter flow enables a more holistic view of execution, as opposed to the limited capture window provided by ILAs. At the same time, the FirePerf-injected counters still enable flexibility, determinism, and reproducibility (unlike post-silicon counters), while maintaining the fidelity of cycle-exact simulation (unlike software architectural simulators).

5.4 Using FirePerf to optimize Linux networking performance

In this case study, we demonstrate the capabilities of FirePerf by using the FirePerf tools to systematically identify optimization opportunities in the Linux networking stack with a two-node cluster of Ethernet-connected RISC-V SoCs. We walk through, step-by-step, how an architect would harness the FirePerf flow to make decisions about when and what to optimize to produce a specialized hardware/software system for high-bandwidth networking. By using FirePerf, we attain an $8\times$ improvement in maximum achievable bandwidth on a standard network saturation benchmark in comparison to the off-the-shelf open-source version of the SoC and software stack.

Baseline Hardware/Simulator Configuration

Cluster Configuration. We run network bandwidth saturation benchmarks on one and two-node clusters simulated in FireSim. For two-node clusters, the Ethernet network outside of the nodes is modeled with FireSim's built-in network model. The two nodes connect to the same two-port Ethernet switch model using simulated links with 200 Gbit/s bandwidth and 2 µs latency. For reference, our two-node cluster simulations with FirePerf flame graph instrumentation (for two cores on each SoC) and 15 AutoCounter-inserted performance counters run at $\approx 8-10$ MHz, in contrast to the equivalent FireSim simulation without FirePerf which runs at ≈ 40 MHz.

SoC Nodes. Our baseline SoC nodes are instantiations of the open-source FireChip SoC, described earlier in Chapter 5.2. We instantiate two configurations of FireChip, one with a single in-order Rocket core and one with two in-order Rocket cores. Both configurations have private 16 KiB L1 I/D caches, a 1 MiB shared L2 cache, 16 GiB of DDR3 DRAM [34], and a 200 Gbit/s Ethernet NIC. Each design boots Linux and is capable of running complete Linux-based applications.

The `iperf3` benchmark

Our driving benchmark is `iperf3` [64], a standard Linux-based benchmark for measuring the maximum achievable network bandwidth on IP networks. The `iperf3` benchmark is usually run with one `iperf3` process running as a server on one node and another `iperf3` process running as a client on a separate node, with the machines connected by some form of IP networking. In the default configuration, which we use throughout this work, the `iperf3` client is aiming to drive as much network traffic over TCP to the `iperf3` server as possible through one connection.

In our experiments, we configure `iperf3` in two modes. In the *networked* mode, the `iperf3` server and client processes are running on separate simulated nodes (using the previously described two-node FireSim simulation). This measures performance across the Linux networking stack, the Linux NIC driver, the NIC hardware, and the simulated network (links and switches). On the other hand, in the *loopback* mode, both the `iperf3` server and client processes are running on the same simulated node. This allows us to isolate software-only overheads in Linux that do not involve the NIC hardware implementation, the network simulation (links/switches), or the NIC's Linux driver. In essence, the loopback mode allows us to determine an approximate *upper*

bound network performance achievable on the SoC (since only software overhead is involved in loopback), independent of the particular NIC hardware used.

For all experiments, a flame graph and stack trace are generated for each core in each simulated system. For example, a networked `iperf3` run on dual-core nodes will produce 2 cores $\times$ 2 nodes $=$ *four* flame graphs. AutoCounter performance counter traces are also produced in a similar manner, but are produced per simulated node. For both kinds of traces, only the relevant workload is profiled in the trace—the software workload is preceded by a call to a `start-trigger` binary and followed by a call to an `end-trigger` binary, which issue the special instructions described earlier that allow starting/stopping tracing from within the simulation.

Linux 4.15 vs. Linux 5.3

The default Linux kernel version supplied with open-source FireSim is 4.15, the first upstream release to officially support RISC-V. At time of writing, this is also the default kernel version that ships with the SiFive HiFive Unleashed board [190], which we will later use to demonstrate one of the improvements we discover with FirePerf on real silicon. As a precursor to the experimentation in this case study, we first upgrade to 5.3-rc4, the latest mainline branch at time of writing. Unlike 4.15, 5.3 also contains the necessary architecture-specific support for running the commonly used `perf` software profiling tool on RISC-V systems. As we will see in the following baseline comparison, 5.3 also provides a slight (albeit not sufficient) improvement in maximum achievable bandwidth in `iperf3`.

Baseline Performance Results

Prior work [118] has demonstrated that the hardware system we are aiming to optimize is capable of driving in excess of 150 Gbit/sec when running a bare-metal bandwidth-saturation benchmark. However, this same work identifies that the system is only capable of driving 1.4 Gbit/sec over TCP on Linux. We begin our case study by validating this baseline result when running `iperf3` in simulation and analyzing the information we collect with FirePerf.

`iperf3` results on baseline hardware/software configurations

Table 5.1 outlines the sustained bandwidth achieved when running `iperf3` in networked and loopback modes on the two off-the-shelf hardware configurations, single and dual core systems, without any of the optimizations we will identify with FirePerf in this case study. Firstly, the results demonstrate that bumping the kernel version was worthwhile—we see performance improvements or similar performance across-the-board. Consequently, going forward we will only analyze the system running Linux 5.3.

[2] *Reading flame graphs*: The x-axis represents the portion of total runtime spent in a part of the stack trace, while the y-axis represents the stack depth at that point in time. Entries in the flame graph are labeled with and sorted by function name (*not* time). In these flame graphs, colors are not particularly meaningful—they simply help visually distinguish bars.

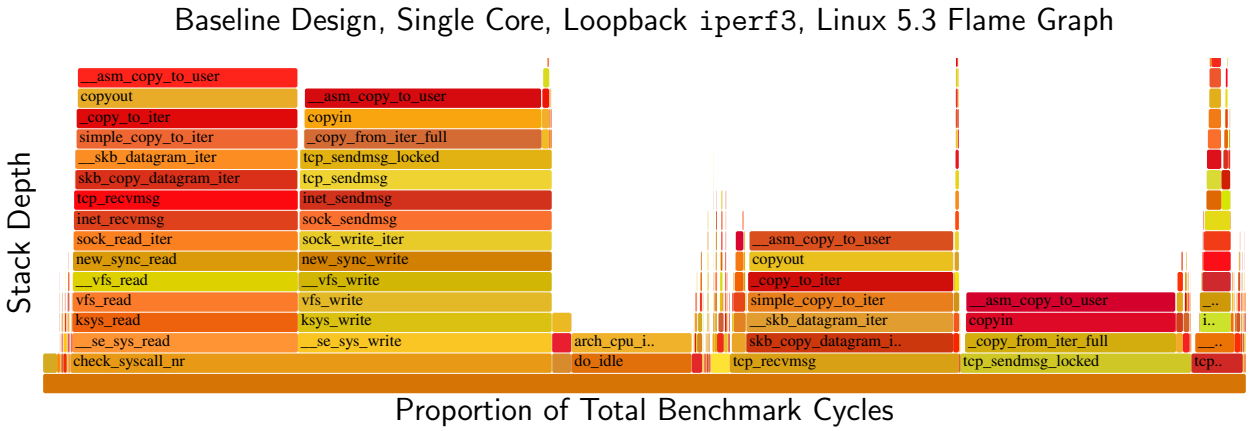


Figure 5.2: Flame graph[2] for Baseline, Single Core, Loopback on Linux 5.3. This flame graph shows that as a percentage of overall time spent in the workload, the `__asm_copy_{to,from}_user` function in Linux dominates runtime. Furthermore, the userspace `iperf3` code consumes a negligible amount of time (it is one of the small, unlabeled bars on the bottom left). This flame graph suggests that even prior to interacting with the NIC driver and NIC hardware, there is a significant software bottleneck hampering network performance.

	Linux 4.15-rc6		**Linux 5.3-rc4**	
	Single (Gbit/s)	**Dual (Gbit/s)**	**Single (Gbit/s)**	**Dual (Gbit/s)**
Networked	1.58	1.74	1.67	2.12
Loopback	1.54	2.95	4.80	3.01

Table 5.1: `iperf3` maximum achieved bandwidth for the baseline open-source hardware/software configuration on two versions of Linux.

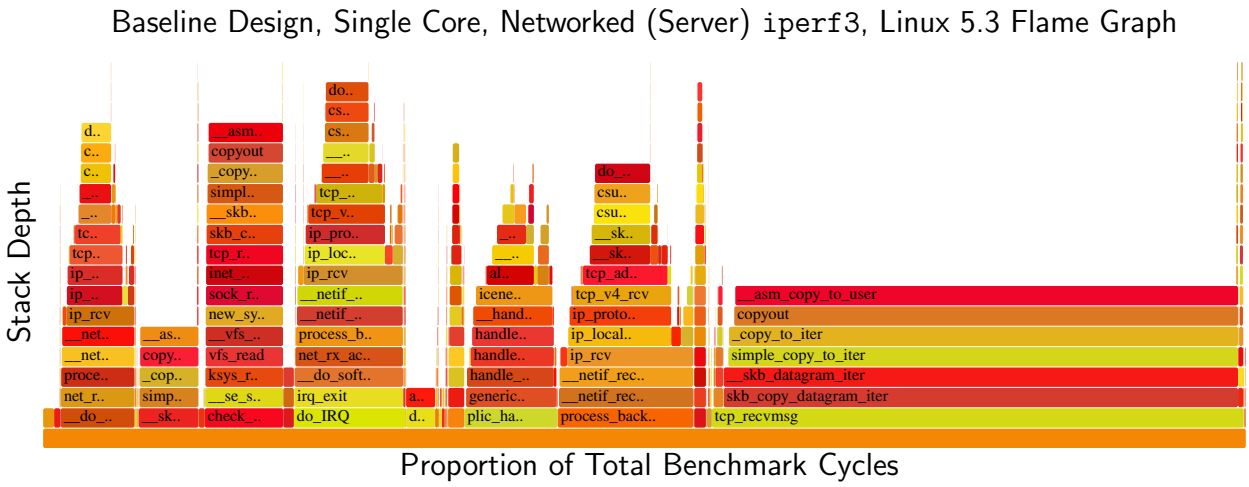


Figure 5.3: Flame graph for Baseline, Single Core, Networked on Linux 5.3 (server-side). This flame graph shows that while not as severe as the loopback case, in the networked case `__asm_copy_{to,from}_user` still dominates runtime as a percentage of overall time spent during the workload. Due to space constraints, we elide the client-side flame graph. It has greater CPU idle time, but `__asm_copy_{to,from}_user` similarly plays a significant role on the client.

Examining the baseline results for Linux 5.3, the best-case performance in the networked case is observed on the dual-core configuration, giving 2.12 Gbit/s, approximately two orders-of-magnitude lower than the hardware is capable of driving when running the bare-metal test. The loopback test, which isolates software overhead and does not involve the NIC hardware or NIC driver, does not fare much better, achieving 4.80 Gbit/sec in the single-core case. The fact that single-core performs better than dual-core is startling. We will find that in this case, the system happened to avoid a particular performance pathology that we will explore in Chapter 5.4. Once this pathology is repaired, we will find that dual-core loopback performs better than single-core loopback, as expected. Overall, this loopback result means that in its current state, regardless of NIC hardware/driver implementations, this system is only capable of achieving around 4.80 Gbit/s of network bandwidth. Flame graphs for two of these runs are shown in Figures 5.2 and 5.3—we will explore these in-depth in later sections.

Is the NIC implementation a bottleneck?

Given that the bare-metal bandwidth test can drive 150 Gbit/s onto the network, we suspect that the NIC hardware is not the primary bottleneck. To validate this and provide a framework for understanding the performance of the NIC hardware for later use, we instrument several potential bottlenecks in the NIC design. Figure 5.4 identifies key parts of the NIC microarchitecture. To understand NIC behavior, we add counters with AutoCounter to track several of these potential bottlenecks:

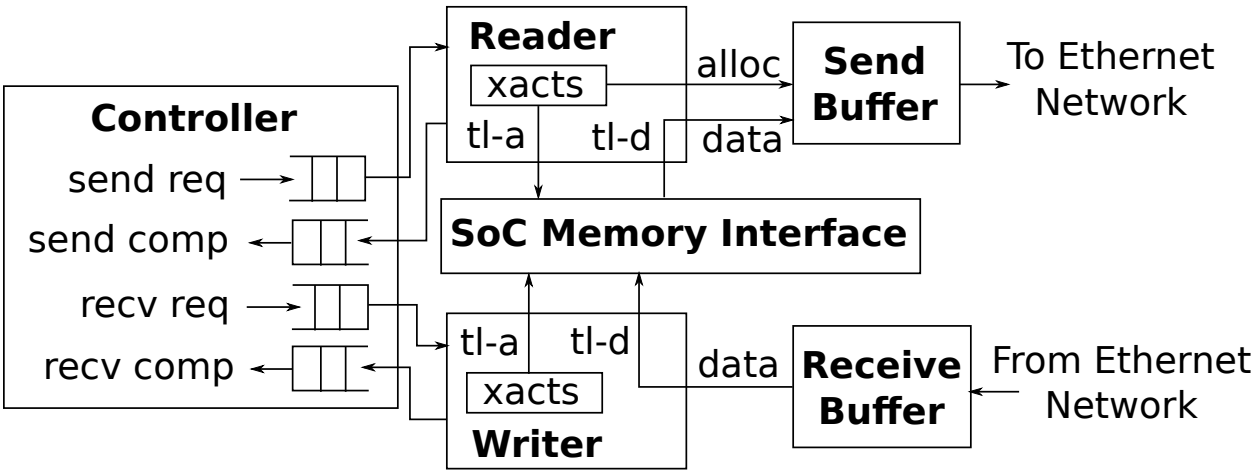


Figure 5.4: FireChip NIC microarchitecture.

- Send request/send completion queue entry counts
- Receive request/receive completion queue entry counts
- Reader memory transactions in-flight
- Writer memory transactions in-flight
- Availability of receive frames and send buffer fullness
- Hardware packet drops

The request and completion queues in the controller are the principal way the device driver interacts with the NIC. To initiate the sending or receipt of a packet, the driver writes a request to the `send` or `recv` request queues. When a packet has been sent or received, a completion entry is placed on the completion queue and an interrupt is sent to the CPU. The reader module reads packet data from memory in response to send requests. It stores the data into the send buffer, from which the data is then forwarded to the Ethernet network. Packets coming from the Ethernet network are first stored in the receive buffer. The writer module pulls data from the receive buffer in response to receive requests and writes the data to memory. If the receive buffer is full when a new packet arrives, the new packet will be dropped.

Figure 5.5 shows that, at this point, the NIC hardware is not a bottleneck when running `iperf3`. The histogram shows the number of cycles spent at different levels of send queue occupancy. We clearly see that the NIC is hampered by software not supplying packets quickly enough, as the queue is empty most of the time. Similarly low utilizations are visible in the other injected performance counters in the NIC.

With this understanding, the following sections will first aim to optimize parts of the software stack before we return to analyzing the hardware. As we optimize the software stack, we will return to Figure 5.5 to demonstrate that our software improvements are improving hardware utilization of the NIC.

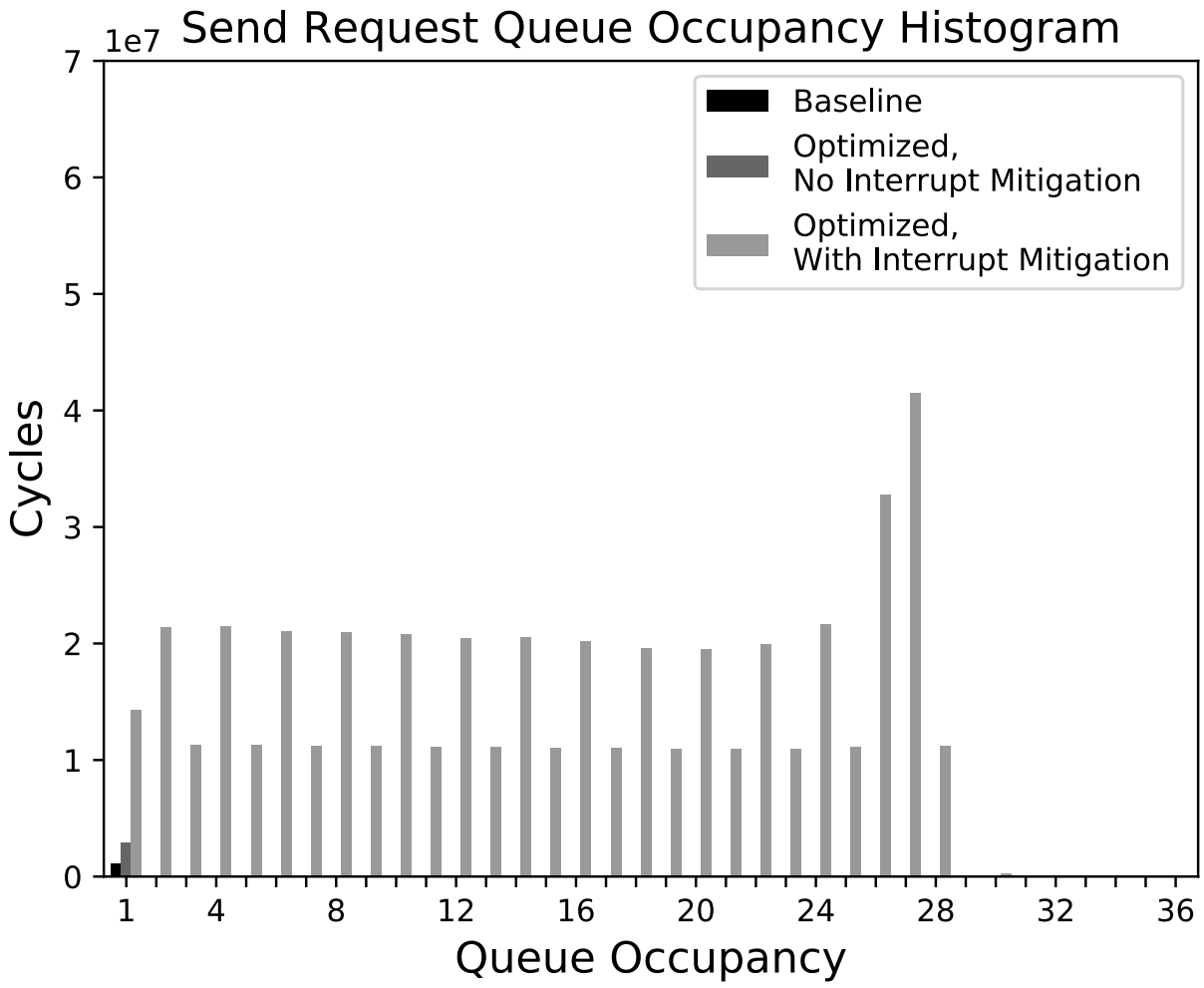


Figure 5.5: NIC send request queue occupancy analysis collected via AutoCounter performance counter instrumentation.

Loopback	Single (Gbit/s)
Baseline	4.80
Baseline `straced`	5.43
Software-optimized `__asm_copy_{to,from}_user`	6.36
Hwacha-accelerated `__asm_copy_{to,from}_user`	16.1

Table 5.2: `iperf3` maximum achieved bandwidth on loopback, single-core for various `__asm_copy_{to,from}_user` optimizations.

Optimizing `__asm_copy_{to,from}_user` in the Linux Kernel

As shown in Table 5.1, even our best-case result (loopback mode) falls orders-of-magnitude short of what the NIC hardware is capable of driving, indicating that Linux-level software overhead is a significant limiting factor. Before experimenting with the NIC hardware, in this section we identify and improve a performance pathology in a critical assembly sequence in the Linux/RISC-V kernel port that significantly improves loopback performance and to a lesser extent performance in the networked case.

The flame graphs in Figures 5.2 and 5.3 show that one function in particular, `__asm_copy_to_user`[3], dominates the time spent by the processor in the loopback case and is nearly half the time spent by the processor in the networked case. This is the assembly sequence[4] in the Linux kernel that implements user-space to kernel-space memory copies and vice-versa. Naturally, if this system is to be optimized, significant improvements need to be made within this function.

Software-only optimization of `__asm_copy_{to,from}_user`

It turns out there is a key performance flaw in the code: When the source and destination buffers do not share the same alignment, the original implementation resorts to a simple byte-by-byte copy, thus severely under-utilizing memory bandwidth. We improve `__asm_copy_{to,from}_user` by enabling word-oriented copies for the 8-byte-aligned subset of the destination. Data re-alignment for 64-bit stores is handled by right-shifting the 64-bit load data and bitwise-ORing with the left-shifted previous word.

Because this pathology is only triggered when the assembly sequence happens to receive unaligned buffers, we see wide variation in loopback performance depending on environmental and

[3]Collectively denoted throughout this work as `__asm_copy_{to,from}_user`, since `__asm_copy_to_user` and `__asm_copy_from_user` are equivalent symbols that refer to the same assembly sequence.

[4]`https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/riscv/lib/uaccess.S?h=v5.3-rc4`

```
# Extract upper part of word i
srl %[temp0], %[data], %[offset]
# Load word i+1 from source
ld %[data], ((i+1)*8)(%[src])
# Extract lower part of word i+1
sll %[temp1], %[data], 64-%[offset]
# Merge
or %[temp0], %[temp0], %[temp1]
# Store to destination
sd %[temp0], (i*8)([%dest])
```

Figure 5.6: Realignment code for optimized __asm_copy_{to,from}_user implementation.

timing conditions. In Chapter 5.4, we will find that traditional profiling tools that run within the simulated system can significantly perturb the outcome of `iperf3` benchmarks, precisely because they impact the proportion of unaligned vs. aligned buffers passed to the __asm_copy_{to,from}_user function. The software-optimized row in Table 5.2 shows the overall speedup achieved by the software fix. Because this is a generic software fix to the Linux kernel, it also improves networking performance on shipping commercial RISC-V silicon, as we will demonstrate in Chapter 5.5.

An additional question is whether our new __asm_copy_{to,from}_user implementation is close to an optimal implementation, i.e., have we exhausted the capabilities of the hardware with our new software routine? To understand this, we add AutoCounter instrumentation with fine-grained triggers inside the Rocket core to collect the following information about the new __asm_copy_{to,from}_user implementation:

- 1.39 bytes are copied per cycle (compared to baseline of 0.847)
- IPC during copies is 0.636
- 56.1% of dynamic instructions are loads/stores during copies
- The blocking L1D cache is 51.1% utilized during copies
- The L1D cache has a 0.905% miss rate during copies

These numbers may not seem outwardly impressive, but for unaligned copies, it should be noted that L1D bandwidth is not the fundamental limiting factor with a single-issue in-order pipeline like Rocket. In the key unaligned block-oriented copy loop, each 64-bit data word requires five instructions to perform realignment, shown in lightly stylized assembly in Figure 5.6.

Assuming an ideal IPC of 1, the maximum throughput is therefore 8/5 = 1.6 bytes per cycle. The actual sustained performance is 86.9% of this peak, with losses due to the usual overhead of loop updates, edge case handling, I-cache and D-cache misses, and branch mispredicts. Even

	Baseline		Hwacha-accel.	
	Single (Gbit/s)	**Dual (Gbit/s)**	**Single (Gbit/s)**	**Dual (Gbit/s)**
Networked	1.67	2.12	2.82	3.21
Loopback	4.80	3.01	16.1	24.9

Table 5.3: `iperf3` maximum achieved bandwidth for the Hwacha-accelerated system, as compared to baseline. The *Single* and *Dual* columns refer to the number of cores

factoring in an additional 13.1% speed-up to hit peak, the pure software implementation falls significantly short of the Hwacha-accelerated version we introduce in the following section.

Hardware acceleration of `__asm_copy_{to,from}_user`

Even with this software fix, the overall potential network performance is still capped at 6.36 Gbit/s. Since the software fix is a relatively compact assembly sequence that we could analyze with instrumentation in the previous section, we know that we are approaching the best-case implementation for the in-order Rocket cores on which our system is based. To achieve further improvement, we augment the system with the open-source Hwacha vector accelerator [137, 136, 135], a co-processor that attaches to Rocket over the RoCC interface. Because we are able to capture system-level profiling information pre-silicon, we can easily demonstrate that we have maximized the capabilities of the baseline design and thus can trivially justify the inclusion of this accelerator in our specialized system for network processing, assuming we see further speedups from its inclusion.

We write a vectorized implementation of `__asm_copy_{to,from}_user` that dispatches the copying to Hwacha, which can achieve a much higher memory bandwidth utilization than the in-order Rocket cores. Table 5.3 shows the significant speedups achieved with the integration of Hwacha into the design, across the two hardware configurations and two `iperf3` modes. While prior work has pointed out the need for systems-level accelerators for `memcpy()` [110], FirePerf allows us to systematically identify and justify when such improvements are necessary, pre-silicon. As we move forward to continue optimizing network performance on our system, we assume from this point forward that we are running on a system with hardware-accelerated `__asm_copy_{to, from}_user` calls with Hwacha.

Comparing with in-band tracing: Tracing with `strace`

As a brief aside, let us explore the challenges involved in solving the `__asm_copy_{to,from}_user` performance pathology discussed in the previous section using an existing profiling/tracing tool. As a post-silicon alternative to FirePerf, one common tool used to understand application behavior

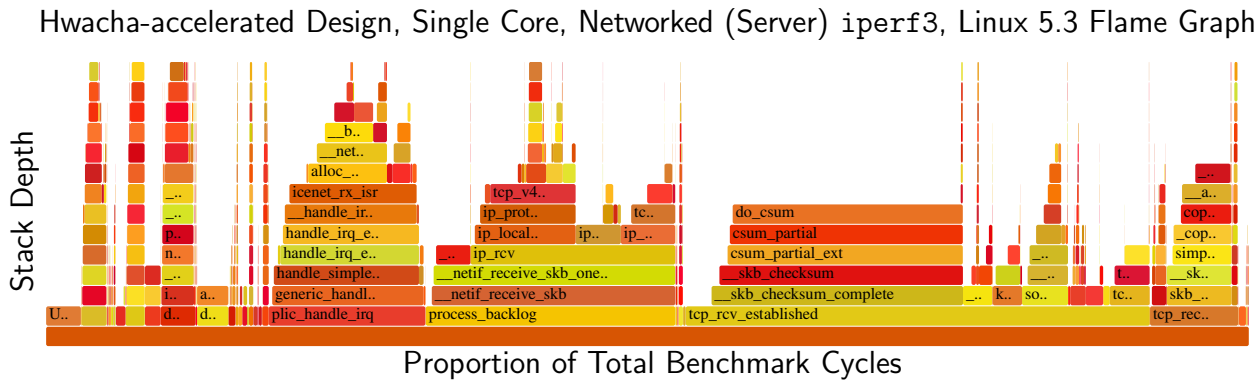


Figure 5.7: Flame graph for Hwacha-accelerated, Single Core, Networked on Linux 5.3 (server-side). This flame graph shows that a previously insignificant software routine consumes a significant number of cycles in the networked case once kernel-userspace copies are accelerated: `do_csum`. Due to space constraints, we again elide the client-side flame graph—it has greater CPU idle time, but `do_csum` similarly plays a significant role on the client.

is `strace`, a Linux utility that allows developers to inspect system calls made by a user program, such as reads and writes to a network socket.

When running `iperf3` within `strace` on our system without the optimizations introduced in the previous section, we noticed a startling result: `iperf3` performance *improved* under `strace`, contrasting with the common wisdom that profiling tools introduce overheads that worsen the performance of the system being measured. The "Baseline Loopback `straced`" row in Table 5.2 demonstrates this result. As it turns out, while running with `strace`, the buffers passed to the `__asm_copy_{to,from}_user` sequence in the course of running `iperf3` happen to be aligned more often, avoiding the performance pathology we discovered in the previous section! We confirmed this result both by logging addresses within the kernel and observing flame graphs with and without `strace` measuring the `iperf3` run. The flame graphs confirm that `__asm_copy_{to,from}_user` is able to move more bytes per cycle in the `straced` case, suggesting that the byte-oriented copy is being avoided. Unlike `strace`, because FirePerf is an *out-of-band* tool, there is no such danger of perturbing the outcome of the system being measured.

Checksum Offload

Now that `__asm_copy_{to,from}_user` has been optimized, there are no further obvious software routines to optimize in the loopback-only `iperf3` runs. Going forward, we will focus only on the networked runs of `iperf3` that involve the actual NIC hardware. Looking at the updated flame graph for the Hwacha-accelerated networked `iperf3` case in Figure 5.7, we find that one new software routine appears to consume significant cycles—`do_csum`, which implements checksumming for network protocols in Linux. After implementing hardware checksum offload in the NIC, we

see improved performance in the networked case, as shown in the "+Checksum Offload" row of Table 5.4.

Interrupt Mitigation

Once `do_csum` is optimized away, there is no clear hotspot function visible in the flame graphs. Instead, we again analyze the performance counters for the NIC's send-request queue to gauge the impact of our current optimizations on NIC utilization. The "Optimized, No Interrupt Mitigation" bars in Figure 5.5 reflect the queue utilization with Hwacha-accelerated copies and checksum offload. We can see that it has improved somewhat from the baseline, implying that software is doing a better job feeding the NIC, but still remains relatively low.

We examine the low-level function in the Linux device driver (`icenet_start_xmit`) responsible for passing an outgoing packet to the NIC. By directly extracting information from the trace itself about the timing and control flow through `icenet_start_xmit`, we find that the method by which the NIC and driver acknowledge request completions introduces significant overhead. There are almost exactly twice as many jumps entering the body of `icenet_start_xmit` as packets sent and a bimodal distribution of call lengths centered around 2000 cycles and 8 cycles. Looking at the detailed trace, the `icenet_start_xmit` function, which should be called repeatedly in a tight loop for bulk packet transmission, is almost always interrupted by the NIC to trigger buffer reclamation for completed sends. These frequent interrupt requests (IRQs) prevent packets from being batched effectively.

With this insight, we modify the Linux NIC driver to support the Linux networking subsystem's NAPI facility, which adaptively disables device IRQs and switches to polling during high activity. This significantly reduces IRQ load at the cost of some latency, allowing us to reach the results shown in row "+Interrupt Mitigation" in Table 5.4. The "Optimized, With Interrupt Mitigation" bars in Figure 5.5 represent NIC queue occupancy once interrupt mitigation is enabled. We see a significant increase in queue occupancy which manifests as improved network performance.

Conversely, it would be difficult to observe this phenomenon with the standard `perf` tool, whose sampling mechanism (being based on supervisor timer interrupts) lacks any visibility into critical regions. In particular, "top-half" IRQ handlers, which run with interrupts globally disabled, would be completely absent from the `perf` capture. Mitigating this deficiency requires repurposing platform-specific non-maskable interrupts (NMIs). However, these are not supported generically in the `perf` framework and are not enabled by all architectures. Since FirePerf is able to precisely record all executed instructions *out-of-band*, the true IRQ overhead becomes obvious.

A curious result is the overall lack of improvement on the single-core system. We instrument the FireSim switch model to dump all Ethernet traffic to a `pcap` file. Analysis of the TCP flow and kernel SNMP counters indicate a higher transfer rate from the `iperf3` client, but the server becomes compute-bound and cannot keep pace. With very rare exceptions, all TCP segments arrive in order, yet discontinuities in the Selective Acknowledgement (SACK) sequences suggest that packets are being internally dropped by the server during upper protocol processing. This leads to frequent TCP fast retransmissions (1% of packets) that degrade the effective bandwidth.

Jumbo Frames

From the flame graph and performance counters at this point (not shown), we no longer see obvious places for improvement in the software or hardware. With the understanding that the bottleneck is the receive path in software, one further avenue for improvement is to amortize the overhead of individual packet processing with a larger payload. By default, our system uses the standard Ethernet Maximum Transmission Unit (MTU) size of 1500, which sets a limit on the length of an Ethernet frame. However, the loopback driver, which produces the upper-bound result in excess of 20 Gbit/s from Table 5.3, defaults to an MTU of 65536. The Ethernet equivalent is using jumbo frames with a commonly chosen MTU of 9000. This is a standard optimization for high-performance networking in datacenter contexts—for example, Amazon's networking-focused EC2 instances default to an MTU of 9001 [157]. Given this insight, we implement jumbo frame support in our NIC RTL and driver. The final speedup is shown in row "+Jumbo Frames" of Table 5.4. In combination with earlier improvements, the system is now capable of 17.5 Gbit/s on `iperf3`.

Final performance results

Table 5.4 summarizes the optimizations discovered with FirePerf throughout the case study and their respective contribution to the overall speedup. As is generally the case with system-integration-level improvements, there is no silver-bullet—the overall speedup consists of several small speedups compounded together.

To demonstrate that our final results are realistic, we compare against a real datacenter cluster by running `iperf3` with identical arguments as our prior simulations on two `c5n.18xlarge` instances on Amazon EC2. These instances are optimized for networking, with AWS Nitro hardware acceleration [28], and support 100 Gbit/s networking. We also place the instances in a placement group to ensure that they are physically co-located. By default, these instances have jumbo frames (9001 MTU) enabled and give a result of 9.42 Gbit/s on `iperf3`. Reducing the MTU to the standard Ethernet MTU (1500), we see a result of 8.61 Gbit/s. Returning to our simulated cluster, when we configure our simulated nodes to have a similar end-to-end network latency as the two nodes on EC2, we obtain results of 17.6 Gbit/s and 6.6 Gbit/s for jumbo frames and standard MTU, respectively. Naturally, the EC2 instances have to contend with a less-controlled environment than our simulated nodes. However, these results show that our achieved bandwidth is reasonable for a single network flow between two datacenter nodes.

5.5 Applying findings to commercial chips

The software-only optimization in the Linux kernel `__asm_copy_{to,from}_user` function that we developed in the case study applies to RISC-V systems in general, not only the FireChip SoC. To demonstrate the impact of this improvement on real silicon, we apply our patch to the `__asm_copy_{to,from}_user` function to the Linux kernel for the SiFive HiFive Unleashed board, a commercially available RISC-V platform that includes a Cadence Gigabit Ethernet MAC. We then connect the HiFive Unleashed board directly to an x86 host with an Intel Gigabit NIC and run `iperf3` in the

	Single Core (Gbit/s)	**Dual Core (Gbit/s)**
Baseline	1.67	2.12
+Hwacha-accel. __asm_copy_{to,from}_user	2.82	3.21
+Checksum Offload	4.86	4.24
+Interrupt Mitigation	3.67	6.73
+Jumbo Frames	12.8	17.5

Table 5.4: Final `iperf3` maximum achieved bandwidth results for each optimization. Features are cumulative (i.e. "+Interrupt Mitigation" also includes "+Checksum Offload").

HiFive Role, MTU	**Baseline** __asm_copy_{to,from}_user	**Optimized** __asm_copy_{to,from}_user	**Speed-up**
Server, 1500	572 Mbit/s	935 Mbit/s	1.63
Server, 3000	553 Mbit/s	771 Mbit/s	1.39
Client, 1500	719 Mbit/s	739 Mbit/s	1.03
Client, 3000	483 Mbit/s	829 Mbit/s	1.72

Table 5.5: `iperf3` performance gain on commercial RISC-V silicon by deploying __asm_copy_{to,from}_user fix discovered with FirePerf.

same networked mode as our case study. Table 5.5 shows the result of this benchmark, before and after software __asm_copy_{to,from}_user optimization. We alternate between the HiFive and x86 host being client/server and vice-versa, as well as trying a large MTU. We see improvements in all cases, and in the "Server, 1500 MTU" case, the HiFive is now able to saturate its link.

5.6 Related work

Prior work has demonstrated the use of various profiling techniques to analyze system-level performance bottlenecks, including using pre-silicon abstract software simulation, as well as post-silicon software-level profiling and hardware tracing.

Abstract system-level simulators have long been used in the architecture and design automation communities for performance estimation and analysis [36, 231, 156, 222, 174, 187, 163, 57, 104]. In particular, [35] used system simulation to evaluate the interaction between the OS and a 10

Gbit/s Ethernet NIC. In contrast, our case study does not rely on timing models of particular NIC components but rather optimizes a full SoC/NIC RTL implementation that can be taped-out. FirePerf targets a different phase of the design flow. FirePerf focuses on optimizing the design at the stage where it is being implemented as realizable RTL, after the high-level modeling work has been done. Implementing a design exposes new bottlenecks, since no high-level simulator can capture all details with full fidelity. Other prior work focuses on debugging in the context of co-simulation frameworks [183, 6], rather than application performance analysis.

Sampling-based methods have also been widely used for profiling [148, 62, 169, 223]. These are proficient at identifying large bottlenecks but may not capture more intricate timing interactions, such as latency introduced by interrupts during the NIC transmit queuing routine as identified using FirePerf.

At the post-silicon software profiling level, in addition to coarser-grained tools like `strace` and `perf`, other work [126] has enabled cycle-accurate profiling of the FreeBSD networking stack. This work measures function execution time on real hardware by using the processor timestamp register, which is incremented each clock cycle. In order to reduce the overhead of reading the timestamp register, they profile only functions that are specified by the user. In contrast, FirePerf's out-of-band instrumentation allows for cycle-accurate profiling of the entire software stack with no overhead, and therefore does not require prior knowledge about details of the software networking stack. Other work aims to perform out-of-band profiling post-silicon. [225] uses hardware tracing of network frames and network emulation techniques to optimize a system for 10 Gbit/s Ethernet but does not directly profile software on network endpoints. Additional case-studies demonstrate the intricate methods required for system-level post-silicon profiling and performance debugging [149].

Some methods to reduce the overhead of software profiling and debugging come in the form of architectural support for software debugging such as IWatcher [232]. The triggers used in FirePerf use similar concepts to those in IWatcher for targeted observability. Other techniques exploit side channels for out-of-band profiling [188] at the cost of coarser granularity and non-negligible imprecision.

Prior FPGA-accelerated simulators [216, 50, 49, 165, 204] do not directly simulate tapeout-ready RTL like FireSim, but rather use handwritten FPGA-specific models of designs. Additionally, most of these works do not mention profiling or only suggest it as an area for future exploration, with the exception of Atlas [216], which includes profiling tools particularly for transactional memory, rather than automated general-purpose profiling. By adding TraceRV and AutoCounter within the FireSim environment, FirePerf addresses a common complaint against FPGA prototypes and simulators, providing not just high fidelity and simulation performance (10s of MHz with profiling features), but also high levels of introspection.

The results of our case study have also emphasized the importance of offloading networking stack functions to hardware and support further research into balancing software and hardware flexibility in SmartNICs [70], as well as specialization for network-centric scale-out processors [67].

5.7 Discussion and future work

Open-sourcing. The FirePerf tools are open sourced as part of FireSim, which is available on GitHub: `https://github.com/firesim/firesim`. Documentation that covers how to use FireSim and FirePerf is available at `https://docs.fires.im`. The artifact appendix at the end of this chapter also provides instructions for reproducing the experiments in this work.

Extensibility. Several opportunities exist to extend the FirePerf tools to gain even more introspection capability into FPGA-simulated designs. For example, we describe FirePerf in this chapter in the context of optimizing a networked RISC-V SoC. However, because the ISA-specific components of FirePerf stack unwinding are provided by common libraries (e.g. `libdwarf` and `libelf`), other ISA support is possible.

Furthermore, in this work we were primarily interested in analyzing OS-level overheads. As shown in the flame graphs in the case study, time spent in userspace is a small fraction of our total CPU cycles. Accordingly, the current stack trace construction code does not distinguish between different userspace programs, instead consolidating them into one entry. Handling userspace more effectively will require extensible plugin support per-OS.

Lastly, while the designs we simulate in the case study supply top-level trace ports, FIR-RTL passes available in FireSim can also automatically plumb out signals (like committed instruction PC) from deep within arbitrary designs, removing the need to rely on having a standard `TracedInstruction` port in the SoC design.

Achieving *Introspection Parity* between FPGA and Software Simulation. Traditionally, FPGA-simulators and open-source RTL have not been widely adopted in architecture research due to the infrastructural complexity involved in deploying them. With cloud FPGAs and FireSim, many of these difficulties are abstracted away from the end-user. However, prior to FirePerf, there remained a gap between the level of introspection into design behavior available in FPGA-simulation of open hardware vs. abstract software simulators. We believe that open-source tools like FirePerf can make profiling of RTL designs in FPGA simulation as productive as software simulation. Furthermore, cover points can provide a consistent interface for open-source hardware developers to expose common performance metrics to FPGA simulation environments for use by architecture researchers, bridging the gap between open-RTL and architecture research.

Full-system workloads vs. Microbenchmarks. A key case for FPGA-accelerated simulation is that FPGA simulators have sufficiently high simulation rates to enable running real workloads. As our case study has shown, the full range of emergent behavior of a pre-emptive multitasking operating system is difficult to re-create in a microbenchmark that can be run on software simulators. Instead, when feasible, running FPGA-accelerated simulation with introspection capabilities is a productive way to rapidly understand system behavior.

5.8 Conclusion

In this work we proposed and implemented FirePerf, a set of profiling tools designed to integrate with FPGA-accelerated simulation platforms, to provide high-performance end-to-end system-

level profiling capabilities without perturbing the system being analyzed. We demonstrated FirePerf with a case study that used FirePerf to systematically identify and implement optimizations to achieve an 8× speedup in Ethernet network performance on an off-the-shelf open-source RISC-V SoC design.

5.9 Retrospective

Further extending our goals of building an agile hardware/software co-design methodology, FirePerf demonstrated that our new profiling tools could enable a few Ph.D. students to rapidly deep-dive into a domain, even one with complicated interactions between software (the kernel) and a new hardware design (our NIC), and implement a high-performance solution. Having been upstreamed to FireSim in 2020, several FireSim users have since written peer-reviewed publications that use the FirePerf tools and extend FirePerf's capabilities. This work also underwent the ASPLOS conference's first-ever artifact evaluation process and received all available badges.

Chapter 6

A hardware accelerator for protocol buffers

In this chapter, we employ the data-driven hardware/software co-design methodology described earlier to understand and accelerate a critical system-level overhead (or datacenter tax) in hyperscale servers: software serialization and deserialization via frameworks such as Google's Protocol Buffers. Using insights from Google's hyperscale datacenter fleet, we design custom hardware to accelerate protocol buffers and evaluate it in the context of Hyperscale SoC. Note that representations of fleet data in this chapter are as they were in the original publication of this chapter at MICRO 2021 [113].

6.1 Introduction

Building internet-scale applications for WSCs requires efficient communication between software components (i.e., services) in a distributed environment, which is commonly achieved via remote procedure call (RPC). Because the remote callee cannot directly access the caller's memory space to read arguments and supply a response, and may even be written in a different programming language, exchanged data must undergo conversion to and from a shared interchange format, via *serialization* and *deserialization* operations. In addition to inter-service communication via RPC, serialization and deserialization are also commonly used when persisting data to durable storage.

To ensure that serialization and deserialization are handled in a principled way across the multitude of services and data producers/consumers running in a warehouse-scale computer, service developers employ a common serialization framework, which ensures interoperability between components by pairing a standardized wire format with language-specific APIs that allow applications to produce and consume serialized objects. A vast number of these frameworks have been created [173, 43, 17, 71, 72, 102, 65, 224], constituting a large design space encompassing tradeoffs in performance, flexibility, ease-of-use, backwards compatibility, and schema evolution. In a hyperscale context, backwards compatibility and schema evolution become particularly important to manage complexity, build reliable systems, and ensure long-term accessibility of data persisted to durable storage [209, 5].

Naturally, this functionality comes at a performance cost—prior work has shown that around

5% of fleet-wide cycles in Google's Warehouse Scale Computers (WSCs) were spent in the Protocol Buffers ("protobuf") serialization framework in 2015 [110]. In 2020, Facebook identified that serialization and deserialization consume on average over 6% of cycles across seven key microservices in their fleet [198].

Fortunately, the warehouse-scale context is a natural environment for hardware specialization [109, 73, 23, 178, 70, 31, 146] as the cost of building custom processors is amortized over the high volume of deployed hardware systems. To understand the trade-offs and opportunities in hardware acceleration for serialization frameworks, we present the first in-depth study of serialization framework usage at scale by characterizing protobufs usage across Google's datacenter fleet (Chapter 6.3) and use this data to construct HyperProtoBench, an open-source[1] benchmark representative of key serialization-framework user services at scale (Chapter 6.5). In doing so, we also identify key insights that challenge common assumptions about serialization framework usage (Chapter 6.3).

We use these insights to co-design hardware and software to develop a novel hardware accelerator for protobuf message serialization and deserialization, implemented in Chisel RTL [25] and integrated into a Linux-capable RISC-V SoC [14] (Chapter 6.4). Applications can easily harness the accelerator, as it integrates with a modified version of the open-source protobuf library and is wire-compatible with standard protobufs. We have fully open-sourced[2] our RTL, which, to the best of our knowledge, is the only such implementation currently available to the community.

We also present a first-of-its-kind end-to-end evaluation of our entire RTL-based system running hyperscale-derived benchmarks and microbenchmarks (Chapter 6.5 and Appendix B). We boot Linux on the system using FireSim [118] to run these benchmarks and implement the design in a commercial 16nm-class FinFET process to obtain area and frequency metrics. We demonstrate an average 6.2× to 11.2× performance improvement vs. our baseline RISC-V SoC with BOOM OoO cores [230] and despite the RISC-V SoC's weaker uncore/supporting components, an average 3.8× improvement vs. a Xeon-based server.

In addition to advancing the state-of-the-art in serialization framework acceleration, this work is the first to demonstrate the power of combining a data-driven hardware-software co-design methodology based on large-scale profiling with the promise of agile, open hardware development methodologies [88, 134]. In this vein, our entire evaluation flow (RTL, benchmarks, including hyperscale-derived benchmarks, and supporting software and simulation infrastructure) has been open-sourced for the benefit of the research community and our results have been reproduced by external artifact evaluators (Appendix B).

6.2 Protobuf serialization library overview

The protobuf library is an open-source, schema-oriented, data and service description system [173]. Protobufs are widely used for service-oriented design in modern hyperscale systems, including at Google. Protobufs are also used for in-memory data representation, persisting data to durable

[1] `https://github.com/google/HyperProtoBench`. See Appendix B for archival URL.

[2] `https://github.com/ucb-bar/protoacc`. See Appendix B for archival URL.

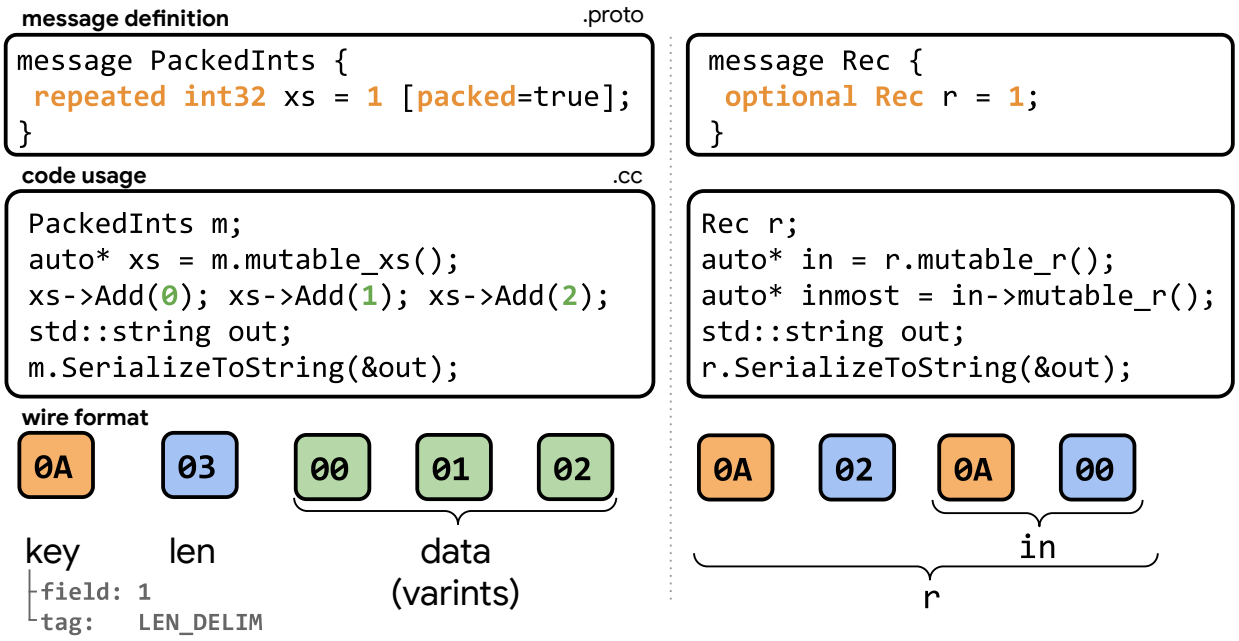


Figure 6.1: Encodings with repeated and recursive types. Empty messages (`inmost`) take no bytes in encoded form.

storage, and as a schema for columnar storage (e.g. Google's Dremel/BigQuery [152, 151]). A protobuf user defines the contents of a message in a `.proto` file written in the protobuf language, either `proto2` or `proto3`[3]. The protobuf compiler (`protoc`) ingests `.proto` files and generates language-specific code to allow user programs to populate, read, and perform other operations on protobuf messages.

Message structure

Schema and message definition

A protobuf message is a collection of fields. In the protobuf schema, each message field has a type, name, field number, and potential qualifiers including `optional`, `required`, and `repeated` (with `packed` for a more efficient encoding). Scalar field types include doubles, floats, various variable and fixed-width integer types, bools, strings, and bytes. The "Protobuf Type" column in Table 6.1 lists these types. A field's type can also be a user-defined message type, allowing for messages to contain sub-messages; messages may be nested arbitrarily deeply and recursively structured. The `repeated` qualifier marks that a field is a vector of elements of its assigned type, which can also be a user-defined message type. The top row of Figure 6.1 shows two example message definitions.

[3]As discussed in Chapter 6.3, the vast majority of protobuf usage in Google's fleet is `proto2`. Thus, "protobufs" implicitly refers to `proto2` in the rest of this chapter.

Performance-similar Types	**Protobuf Type** (includes repeated of each type)	**Sizes** (bytes)
bytes-like	`bytes`, `strings`	See Fig. 6.4c buckets
varint-like	{`s`,`u`}`int`{`64`,`32`}, `int`{`64`,`32`}, `enum`, `bool`	1-10, by 1
float-like	`float`	4
double-like	`double`	8
fixed32-like	`fixed32`, `sfixed32`	4
fixed64-like	`fixed64`, `sfixed64`	8

Table 6.1: Classification of protobuf field types.

This structure enables forward portability and schema evolution. Namely, fields are numbered for stability across field name changes, and fields may be optionally present, enabling sparsity for deprecated/unused fields. Schema evolution, upgrade paths, and host language integration are critical for productively using a serialization framework at hyperscale, where services cannot be monolithically upgraded, and persisted data must be highly available for long periods of time [5].

Wire format

Before we discuss the wire format, it is important to note that variable-length integers ("varints") are used heavily in the protobuf wire format. The protobuf varint algorithm repeatedly consumes 7 bits at-a-time in a loop from the least-significant side of a fixed-width input value until no non-zero bits remain. For each 7-bit group, it outputs a byte containing the original bits and a continuation bit, which if set, indicates that more bytes follow. As we will see, varint handling is a prime candidate for acceleration—fixed-function hardware can easily handle varint encoding/decoding in a single cycle.

On the wire, protobuf messages appear as a sequence of bytes containing a set of (key, value) pairs that represent fields in the message. Each field's key is a varint-encoded version of the field number concatenated with a three-bit *wire* type. Wire types can be one of: varint (field types {`s`,`u`}`int`{`64`,`32`}, `int`{`64`,`32`}, `enum`, and `bool`), 64-bit (field types `double` and `(s)fixed64`), length-delimited (field types `string`, `bytes`, sub-messages, and packed repeated fields), start group (deprecated), end group (deprecated), and 32-bit (field types `float` and `(s)fixed32`). A critical observation from this mapping is that the wire type is *not* sufficient to determine the language/schema type of a field. For the 32-bit and 64-bit wire types, C++ values are directly copied into the wire format. For the varint wire type, the *varint encoding* is applied to the C++ values before they are copied to the wire format. The values of the length-delimited wire type first contain a varint-encoded length in bytes, which represents either the length of a string or byte array, the length of a sub-message, or the length of a packed repeated field. This length is followed by either the string or bytes data, the wire-format version of a sub-message, or encoded values in a packed repeated field. Finally, unpacked repeated fields appear on the wire as multiple (key, value)

pairs that all have the same key. The bottom row of Figure 6.1 shows examples of two messages encoded in the wire format.

In-memory format

As previously mentioned, given a message schema, the protobuf compiler will generate language-specific code for each message type. For example, for C++, the compiler generates a class for each message type which encapsulates the field data. Users expect to work with protobuf messages as standard C++ objects: scalar fields are stored as the expected C++ primitive type, string/byte fields are stored as `std::string`s, repeated fields are stored similar to vectors, and sub-messages are stored as pointers to objects of their corresponding type. All members are wrapped in accessors (e.g., setters, getters). The middle row of Figure 6.1 shows examples of two messages used in C++ code.

Serialization and deserialization

The two key operations in protobufs are *serialization* and *deserialization*. Serialization converts the in-memory, language-specific protobuf message representation (Chapter 6.2) to the standard protobuf wire format (Chapter 6.2). This wire-format version of a message can then be exchanged with any other program that uses protobufs, regardless of programming language, host machine, operating system, and compiler. To unpack the wire format into a usable object again, the *de*serialization process converts a wire-format message back to the in-memory language-specific protobuf object.

Serialization and deserialization are inverse operations, but deserialization is more complex for two reasons. Firstly, deserialization is inherently a serial process: the deserializer receives a single stream of bytes and the key (and potentially, value) of the N^{th} field in the encoded format must be decoded before the $(N+1)^{th}$ field, as the location of the $(N+1)^{th}$ field is unknown until the size of the N^{th} field is known (based on wire-type or explicit length). On the other hand, serialization has ample opportunity for parallelism: serialization of individual fields can be performed in parallel with one final serial step that concatenates the serialized fields into one output buffer.

Secondly, deserialization requires the accelerator to construct objects in the in-memory language format (including e.g., `std::string` objects in C++) and allocate memory for them; serialization only needs to *traverse* language-format objects.

Arena allocation

One notable performance optimization available in upstream protocol buffers is arena allocation [42], which reduces message construction/destruction overheads by pre-allocating a large chunk of memory called the *arena*. Allocation of individual messages in the arena is simplified to a pointer increment. The accelerator we implement uses its own form of arena allocation, as discussed in Chapter 6.4.

6.3 Profiling protobuf usage at hyperscale

In this section, we explore the usage of Protocol Buffers at scale across Google's datacenter fleet to motivate requirements for a hardware accelerator for serialization and deserialization and quantify accelerator design trade-offs.

Data sources

We rely on three internal data sources at Google to glean insights on protobuf usage at scale: Google-Wide Profiling (GWP) CPU cycle profiles, `protobufz`, and `protodb`.

Google-Wide Profiling (GWP) CPU cycle profiles

CPU cycle profiles are collected from machines across Google's fleet using Google-Wide Profiling (GWP) [180]. The collected profiles include workload names, stack traces, and cycle counts, which allow us to identify where CPU time is spent in software. In particular, this data allows us to identify how much time is spent in different operations inside the protobuf library and generated code, including serialization, deserialization, and others.

`protobufz`

The `protobufz` sampler provides dynamic (i.e., runtime) information about the structure of protobuf messages that are serialized and deserialized throughout the software stack running on Google's datacenter fleet. GWP randomly chooses machines to visit; when a machine is visited, the protobuf message sampler runs for several seconds and randomly selects top-level messages to be sampled. A top-level message is defined as a message on which deserialize or serialize is called directly; that is, a sub-message only appears in the data if its parent is also chosen. When a top-level message is sampled, complete information about the message and its sub-messages is captured. This includes sizes and types of all present fields, including fully qualified names for sub-message types. The `protobufz` data also includes the path of the `.proto` file in which the protobuf message is defined. This allows reconstruction of the complete hierarchy of a sampled message and joining the dynamic protobuf structure data with other data sources.

`protodb`

The `protodb` database provides static information about all `.proto` files defined in Google's codebase. This allows us to collect detailed information about each defined message type, such as the version of the protobufs language a message is defined against, whether repeated fields are packed, and the range of field numbers defined in a message.

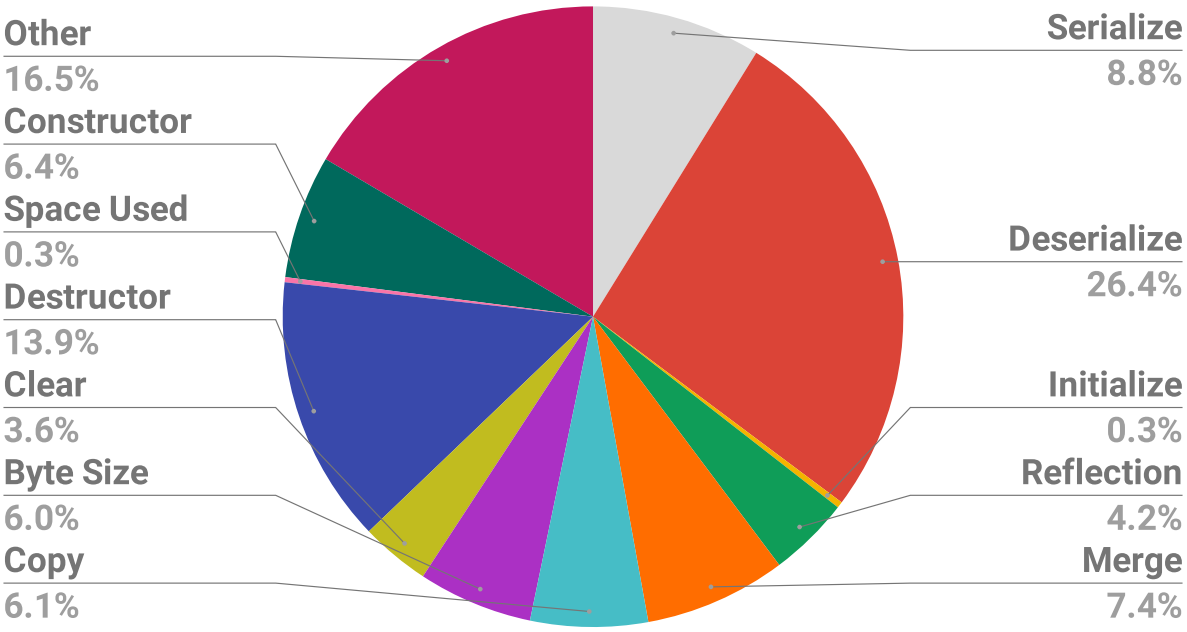


Figure 6.2: Fleet-wide C++ protobuf cycles by operation.

What is the opportunity for fleet-wide CPU cycle savings?

Using GWP CPU cycle profiles, we find that protobuf operations constitute 9.6% of fleet-wide CPU cycles in Google's infrastructure. These cycles are dominated by C++ protobuf usage: 88% of fleet-wide protobuf cycles are spent in *C++* protobufs. As a result, we will focus on C++ protobufs in the rest of this work. Chapter 6.7 discusses future support for other languages.

Figure 6.2 shows the classification of cycles spent within C++ protobufs, by operation. A few notable items are immediately visible. Firstly, deserialization alone is a significant contributor to overall CPU cycles—2.2% of fleet-wide CPU cycles are spent in C++ protobuf deserialization. Serialization cycles are also significant, with serialization in C++ consuming 1.25% of fleet-wide CPU cycles[4]. Because these are relatively coarse-grained operations, they are natural avenues to explore for acceleration opportunities. The "other" operator in Figure 6.2 represents a miscellany of glue code that is not clearly amenable to acceleration. This work focuses on the task of accelerating C++ protobuf serialization and deserialization, presenting the opportunity to accelerate/offload 3.45% of CPU cycles across Google's fleet. Chapter 6.7 discusses several other protobuf operations, which are relatively straightforward to accelerate once deserialization and serialization are handled.

[4]Virtually all calls to `Byte Size` occur during serialization, so this accounts for Serialization's 8.8% of protobuf cycles and `Byte Size`'s 6.0% of protobuf cycles in Figure 6.2.

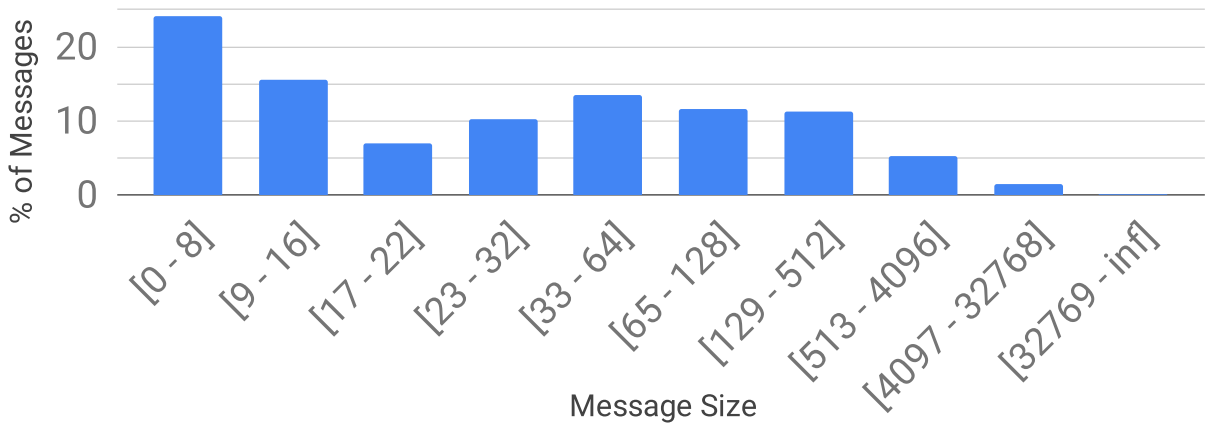


Figure 6.3: Fleet-wide top-level message size distribution.

Which proto version should we implement?

As discussed in Chapter 6.2, two versions of the Protocol Buffers language are currently supported, `proto2` and `proto3`. Although `proto3` was released in mid-2016, 96% of protobuf bytes serialized/deserialized in Google's fleet remain defined in the `proto2` language. Therefore, we target `proto2` in our accelerator design. This also suggests that usage of serialization framework APIs and formats tends to be stable over time, making hardware acceleration viable.

Should we optimize accelerator placement for the RPC stack?

To understand where to place a protobuf accelerator in the system (e.g., in-core, near-core, as a bus peripheral, CXL, PCIe, etc.), we would like to know how serializations and deserializations are initiated. One commonly assumed source of protobuf usage is the RPC stack. In Google's fleet, we find that only 16.3% of deserialization cycles are from the RPC stack and only 35.2% of serialization cycles are from the RPC stack. This challenges the common assumption that a protobuf accelerator should be placed on a PCIe-attached NIC. Instead, it is clear that other serialization and deserialization users (e.g. storage users) must be accounted for when deciding where to place a protobuf accelerator in the system.

What is the granularity of operations the accelerator needs to handle?

Another factor when deciding accelerator placement is understanding the offloading overhead that can be tolerated, which depends on offload granularity. While we do not have a mechanism to directly attribute cycle counts to individual serialization and deserialization operations, we can observe the distribution of top-level message sizes (including their sub-messages) as a proxy.

Figure 6.3 shows the distribution of message sizes observed in Google's fleet. Buckets are labeled with their inclusive byte bounds; that is, the `[0 - 8]` bucket counts the number of messages where the total encoded message size (including all sub-messages) was 0 to 8 bytes. Interestingly, the vast majority of messages are very small: 24% of messages are 8 bytes or less, 56% of messages are 32 bytes or less, and 93% of messages are 512 bytes or less. Based on this distribution, a near-core accelerator is likely necessary to efficiently handle the vast majority of messages. Also notable is that protobuf benchmarks used by prior work [170] tend to focus on only a small part (e.g., one bucket) of this distribution.

While message count is important, it is also important to keep in mind the *volume of data* in each of the message-size buckets. While we cannot directly collect this data due to infrastructure limitations, we can see that the `[32769 - inf]` bucket, which represents 0.08% of messages, contains *at least* 13.7× as many message *bytes* as the `[0 - 8]` bucket. This volume of data encoded in large messages could tolerate a higher offload overhead, while still observing a speedup. We will return to the discussion of accelerator placement trade-offs in Chapter 6.3.

What types of data movement and field encodings should the accelerator support?

In addition to the acceleration opportunities inherent in parsing or constructing protobuf message *structure* in hardware, there may be opportunities to speed up the processing of individual field values, depending on the commonly used field types in the fleet.

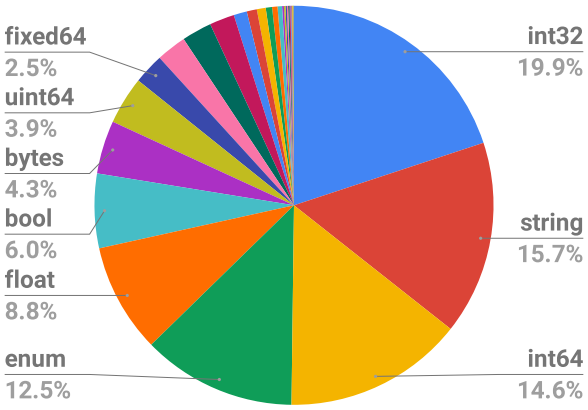
Which field types are most commonly used?

The various protobuf field types discussed in Chapter 6.2 present differing opportunities for acceleration. For example, handling an `int64` field requires encoding or decoding two varints, the key and value, which is expensive in compute-per-byte terms on a CPU. On the other hand, handling a large `bytes` field is relatively cheap as it only requires encoding or decoding two varints, the key and length, and then `memcpy`-ing a large amount of useful data.

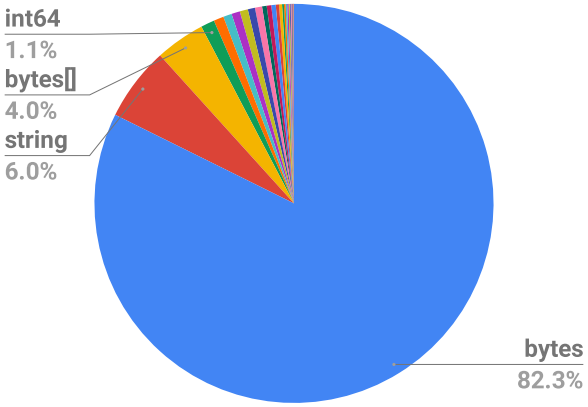
Figure 6.4a shows the proportion of observed fields of the most frequently used primitive types across Google's fleet. In this plot, sub-messages are accounted for via the primitive fields they contain but are not noted as separate fields themselves. Looking at field counts, we see very promising avenues for acceleration. Firstly, over 56% of fields are a form of varint (`int32`, `int64`, `enum`, `bool`, `uint64`), which are well-suited to acceleration. There are also a significant number of `string` and `bytes` fields, which can benefit from acceleration depending on field size.

Which field types account for the most data volume?

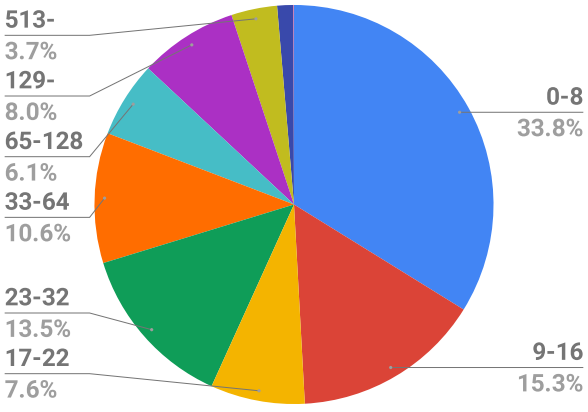
Field *counts* do not necessarily present the full picture. Ideally, we would like to know the total number of CPU cycles spent serializing and deserializing each field type. Unfortunately, the fleet-wide profiling mechanisms do not provide this level of detail. However, as a proxy, we can instead



(a) % of fields observed by type.



(b) % of message bytes observed by type.



(c) % of `bytes` fields observed by field size.

Figure 6.4: Fleet-wide field type and `bytes` field breakdowns.

obtain the number of bytes of data attributed to each field type, fleet wide. Figure 6.4b presents this data.

Startlingly, we see a very different picture when looking at the weighted (by bytes of data) field-type breakdown. `Bytes`, `string`, and repeated `bytes` and `string` fields constitute over 92% of the bytes of protobuf messages handled. If these fields tend to be very large, then the cost of handling a varint (for the field's key) is relatively small compared to the cost of performing a `memcpy` and therefore there is less opportunity for acceleration beyond `memcpy` acceleration and offloading.

How large are `bytes` fields?

To better understand the breakdown of this large amount of `bytes` and `string` data in protobuf messages, we collect data on the distribution of `bytes` field sizes, as shown in Figure 6.4c. Figure 6.4c uses the same bucket bounds as Figure 6.3; a slice labeled `0-8` in Figure 6.4c represents the percentage of `bytes` fields that were 0 to 8 bytes (inclusive) in size. Not labeled are the `4097-32768` and `32769-inf` buckets, which constitute 1.3% and 0.06% of observed fields respectively. In this view, we can see that small `bytes` fields dominate in terms of count, but data *volume* is a different story; the `32769-inf` bucket contains at least 7.2× as many *bytes* of data as the `0-8` bucket.

Which field types are responsible for the most CPU cycles in serialization and deserialization?

The data so far paints a murky picture of where opportunities for protobuf acceleration lie. To better understand how time is spent in protobuf serialization and deserialization fleet-wide, we develop a model that converts from counts and bytes of different field types into CPU cycles (or time) spent handling each type. To enable this, we first group together protobuf field types that require a similar amount of "work" to be serialized or deserialized, as shown in Table 6.1. Within the bytes-like and varint-like groups, we subdivide by field size since as discussed earlier, size can have a significant impact on serialization and deserialization performance. For varint-like fields, the fleet-wide `protobufz` histogram data provides exact labels on size bins, so we can directly determine how much data each of the varint sizes (1 to 10 bytes) contribute to the overall number of protobuf message bytes. For bytes-like fields, the profiling system collects 10 buckets with ranges shown in Figure 6.4c. To interpolate field sizes from the buckets for bytes-like fields, we select the midpoint of each bucket to represent the size of each field in the bucket, and then adjust the size of the largest bucket (32769 to infinity bytes) as necessary to obtain the total number of bytes of bytes-like fields. Altogether, this process classifies the fleet-wide bytes-of-protobuf message data into 24 slices based on pairs of `[field-type-like, size]`.

Next, for each of these 24 pairs, we construct a protobuf microbenchmark to measure serialization and deserialization performance in terms of time spent per-byte of encoded data. Combining these results with the fleet-wide bytes-per-field-type data, we obtain estimated deserialization and serialization time (or cycles) spent per-field-type across Google's fleet.

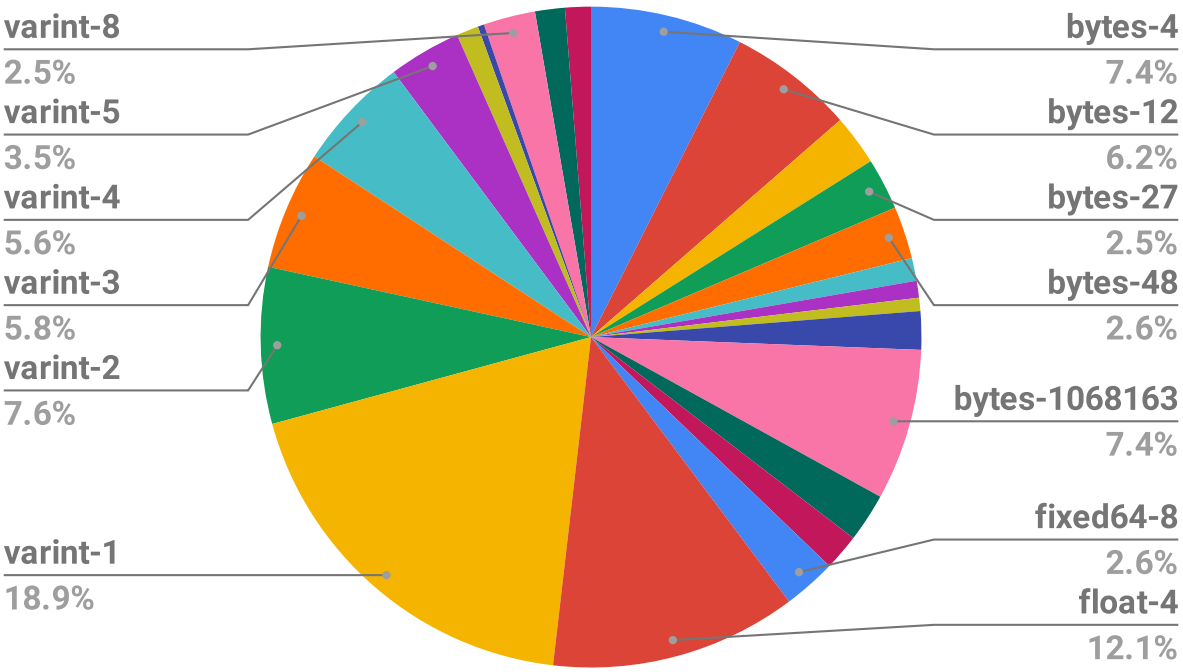


Figure 6.5: Estimated deserialization time by field type, fleet wide.

Figure 6.5 shows the estimated breakdown of deserialization time across the fleet. Several important insights can be derived from this analysis. Firstly, we notice that there is no single silver-bullet—*the accelerator will need to improve deserialization performance across the swath of field types and sizes*. Furthermore, the cases where the CPU performs best (large bytes-like fields) are a relatively small proportion of overall deserialization cycles—only 14% of time is spent deserializing protobuf data at higher than 1GB/s. While somewhat counter-intuitive, the difference in bytes-percentage between Figure 6.4b (amount of data) and Figure 6.5 (cycles) arises precisely because handling of large bytes-like fields on a CPU is so much faster per-byte than for example, a small varint-like or small bytes-like field; in our microbenchmarks, the large bytes-like field is 100-500x faster to handle per-byte. Figure 6.6 paints a similar picture for serialization. Although the largest byte bucket is relatively more significant than in the deserialization case, there is still ample opportunity in other field types. Overall, this analysis demonstrates that there are significant opportunities for acceleration in protobuf deserialization and serialization apart from fast `memcpy`.

What is the ideal accelerator programming interface?

To enable serialization frameworks to generate programming information for a serialization/deserialization accelerator, prior work [170] has suggested dynamically constructing per-*message-instance* programming tables of type/address (with implicit field presence) information for each

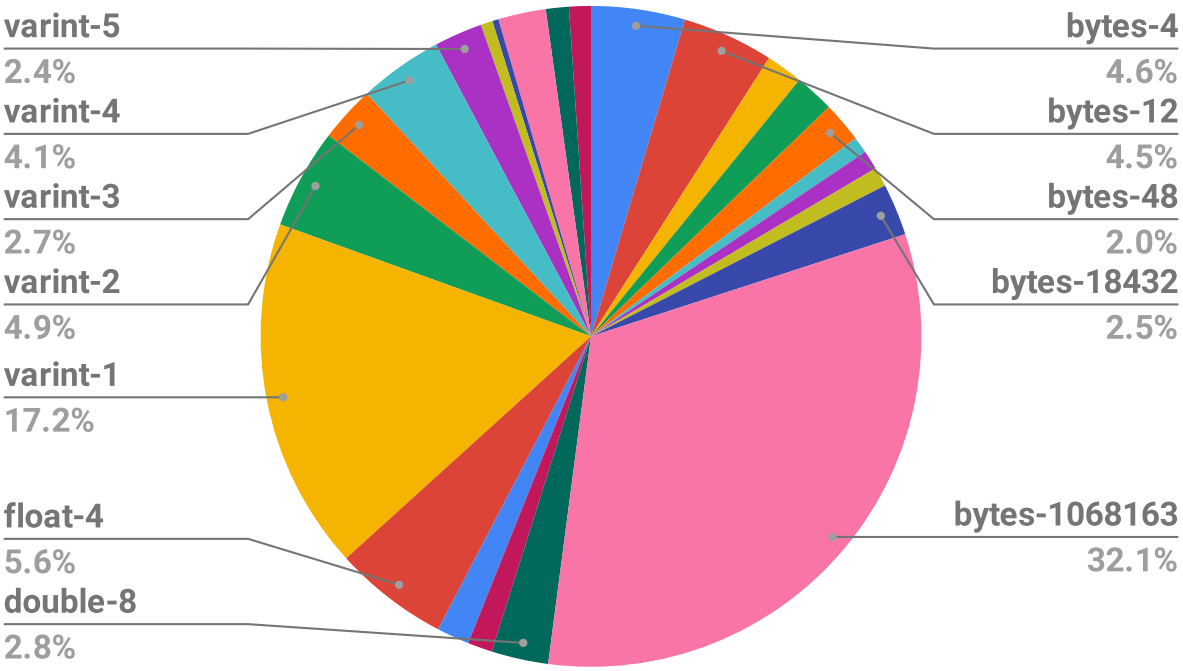


Figure 6.6: Estimated serialization time by field type, fleet wide.

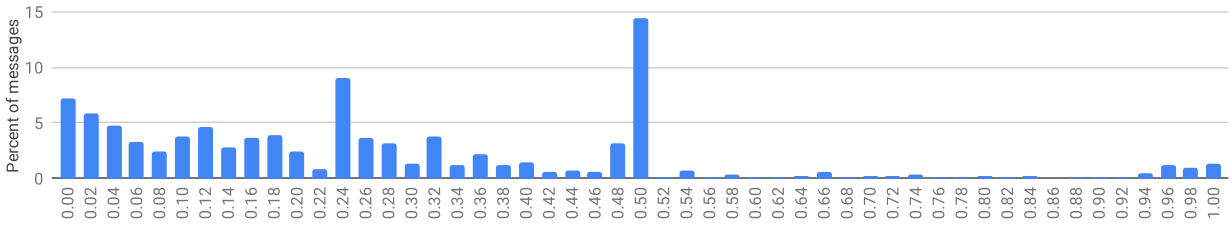


Figure 6.7: Field number usage density distribution for all message types, weighted by # of observed msgs. of each type.

populated field in a message to be serialized. While this can simplify accelerator implementation, this requires the protobuf compiler to add computationally expensive schema-management code to all generated field setters and clear methods that previously consisted of only cheap loads and stores. In contrast, our approach is to produce one Accelerator Descriptor Table (ADT) per-*message-type* (Chapter 6.4), resulting in a drastic reduction in programming table state. Our ADTs are automatically generated by the protobuf compiler and fully populated when the program is loaded, removing the need to inject costly schema-management code into all field setters and clear methods.

With our fixed, *per-message-type* ADTs, however, separate state is required to maintain field-presence information (i.e., whether or not a field has been set in a particular message object) for serialization purposes. We modify the internal per-message-instance `hasbits` bit field already generated by the protobuf compiler, to a *sparse* representation, so that the accelerator can directly index into it by field number.

More quantitatively, while prior work [170] *writes* an extra 64 bits per-present-field (a conservative assumption for the size of a schema entry) compared to our design, our design *reads* an extra bit per-field in the range of defined field numbers (due to the sparse `hasbits` representation) compared to the prior work. Thus, a *field number usage density* (= average # of present fields for a message type divided by the range of defined field numbers for that type) value of greater than $\frac{1}{64}$ (which falls in the "0.00" bucket in Figure 6.7) favors our accelerator design; Figure 6.7 shows that at least 92% of observed messages fleet-wide have a density greater than $\frac{1}{64}$, heavily favoring our accelerator design. We will build on this discussion in Chapters 6.4 and 6.4, where we discuss our accelerator programming tables and serializer frontend design.

How do we size sub-message metadata tracking structures in the accelerator?

Another important question that will arise when designing a protobuf accelerator is that of handling sub-messages. Recursing into a sub-message in hardware requires maintaining additional state per-level of hierarchy (Chapter 6.4 and Chapter 6.4), which can become expensive. Fortunately, we find that across Google's fleet, 99.9% of bytes of protobuf data handled are at depth 12 or less, with 99.999% at depth 25 or less. We also find that the maximum observed depth is less than 100. This suggests that a small amount of state can be allocated on-chip in the accelerator to handle the vast majority of message data, while trapping or spilling to DRAM is acceptable to handle less common cases.

Key insights for accelerator design

To conclude this section, we outline the key insights from our profiling study that impact the design of a protobuf accelerator:

• A hardware accelerator for protobuf serialization/deserialization could eliminate up to 3.45% of fleet-wide cycles at Google, a significant savings at scale (Chapter 6.3).

• Usage of serialization framework APIs and formats tends to be stable over time, making hardware acceleration viable (Chapter 6.3).

• A protobuf accelerator is most amenable to being placed near the CPU core. A common proposal is to place the accelerator on a PCIe-attached NIC. This is unlikely to be fruitful for several reasons:

  – Over 83% of deserialization cycles and over 64% of serialization cycles in Google's fleet are not RPC-related and offloading them over PCIe would introduce significant unnecessary data movement (Chapter 6.3).
  – Accesses into the in-memory protobuf representation performed during serialization and deserialization are ill-suited to being performed over PCIe (due to its high latency [158]). The accesses are commonly small and irregularly strided (e.g. ints, floats) or require multiple chained pointer dereferences (strings/bytes/ repeated/sub-messages). This is particularly problematic for deserialization, which must process the serialized input sequentially, field-by-field (Chapter 6.3).
  – The in-memory representation is commonly sparsely populated, so an optimization such as bulk-copying an entire in-memory protobuf object over PCIe is too wasteful. In a similar analysis as Chapter 6.3, we find that over 90% of messages fleet-wide only contain values for less than 52% of their defined fields, on average.
  – To make on-NIC acceleration truly worthwhile, a SmartNIC must also handle all encapsulations between protobuf serialization/deserialization and frame egress/ingress.

• Trying to achieve acceleration at individual field-granularity (only accelerating varint processing or `memcpy`) is unlikely to be fruitful—a protobuf accelerator will need to understand complete message structure (e.g. processing fields in parallel during serialization), handle a wide variety of field types efficiently (Chapter 6.3), *and* be able to handle fast `memcpy` (Chapter 6.3).

• To program our accelerator, we will use fixed, per-type schema tables combined with dynamic, per-instance presence-tracking bit fields. This scheme is more memory and CPU efficient than prior work [170] (Chapter 6.3).

• To handle submessages in our accelerator, we will only need to maintain on-chip sub-message context stacks of depth 25 for most messages (Chapter 6.3).

6.4 Protobuf accelerator design and internals

This section details the design and implementation of our protobuf accelerator, consisting of the deserializer and serializer units, as well as the software modifications required to exercise the accelerator within the context of our complete accelerated RISC-V SoC design.

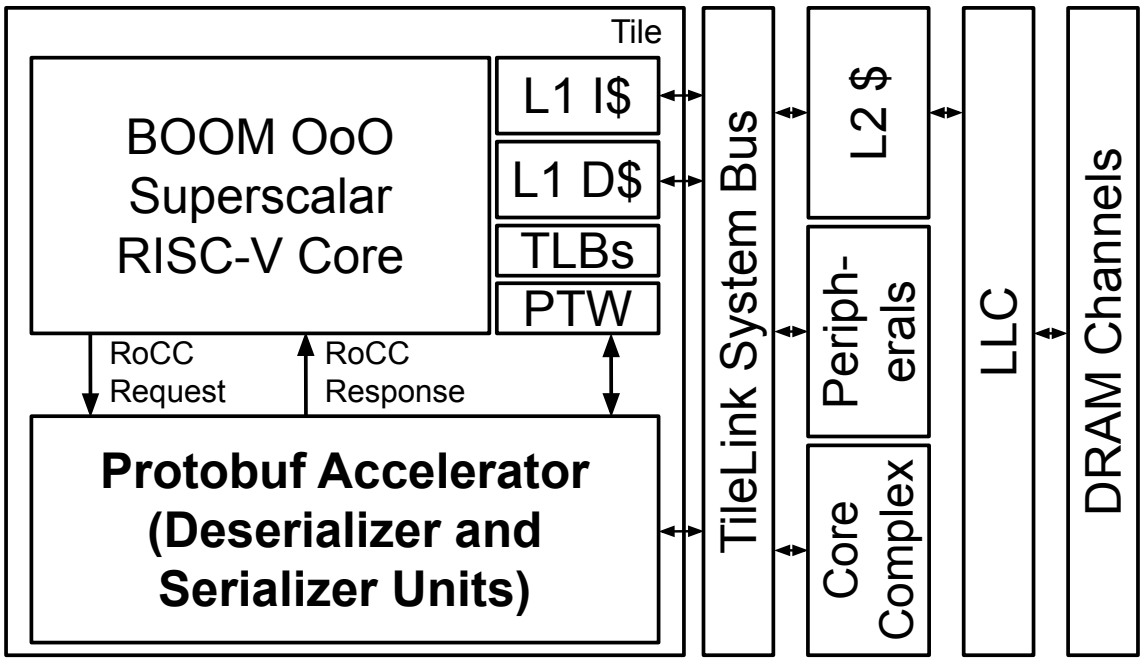


Figure 6.8: Top-level block diagram of our RISC-V SoC with an OoO superscalar core and protobuf accelerator.

System overview

The protobuf accelerator is implemented in Chisel RTL [25] and incorporated into the Chipyard RISC-V SoC generator ecosystem [14]. Figure 6.8 shows the overall architecture of the accelerated SoC. We configure the SoC to use BOOM, an OoO superscalar RISC-V core with performance comparable to ARM A72-like cores [230].

The accelerator receives commands directly from the BOOM application core in the SoC via the RoCC interface [22, 40], which allows the CPU to directly dispatch custom RISC-V instructions in its instruction stream to the accelerator with low latency (ones-of-cycles). These *RoCC instructions* [22] can supply two 64-bit register values from the core to the accelerator. The accelerator accesses the same unified main memory space as the CPU using the coherent 128 bit-wide TileLink system bus [97]. Accesses to main memory made by accelerator components go through the *memory interface wrappers* shown in Figures 6.9 and 6.10. These maintain TLBs and interact with the page-table walker (PTW) to perform translation and thus allow the accelerator to use virtual addresses. These also manage tracking OoO responses from the system bus and support a configurable number of outstanding requests, depending on memory system characteristics and resource constraints. Lastly, as shown in Figure 6.8, all memory accesses made by the accelerator go through the L2 and LLC, which are shared with the application core. Putting these pieces together, offload overhead is minimal: apart from the custom instructions that perform a serialization or deserialization, only a `fence` instruction is required between the user program operating on a

protobuf and the accelerator operating on a protobuf.

Software changes to the protobuf library

We modify the `protoc` compiler to automatically generate *Accelerator Descriptor Tables (ADTs)*, which encode the layout of a protobuf *message type* in application memory and information about its fields. There is one ADT per-*message-type*, rather than per-message-*instance*, and ADTs are populated when the program is loaded, avoiding adding code to the critical path of setting or clearing message fields in user code. When the serialization or deserialization of a message is dispatched to the accelerator, the message's type's ADT is also passed to the accelerator.

Each ADT contains three regions. The 64B header region contains layout information at the message-level, consisting of: (1) a pointer to a default instance (or `vptr` value) of the message type, (2) the size of C++ objects of the message type, (3) an offset into message objects for an array of field-presence bit fields (`hasbits`), and (4) the min and max field number defined in the message. The second ADT region consists of 128-bit wide entries that represent each field in the message type, indexed by field number. Each entry consists of the following details for a field: (1) the field's C++ type and whether the field is repeated, (2) the offset where the field begins in the in-memory C++ representation of the message, and (3) for sub-message fields, a pointer to the sub-message type's ADT. The final ADT region is the `is_submessage` bit field, an array of bits that indicates if a field is a sub-message. This is used to reduce complexity in the serializer, since it can know when it needs to switch contexts into a sub-message without waiting for a full ADT entry read.

In addition to ADT information, the serialization unit in the accelerator must also know which fields in a given C++ protobuf message are actually populated. The protobuf library tracks this information using the private `hasbits` member of each C++ protobuf message object. Each bit in the `hasbits` bit field represents the "presence" of a particular field. `protoc` represents `hasbits` *densely*, but supporting a dense packing in the accelerator would require significant overhead (e.g. a mapping table indexed by field number, introducing an additional 32-bit read per-field). Based on our profiling insights in Chapter 6.3, we find that the dense packing optimization is not significantly helpful in the common cases seen at scale. Thus, to improve accelerator efficiency, we make a different hardware/software co-design trade-off for the accelerator context; we modify the representation of the `hasbits` bit field such that the accelerator can directly index into it, based on field number. To save memory in the common case where field numbers are contiguous but start at a large number, we provide the accelerator with the minimum defined field number in a message type, with respect to which it calculates field-number offsets.

Accelerator memory management

To remove the CPU from the critical path of serialization and deserialization, the accelerator will need to manage a memory region in which it allocates and populates deserialized `C++` message objects and serialized message outputs. Similar to how an arena is constructed in advance when using arena allocation for software-only protobuf processing (Chapter 6.2), the application program pre-

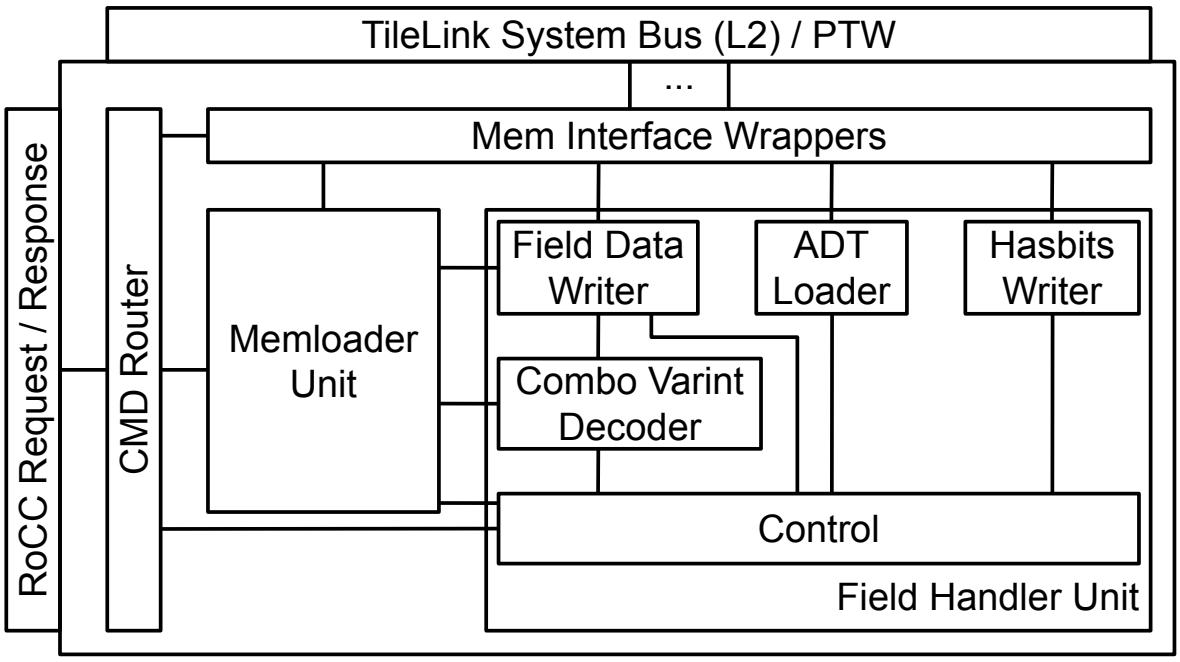


Figure 6.9: Deserializer unit top-level block diagram.

allocates arena memory regions for the accelerator and passes their pointers to the accelerator via two custom RoCC setup instructions ({`ser`,`deser`}`_assign_arena`). In the rest of this chapter, we will refer to standard upstream protobuf arenas (i.e., those from Chapter 6.2) as *software arenas* and arenas given to the accelerator as *accelerator arenas*.

Deserializer unit

The deserializer unit is responsible for receiving a serialized protobuf (as a pointer to a sequence of bytes) and decoding it to populate a corresponding C++ object of that message's type. Figure 6.9 shows the block-level design of the deserializer unit.

To maintain compatibility with standard protobuf software APIs, we expect that the top-level C++ protobuf message object is allocated by the user code (e.g. in the software arena). Any internal objects (sub-messages, strings, and repeated fields) are allocated by the accelerator in the accelerator arena.

Dispatching a deserialization from the CPU

To begin deserialization of a message, the CPU issues two custom instructions through the RoCC interface. The first instruction, `deser_info`, supplies a pointer to the ADT of the message type being deserialized and a pointer to the top-level destination message object for the accelerator to populate. The second instruction, `do_proto_deser`, supplies a pointer to the serialized input

buffer, the smallest defined field number in the message type, and the length of the input buffer, and kicks off deserialization in the accelerator. Once these instructions are issued, the CPU can perform other work, issue more `deser_info` and `do_proto_deser` pairs, or issue a `block_for_deser_completion` instruction, which is committed after all in-flight deserializations are completed. This is a flexible middle ground that allows for batching deserializations, without requiring SW to unnecessarily poll for completion.

Memloader unit

Once a `do_proto_deser` instruction is dispatched to the accelerator, the accelerator begins loading serialized buffer contents from memory using the memloader unit. The memloader exposes a decoupled streaming interface to the rest of the pipeline that allows the consumer to accept a consumer-dictated amount of data per-cycle, up to 16B. A full 16 bytes of buffered data are always exposed on this interface, since the number of bytes the consumer will wish to consume is data-dependent.

Field-handler unit

The field-handler unit implements the core parsing logic required to convert the serialized buffer contents into an in-memory C++ object for the user program to consume. The field handler control is implemented as a state machine that, in a loop, parses a field's key (the `parseKey` state), blocks for detailed type information from the ADT entry for the field (the `typeInfo` state), and then moves into a set of states that handle parsing and writing the field's value based on its detailed type information.

Field-handler unit: `parseKey` state

Each key is encoded as a varint, which can be up to 10B wide. The field-handler unit contains a combinational varint decoder, which can directly peek at the next 10B of the serialized buffer via the memloader's variable-width consumer interface. The varint parser emits the decoded key (as a 64-bit-wide `uint`) and the encoded length N, so the memloader can discard the N-byte key at the end of the cycle. As described in Chapter 6.2, the key consists of two components, the field type and the field number. At the end of the `parseKey` cycle, the field handler dispatches a request to the ADT loader containing the ADT base address for this message type and the field number of the field. The field handler also dispatches a request to the `hasbits` writer, which will set the appropriate bit in the C++ object's `hasbits` bit field to indicate that the field is present in the message.

Field-handler unit: `typeInfo` state

After the `parseKey` state, the accelerator moves to the `typeInfo` state. This state serves to block on the response from the ADT loader in order to obtain detailed type information. Once the response is received, the logic in this state dispatches to one of four state classes: final write states

for scalar fields, string allocation and copy states, repeated-field handling states, or sub-message handling states.

Field-handler unit: final write states for scalar fields

This set of states handles writes for scalar field types: the varint, 64-bit, and 32-bit protobuf wire types. At the end of this stage, the decoded field data is written into memory. The write address is available from the ADT entry previously received in the `typeInfo` state. The decoded value and size depend on the detailed type being handled, which is known from the loaded ADT entry.

To handle the varint wire type, the same combinational varint parser from the `parseKey` state generates a fixed-width value and supplies the number of bytes consumed back to the memloader consumer interface. The ADT entry distinguishes whether the output type is 32-bits or 64-bits wide and signed or unsigned. For signed varints, the decoded value is passed through an additional combinational zig-zag [63] decoding unit.

Field-handler unit: string allocation and copy states

String and byte fields and the other field types we will discuss in the remainder of this section introduce a new wrinkle into the deserialization process—instead of relying on user code to have allocated destination memory, the accelerator must handle memory allocation in the accelerator arena assigned to it by the user program.

Our accelerator constructs string objects that are compatible with modern versions of `libstdc++`, which allows user code to directly operate on strings in the deserialized protobuf message as if it were deserialized by the software protobuf library. The accelerator first decodes and consumes the varint-encoded string length. It then constructs the string object and depending on the length, a separate array for the string contents (i.e. the common small string optimization). A pointer to the newly allocated string object is written into the offset in the C++ message object that is obtained from reading the field's ADT entry. Next, the accelerator consumes the string contents from the memloader and writes them into the allocated buffer in memory.

Field-handler unit: repeated-field handling states

Our accelerator also handles packed and unpacked repeated fields. Packed repeated fields are handled in a similar vein as strings, since they are also represented as length-delimited values. Unpacked values are handled by creating a tagged open-allocation region when the first element in an unpacked repeated field is seen. As more key-value pairs with the same tag are received in the serialized representation, they are copied into the open allocation region. When the accelerator encounters either the end of the current message or a different unpacked repeated field, it closes-out the open allocation region and writes out a final length in *elements* into the repeated-field object in application memory.

Field-handler unit: sub-message handling states

As described in Chapter 6.2, protobuf messages can contain sub-messages. So far, the accelerator has relied on several pieces of information that are supplied by the CPU via RoCC instructions to perform deserialization: the ADT pointer for the top-level message's type, a pointer to the user-allocated C++ object in which the deserialized top-level message should be written, the smallest defined field number in the message type, and the length of the serialized top-level message input in bytes. Going forward, we will refer to these elements as *message-level metadata*.

The deserialization process for sub-messages requires consuming the serialized sub-message content in a depth-first manner, which means we must preserve message-level metadata for each message on the path between the current sub-message and the top-level message. Given the depth-first parsing order, we maintain a hardware stack to track message-level metadata during deserialization. The accelerator always uses the message-level metadata at the top of the metadata stack, allowing reuse of the entire pipeline for sub-message decoding.

Putting these pieces together, the sub-message parsing state prepares the accelerator to consume the serialized sub-message output by modifying the stack entries and by performing memory allocation. In this state, the accelerator first decodes the serialized sub-message field's header, which contains the varint-encoded length of the serialized message in bytes. As with other fields, the ADT entry for the field has already been fetched and contains a pointer to the ADT of the *sub-message's* type. Using this pointer, the accelerator fetches metadata from the aforementioned header region of the ADT for the sub-message type, which contains a pointer to a default instance (or `vptr`) of the type and the size of the type. Given this information, the accelerator allocates and initializes a new C++ object for the deserialized sub-message data and writes a pointer to the newly allocated object into the parent object's field pointer. Finally, the accelerator pushes new entries onto the message-level metadata stacks. When the setup is completed, the accelerator returns to the `parseKey` state, where it begins parsing and populating the *sub-message*.

As the sub-message is being processed and input data is consumed, the accelerator updates the total consumed serialized input length. When this length is equal to the top entry in the stack of the serialized message lengths, the sub-message parsing is completed. Popping an entry from each stack returns the accelerator to parsing the parent message.

Serializer unit

The protobuf accelerator's serializer unit converts a C++ protobuf object populated by a user application into a serialized sequence of bytes. Figure 6.10 shows the block-level design of the serializer unit.

Field serialization order and serializer memory management

One counter-intuitive but critical note about field serialization order is that the accelerator iterates through fields in *reverse* field number order and writes the serialized output from high-to-low addresses. This produces byte-wise identical output as a software serializer serializing in increas-

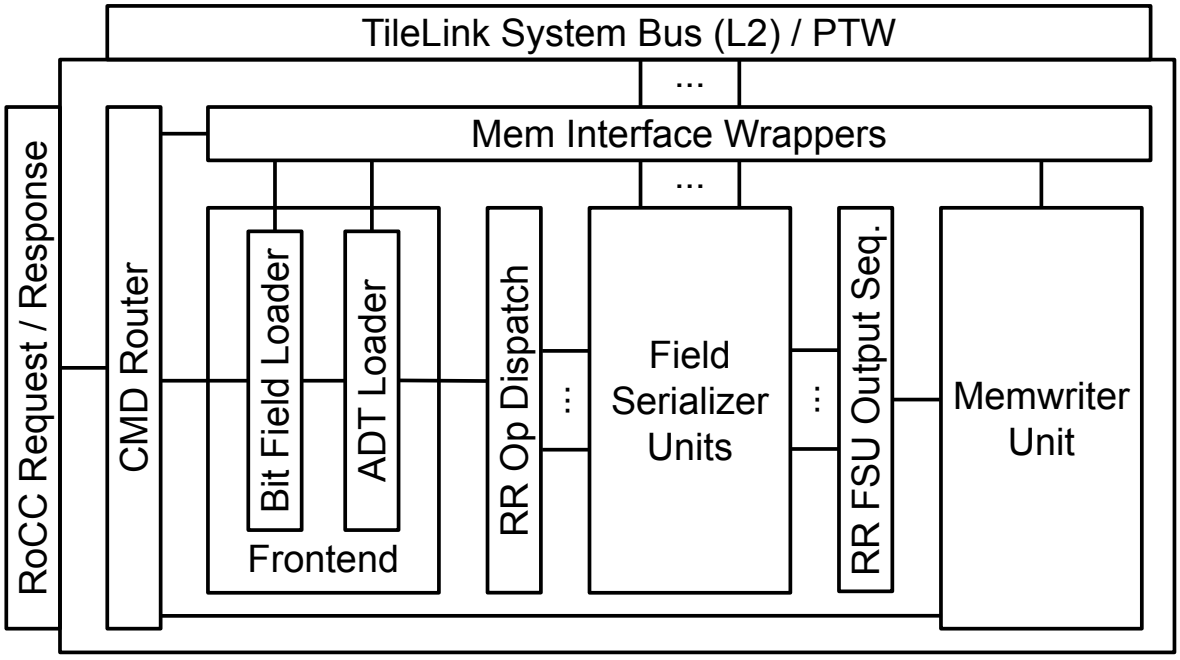


Figure 6.10: Serializer unit top-level block diagram.

ing field number order and writing output from low-to-high addresses, but drastically simplifies the process of populating the length of sub-messages (which appear before the fields in a sub-message). The accelerator arena internally contains two memory regions for serialization: (1) a buffer in which to allocate and write serialized output data and (2) a buffer in which to store pointers to the start of each serialized output in (1).

Dispatching a serialization from the CPU

To dispatch a serialization operation, like before, the user program issues two RoCC instructions. The `ser_info` instruction supplies the offset of the `hasbits` field in the C++ protobuf message object to serialize and the largest and smallest *defined* field numbers for the message type. The `do_proto_ser` instruction supplies a pointer to the top-level ADT of the protobuf message to serialize and a pointer to the C++ representation of the protobuf message to serialize and kicks off a serialization. Like deserialization, the CPU can perform other work, issue more `ser_info` and `do_proto_deser` pairs, or issue a `block_for_ser_completion` instruction, which is committed after all in-flight serializations are completed. After completion, the user program can call a function to get a pointer to the *N*th serialized output (and its length) from the arena.

Frontend

When the accelerator receives the RoCC instructions to initiate a serialization, the accelerator frontend uses the supplied register values to initialize a set of stacks (for sub-message support) that maintain context information for the message being serialized.

Next, the accelerator frontend loads the `is_submessage` and `hasbits` bit fields from memory in parallel, iterates through the fields bit-by-bit, and issues an ADT load request whenever a field is present. For non-sub-message fields, the frontend simply loads ADT information and issues a handle-field-op to the remainder of the pipeline. If a present field is a sub-message, the frontend first updates the current message's context information in the stack. Then, the frontend loads the ADT entry for the sub-message field and the sub-message pointer itself. This information is then pushed onto the context stacks. The handle-field-ops issued to the rest of the pipeline contain a depth field, which allows the memwriter unit to determine when a new sub-message has started. Once these housekeeping steps are completed, the frontend then resumes regular field handling as described previously. After the frontend handles the message's smallest defined field number, it issues a special handle-field-op with field number zero (which the protobuf specification prevents from being used for a user-defined field) to indicate to the remainder of the pipeline that the (sub-)message has been completed. When the end of a (sub-)message is reached, the frontend pops from the context stacks and continues with the parent message (or signals top-level message completion).

Field serializer units

Next, the individual handle-field-ops from the frontend are dispatched round-robin to a set of field serializer units, which produce serialized key, value pairs for individual fields. They load the C++ representation of the field data to serialize from memory, encode it if necessary (e.g. encoding integers as varints), and then make the serialized field data available to their output ports in chunks of parameterizable width. The field serializers also construct and emit the key for each non-sub-message field that is part of the serialized output. Due to space constraints, we do not detail how each individual field type is handled. However, the process of serializing values of each field type is effectively the reverse of deserializing a field of the corresponding type (without needing to perform allocation and C++ object construction), which is discussed in depth in Chapters 6.4 to 6.4.

Memwriter unit

The next stage of the pipeline consumes serialized field data from the parallel set of field serializer units in round-robin fashion and sequences the output into one output stream to feed to the memwriter unit, which writes data to memory. The memwriter also handles the aforementioned special handle-field-ops that indicate the beginning and end of messages and sub-messages. The memwriter maintains a stack of the lengths of the messages currently being handled and pushes and pops from the stack as the handle-field-ops with a new depth or with field number zero are received. When an op with field number zero is received (which signals end-of-message), the

memwriter injects the sub-message's key, which includes the sub-message's serialized length. The need to inject this key affirms why the output buffer is populated from high-to-low address—we must see all serialized sub-message fields before we know the length of the entire serialized sub-message. When an end-of-message op is received for a top-level-message, the memwriter also writes the current output pointer (the address of the front of the completed serialized message) into the next slot in the buffer of serialized message pointers.

6.5 Evaluation

We evaluate our complete accelerated system implemented in RTL using two sets of benchmarks: (1) microbenchmarks that exercise a variety of protobuf features/types and (2) HyperProtoBench, a benchmark suite representative of key serialization framework users at scale. To enable running these benchmarks directly on our RTL design, we run FPGA-accelerated simulations of the design using FireSim [118], which provides high-performance, deterministic, and cycle-exact[5] modeling of the design, while cycle-accurately modeling I/O, including DRAM [34].

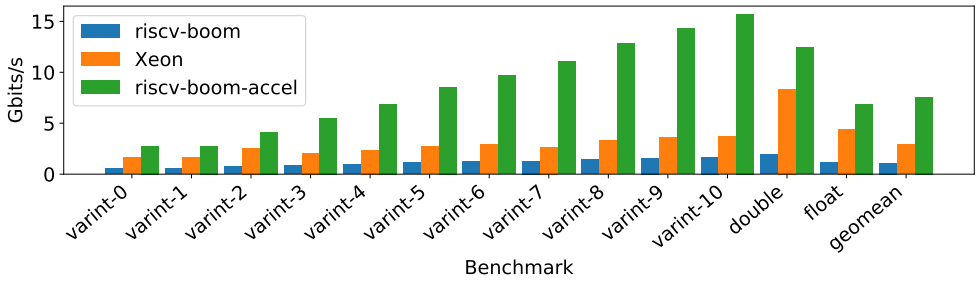
For comparison, each benchmark is run on three systems: the baseline single-core BOOM-based[6] RISC-V system modeled at 2 GHz core frequency ("riscv-boom"), the same RISC-V system with our accelerator attached ("riscv-boom-accel"), also modeled at 2 GHz core and accelerator frequency (based on the critical path results in Chapter 6.5), and one core (2 HT) of a Xeon E5-2686 v4-based server ("Xeon"), running at 2.3 GHz base/2.7 GHz turbo.

Microbenchmarks

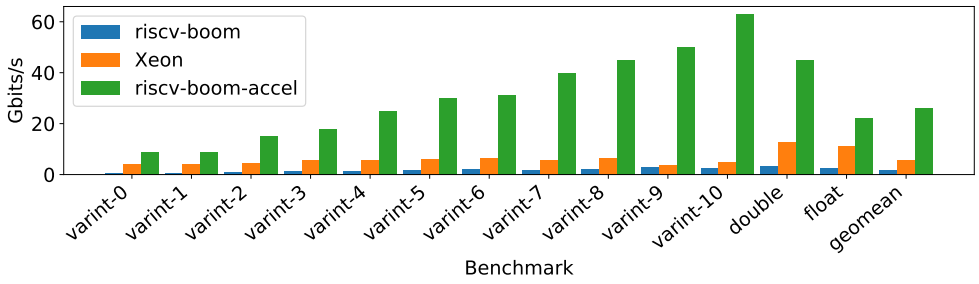
To understand accelerator performance on the various field types supported by protobufs, we developed a set of microbenchmarks that test the performance-similar field types shown in Table 6.1. We also created μbenchmarks to evaluate performance on messages containing sub-messages and repeated fields. Where appropriate (e.g. varints and strings), we also break-down benchmarks by field size. Each μbenchmark tests either serialization or deserialization of messages containing a fixed number of fields of a particular protobuf field type. For varints, doubles, floats, and their repeated equivalents, we set this to five fields per message, so that the middle-sized non-repeated varint's μbenchmark message falls roughly at the median of message sizes shown in Figure 6.3. All other μbenchmarks use one field per-message. Each benchmark performs a timed batch of deserializations and serializations, operating on a pre-populated set of serialized messages or C++ message objects respectively, and reports throughput by dividing the total amount of *serialized* message data consumed/produced by the time to process the batch.

[5]All components of the RISC-V SoC written in RTL, including our accel. design, are modeled bit-by-bit and cycle-by-cycle exactly as they would perform in silicon taped-out using the same RTL.

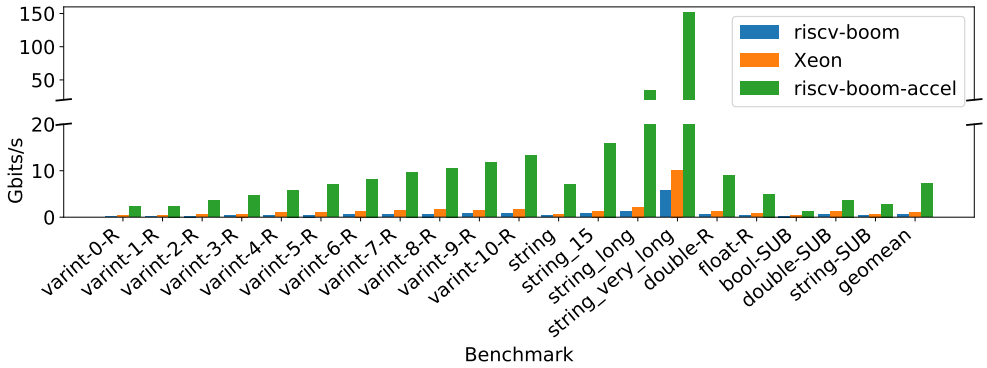
[6]In particular, we use a high-end configuration of SonicBOOM, which performs comparably on IPC with ARM Cortex A72-like cores when running SPEC2017 and achieves higher CoreMarks/MHz than A72-like cores running CoreMark [230].



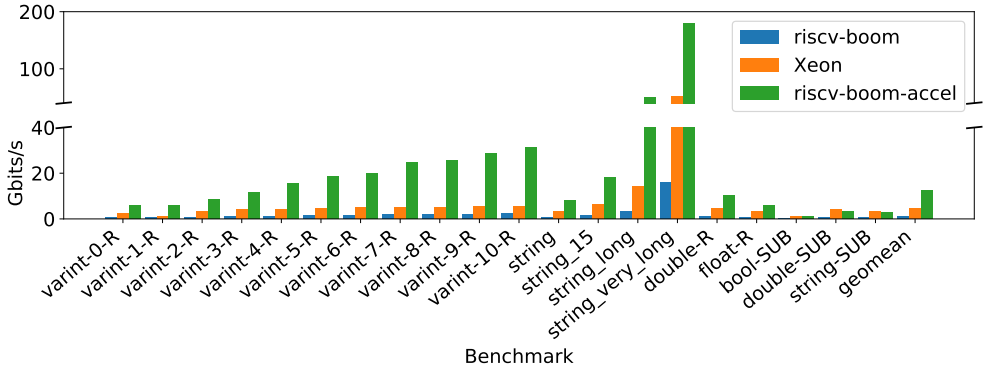
(a) Deserialization, field types that do not require in-accel. memory allocation.



(b) Serialization, field types "inline" in top-level C++ message objects.



(c) Deserialization, field types that require in-accel. memory allocation.



(d) Serialization, field types not "inline" in top-level C++ message objects.

Figure 6.11: Protobuf microbenchmark results.

Deserialization

Figure 6.11a shows the results of running deserialization μbenchmarks for field types that do not require memory allocation in the accelerator. To some degree, all examined systems exhibit the behavior that deserialization throughput of varints increases with the size of the varint field. This is due to a variety of factors, including underutilization of memory bandwidth with small loads, fixed-overhead of handling a field (e.g. key handling), and in the case of the accelerator, single-cycle decoding of all varints. Summarizing these results, we find that our accelerated system performs on average 7.0× faster than the BOOM-based system and 2.6× faster than the Xeon.

Figure 6.11c shows the results of running deserialization microbenchmarks for field types that require the accelerator to perform memory allocation, including repeated fields, strings, and sub-messages. In this figure we also see performance improvements across the board. A key reason for improved performance in these benchmarks is the accelerator's ability to directly allocate memory without requiring CPU intervention. Also, as mentioned in Chapter 6.3, the long-string deserialization case essentially becomes a `memcpy`, which the accelerator handles well. Summarizing these results, we find that the accelerated system performs on average 14.2× faster than the BOOM-based system and 6.9× faster than the Xeon-based system.

Serialization

Figure 6.11b shows the results of running serialization μbenchmarks for field types that are "inline" in C++ message objects. In practice, this is the exact distinction between non-allocated and allocated field types discussed in the deserializer results, however we do not re-use this terminology for clarity. While other platforms show a less consistent increase in throughput based on varint size, the accelerated system shows increased performance as varint size increases. This is similarly due to the improved bandwidth utilization due to larger loads as well as the accelerator's ability to encode fixed-width C++ integer formats into a varint in a single-cycle. We also note that due to this fact, floats and doubles perform similarly to equivalently sized varint fields. Summarizing these results, we find that the accelerated system performs on average 15.5× faster than the BOOM-based system and 4.5× faster than the Xeon.

Figure 6.11d shows the results of running serialization μbenchmarks for field types that are not "inline" in the top-level C++ message object. Similarly to deserialization, one notable result is the very-long and long sizes of string fields, which both essentially become `memcpy` operations. The accelerator again performs well here, but it is interesting to note that the Xeon also performs extremely well on the very-long-string benchmark, notably better than the deserialization case. Summarizing these results, we find that the accelerated system performs on average 10.1× faster than the BOOM-based system and 2.8× faster than the Xeon.

Overall microbenchmark results

To get a sense of the overall performance improvement our accelerator achieves across a variety of field types, we take the geometric mean of the results reported for the four classes of μbenchmark

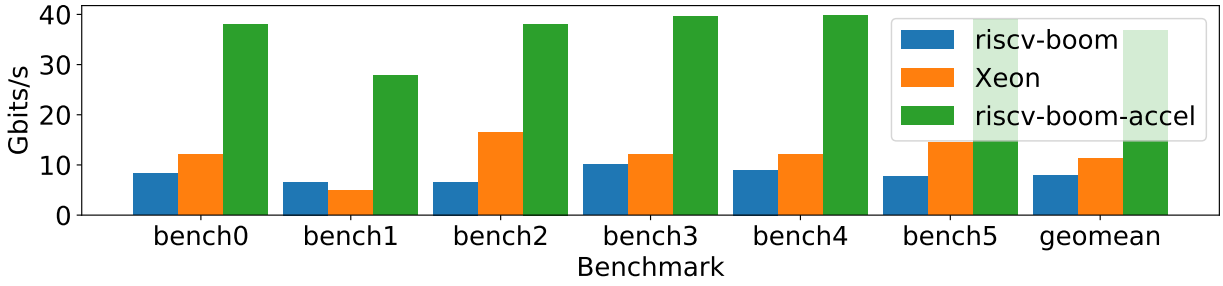


Figure 6.12: HyperProtoBench deserialization results.

shown above, for each of the two hosts we compare against. We find that on average, the accelerated system performs 11.2× better than the BOOM-based system and 3.8× better than the Xeon-based system.

HyperProtoBench: Open-source Google fleet-representative protobuf benchmarks

To gain a better understanding of how our design behaves at scale and to enable more productive research in serialization frameworks by providing insight on how these frameworks are used in a hyperscale context, we have open-sourced *HyperProtoBench*, a collection of benchmarks that represent a significant portion of fleet-wide protobuf deserialization and serialization cycles at Google.

To construct these benchmarks, we collect samples from Google's live production fleet that describe the "shape" of protobuf messages used, per service, using the same mechanisms as described in Chapter 6.3. This shape data includes information about which messages are being serialized/deserialized, which fields are set in those messages, the sizes and types of those fields, and the message hierarchy. Given this input, an internal synthetic protobuf benchmark generator fits a distribution to the input data and then samples from it to produce a benchmark that is representative of a selected production service. For each service, the generator produces a `.proto` file with message definitions representative of those used in the production service and generates a C++ benchmark that constructs, mutates, and serializes/deserializes the protobuf messages appropriately.

To cover as many of the total fleet-wide protobuf serialization and deserialization cycles as possible, we use fleet-wide profiling data to determine the five heaviest users of protobuf deserialization and the five heaviest users of protobuf serialization. In aggregate, these services cover over 13% of fleet-wide deserialization cycles and 18% of fleet-wide serialization cycles. For each of these services, we construct a synthetic benchmark representative of its protobuf usage. This collection of benchmarks comprises *HyperProtoBench*.

Figures 6.12 and 6.13 show the results of running the HyperProtoBench deserialization and se-

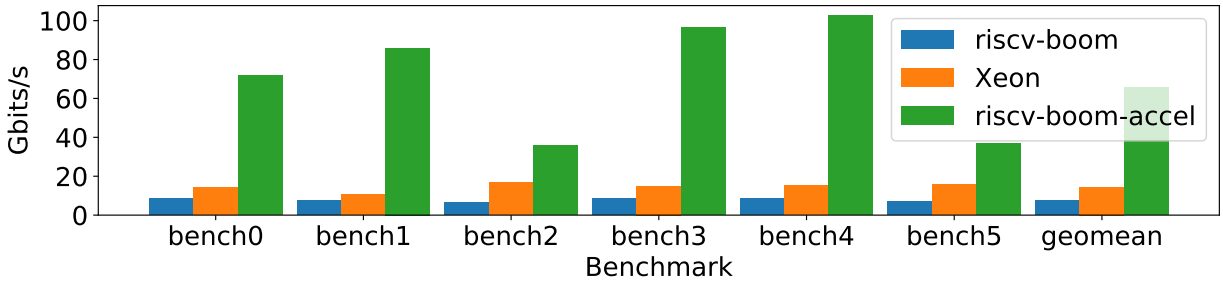


Figure 6.13: HyperProtoBench serialization results.

rialization benchmarks respectively, on the same collection of three systems ("riscv-boom", "riscv-boom-accel", and "Xeon"). We find that our accelerated system achieves on average 6.2× performance improvement compared to our baseline RISC-V SoC with OoO (ARM A72-like) cores and 3.8× performance improvement compared to the Xeon-based system. Extrapolating from the fleet-wide cycles spent in serialization and deserialization, this would result in a savings of over 2.5% of fleet-wide cycles, which at scale translates to hundreds of millions of dollars in savings, across the industry [24, 198].

ASIC critical path and area

To estimate the ASIC critical path and area results for our accelerator design, we run the design through synthesis for a commercial 16nm-class process. The deserializer achieves a frequency of 1.95 GHz with a silicon area of 0.133 mm^2. The serializer achieves a frequency of 1.84 GHz with a silicon area of 0.278 mm^2.

6.6 Related work

Optimus Prime [170] presents an accelerator for serialization/deserialization. Their design requires adding code to all protobuf setters and clear methods to construct/manage their per-*message-instance* schema tables for accelerator programming, which introduces significant memory/compute overhead. As discussed in-depth in Chapters 6.3, 6.4, and 6.4, our work instead uses per-*message-type* ADTs (created once at program load-time) for accelerator programming and uses the existing per-*message-instance* `hasbits` bit field in protobufs to track field presence, avoiding the overheads introduced by Optimus Prime. Further in contrast to our work, Optimus Prime focuses on the serialization process and does not cover the deserialization process in-depth, especially the complexity of managing memory and allocating/constructing C++ objects. Also, our work produces an open-source RTL design which is used as the single source of truth for all evaluation purposes; the RTL design is simulated at high performance using FireSim to gather benchmark performance data and evaluated for area/critical path. Finally, Optimus Prime uses three

microbenchmarks for protobufs, part of the DeathStarBench benchmark [75] for Apache Thrift, and compares against ARM A57 cores while our work runs protobuf benchmarks derived from key Google services and compares against an (ARM A72-like) OoO RISC-V core and a Xeon server. As discussed earlier, the real-world data suggest several non-intuitive design trade-offs.

Cereal [105] presents an accelerator for serialization/deserialization of Java objects. Cereal requires modifications to the JVM and uses a custom wire format that is amenable to hardware acceleration. In contrast, our work maintains compatibility with the existing protobuf wire format and does not require modifications to the language implementation. Additionally, directly serializing language objects is not practical in a production-WSC running many services, since backwards compatibility becomes challenging. For example, fields are commonly added or removed from a message over time, which would alter object layout, requiring services to update in lock-step. The schema and compiler-based design of protobufs (Chapter 6.2) prevents these issues.

Two recent proposals, *Zerializer* [221] and *Breakfast of Champions* [177], suggest adding serialization and deserialization support to PCIe-attached NICs. The former suggests adding (but does not implement) a custom hardware accelerator, while the latter implements a proof-of-concept that re-uses existing NIC scatter-gather functionality to handle serialization and deserialization, but requires a custom zero-copy-friendly serialization API/format. While we place our accelerator near the CPU, it could easily be placed on a PCIe-attached NIC. We discuss placement trade-offs in Chapter 6.3.

HGum [227] and *Fletcher* [166] generate serialization/deserialization hardware for FPGA-CPU/FPGA communication. Unlike our work, HGum implements a custom serialization format while Fletcher generates hardware pipelines specific to a message schema that must be specified when hardware is constructed.

6.7 Discussion and future work

Instruction cache and branch predictor benefits. Reduced I$ pressure and reduced pressure on branch-prediction resources are often overlooked as benefits of protobuf offloading. `protoc` generates large amounts of branch-heavy code to handle serializations and deserializations in software. In some cases, a call to serialize or deserialize can even effectively act like an I$ and branch predictor flush. Offloading serialization and deserialization to an accelerator eliminates both of these pressures. This can save significant CPU cycles, potentially as many as accelerating protobufs itself.

Accelerating other protobuf operations. Figure 6.2 shows several other protobuf operations that consume a non-trivial number of CPU cycles, including merge, copy, clear, constructors, and destructors. Re-using the hardware building blocks from serialization and deserialization and adding new custom instructions for each, a future version of our accelerator would be able to handle merge, copy, and clear, addressing another 17.1% of fleet-wide C++ protobuf cycles. While we did not claim constructors (6.4% of fleet-wide protobuf cycles) as part of the fleet-wide acceleration opportunity for our accelerator, the accelerator *does* address some of these cycles, by constructing sub-message objects during deserialization. A small change to the protobuf API (software accept-

ing a top-level message pointer from the accelerator) would allow the accelerator to fully offload all deserialization-related constructor cycles. Destructor cost (13.9% of protobuf cycles) can be addressed in software by fully migrating to arenas, which the accelerator already supports.

Future support for `proto3` and non-C++ host languages. To our knowledge, the only change needed for `proto3` support in our accelerator is adding support for UTF-8 validation of string fields during deserialization. Adding support for other host languages would require the accelerator to understand the layout of and construct in-memory protobuf message objects for new languages and their standard library components, like strings.

6.8 Conclusion

This work presented an end-to-end study of profiling and accelerating serialization and deserialization, two key datacenter tax components. To understand the trade-offs and opportunities in hardware acceleration for serialization frameworks, we presented the first in-depth study of serialization framework usage at scale by characterizing Protocol Buffers usage across Google's WSC fleet and used this data to construct HyperProtoBench, an open-source benchmark representative of key serialization-framework user services at scale. In doing so, we identified key insights that challenge prevailing assumptions about serialization framework usage.

We used these insights to develop a novel hardware-accelerator for protobufs, implemented in RTL and integrated into a RISC-V SoC. We have fully open-sourced our RTL, which, to the best of our knowledge, is the only such implementation currently available to the community.

We also presented a first-of-its-kind, end-to-end evaluation of our entire RTL-based system running hyperscale-derived benchmarks and microbenchmarks. We booted Linux on the system using FireSim to run these benchmarks and pushed the design through a commercial 16nm-class process to obtain area and frequency metrics. We demonstrated an *average* 6.2× to 11.2× performance improvement (sometimes up to 15.5×) vs. our baseline RISC-V SoC with BOOM OoO (ARM A72-like) cores and despite the RISC-V SoC's weaker uncore/supporting components, an *average* 3.8× improvement (sometimes up to 6.9×) vs. a Xeon-based server.

In addition to advancing the state of the art in serialization framework acceleration, this work is the first to demonstrate the power of combining a data-driven hardware-software co-design methodology based on large-scale profiling with the promise of agile, open hardware development methodologies. In this vein, our entire evaluation flow (RTL, benchmarks, including hyperscale-derived benchmarks, and supporting software and simulation infrastructure) has been open-sourced for the benefit of the research community.

6.9 Retrospective

This work is a key enabler for future hyperscale HW research and is seeing re-use in several ongoing hyperscale HW research projects, as many domain-specific architectures for hyperscale systems see Amdahl bottlenecks if (de)serialization is not accelerated (e.g., [80]). As the push to-

wards microservices continues, we also expect further growth in usage of RPC stack components like protobufs. This work has also influenced industry, including various silicon vendors and underwent MICRO's artifact evaluation process and received all available badges, including results being reproduced. This work received the Distinguished Artifact Award at MICRO '21 and was selected as an IEEE Micro Top Picks 2021 Honorable Mention.

Chapter 7

CDPU: Accelerating general-purpose lossless (de)compression at scale

In this chapter, we employ the data-driven hardware/software co-design methodology described earlier to understand and accelerate another critical system-level overhead in hyperscale servers: general-purpose lossless compression and decompression. Using insights from Google's hyperscale datacenter fleet, we design custom hardware to accelerate multiple compression and decompression algorithms and perform a design-space exploration of our accelerators in the context of Hyperscale SoC. Note that representations of fleet data in this chapter are as they were in the original publication of this chapter at ISCA 2023 [114].

7.1 Introduction

General-purpose lossless data compression and decompression (referred to as "(de)compression" in this work) are another set of critical system-level overheads (or "datacenter taxes" [110, 198]) in hyperscale contexts. Most datacenter taxes implement critical *functionality* like inter-service communication, security, or memory movement. In contrast, (de)compression is unique in that its purpose is not to add functionality, but to *enable a trade-off* between the consumption of two classes of WSC resources: runtime (CPU cycles) and storage/communication capacity (persistent storage capacity, memory capacity [129, 217], and network bandwidth). Unlike other datacenter taxes, service developers must first decide whether to compress at all, and then select an algorithm that achieves satisfactory compression quality within their constraints.

This presents an interesting opportunity for hardware acceleration: an accelerator that radically outperforms software implementations can not only reduce existing CPU cycles in the fleet, but also increase compression usage in general, leading to additive savings in storage, memory, and network bandwidth. However, introducing specialized hardware complicates the design space; the total cost of ownership (TCO) calculation must now account for hardware complexity, area vs. performance, and more.

In this work, we present the first large-scale data-driven analysis of lossless data (de)comp-

ression usage at a major cloud provider by profiling Google's datacenter fleet. We find that (de)compression consumes 2.9% of fleet CPU cycles and 10% to 50% of CPU cycles in key services at Google. This demand is also artificially limited; 95% of bytes compressed in Google's fleet forgo more aggressive forms of compression because of the high compute cost, motivating HW acceleration that changes time vs. data size trade-offs. While profiling fleet usage is helpful, we also find that true co-design for (de)compression processing units (CDPUs) requires a comprehensive evaluation environment, due to the large number of high-level design parameters and their impact on end-to-end performance.

A large body of prior work has improved the microarchitectural state-of-the-art for CDPUs supporting various algorithms in fixed contexts [176, 47, 185, 74, 7, 186, 171, 101, 172, 210, 3, 48, 211, 53, 130, 162, 99, 100]. While these improvements are important, we find that higher-level design parameters like accelerator placement, hash-table sizing, history window sizes, and more can have just as significant of an impact on the value and feasibility of CDPU integration, but are not well-studied in the literature. Thus, we present the first end-to-end design and evaluation framework for CDPUs, which includes: 1. An RTL-based CDPU generator that supports many run-time and compile-time configurable parameters. 2. Integration into a RISC-V SoC for rapid performance and silicon area evaluation with varying CDPU placements and configurations. 3. A (de)compression benchmark, HyperCompressBench, that is representative of (de)compression usage in Google's fleet. All components of this framework are open source[1], enabling the community to build and evaluate CDPUs for both hyperscale systems and their own use cases.

Using our CDPU design framework, we perform an extensive design space exploration running HyperCompressBench. Our exploration spans a 46× range in accelerator speedup, 3× range in silicon area (for a single pipeline), and evaluates a variety of accelerator integration techniques to better understand optimal CDPU designs for hyperscale contexts. Our final hyperscale-optimized accelerator instances are up to 10× to 16× faster than a single Xeon core, while consuming a small fraction (as little as 2.4% to 4.7%) of the area.

7.2 General-purpose lossless compression and decompression fundamentals

Compression algorithms are used to produce a reduced-size representation of source data that can later be fed to a decompressor to exactly reproduce the original data. While the functional goal is only to minimize the output size (maximizing the *compression ratio*, equal to uncompressed divided by compressed size), algorithms must also account for metrics like latency, throughput, and CPU/memory consumption, resulting in a vast design space. In a hyperscale context, compression reduces the consumption of several resources, including storage (bytes written to disk/SSD), network bandwidth (e.g., RPC traffic), and memory (transparently [129, 217] or via application

[1] **CDPU generator, custom Chipyard, custom FireSim**: `https://github.com/ucb-bar/compress-acc`, **HyperCompressBench:** `https://github.com/google/HyperCompressBench`, **Archival URLs:** See Artifact appendix: CDPU (Appendix C)

managed compression). Compression can also implicitly save other resources such as caches, network-on-chip capacity, etc., but we do not explore these in this work.

Compression algorithm fundamentals

Compression algorithms generally contain two main components: a dictionary-coding stage and an entropy-coding stage. During *dictionary coding*, data size is reduced by searching for matches between the input data and a "dictionary" of known values, then encoding the input in terms of the "best" match in the dictionary, de-duplicating repeated strings in the input. LZ77 [233] is a widely used dictionary coding algorithm that uses a sliding window of already processed input data as the dictionary. Matches are encoded as triplets of `(offset, length, literal)`. Such a triplet indicates to the decompressor that `length` bytes should be copied to the output starting from `offset` bytes back in the window of output generated so far. Then, the raw `literal` is also copied to the output, for example to encode data when no matches were found in the dictionary.

Entropy coding compresses symbols (e.g. `(offset, length, literal)` triplets produced by LZ77) by representing more commonly occurring symbols with fewer bits. Popular techniques include Huffman coding [94], arithmetic coding, and Asymmetric Numerical Systems (ANS) [60, 68]. Huffman and arithmetic coding trade-off compression ratio and performance—Huffman is cheaper in CPU cycle cost, but arithmetic coding generally achieves a better compression ratio. ANS (such as tANS/FSE [60, 68]) combines the best of both worlds, with low CPU cost and high compression ratio.

Compression algorithm taxonomy

Compression algorithm developers trade-off compression ratio vs. performance by combining these components in novel ways and tuning parameters within them. For example, they can choose how much effort to expend trying to find an "optimal" match during LZ77-style dictionary coding or change the size of the sliding history window. A larger *window size* typically yields better compression ratios but must be bounded to limit memory consumption. Many algorithms accept a *compression-level* parameter, which also allows users to tune algorithm performance.

In Chapter 7.3, we will analyze six algorithms that are used in Google's fleet. We qualitatively group these into "heavyweight" and "lightweight" classes (which we will justify quantitatively in Chapter 7.3):

Heavyweight algorithms: These prioritize compression ratio over speed. They generally have a large space of parameters and use sophisticated LZ77/entropy-coding techniques.

- **ZStd** [235, 51]: LZ77/Huff./FSE. Params: comp. level + window size.
- **Flate** [234, 59]: LZ77/Huff. Params: comp. level + window size.
- **Brotli** [39, 8]: LZ77/Huff./context modeling/static dictionary [9]. Params: compression level + window size.

Lightweight algorithms: These prioritize speed over compression ratio. They generally use "LZ77-inspired" dictionary coding, little or no entropy coding, and have few parameters.

- **Snappy** [197, 195]: LZ77-inspired, no entropy coding. Fixed window size (64 KiB), no compression levels.
- **Gipfeli** [79, 138]: LZ77-inspired, simple entropy coding. Fixed window size (64 KiB), no compression levels.
- **LZO** [145, 206]: LZ77-inspired dictionary coding, no entropy coding. Supports compression levels.

ZStd and Brotli can also become more lightweight by setting a low compression level. We will explore this in Chapter 7.3.

7.3 Profiling (de)compression usage at hyperscale

In this section, we profile the fleet-wide usage of (de)compression in Google's datacenters to motivate the design of a CDPU and understand design constraints.

Data Sources

Fleet-wide CPU Cycle Data

Google's infrastructure provides fleet-wide runtime information about CPU-cycle consumption using a sampling framework, Google-Wide Profiling (GWP) [180], that randomly samples fleet servers. When a server is profiled, the sampler collects profiles including workload names, stack traces, and cycle counts, enabling determination of where time is spent in the software stack. We use this to classify fleet-wide CPU-cycles spent in (de)compression by algorithm.

Fleet-wide compression/decompression call sampling

An extension of this sampling framework also enables detailed profiling of (de)compression calls in userspace, including collecting the algorithm used, input and output sizes, window sizes, and compression levels. Given the additional engineering effort this requires, data is only collected for the Snappy, ZStd, Flate, and Brotli algorithms, which, as Figure 7.1 shows, are the dominant algorithms in the fleet.

Opportunity for (De)compression Acceleration

WSCs today spend significant compute on (de)compression. In Google's infrastructure, 2.9%[2] of *fleet-wide* CPU cycles are spent in (de)compression; 56% of these cycles are spent in decompression and the rest in compression.

[2]At hyperscale, this can translate to 100s of millions of dollars [24, 198].

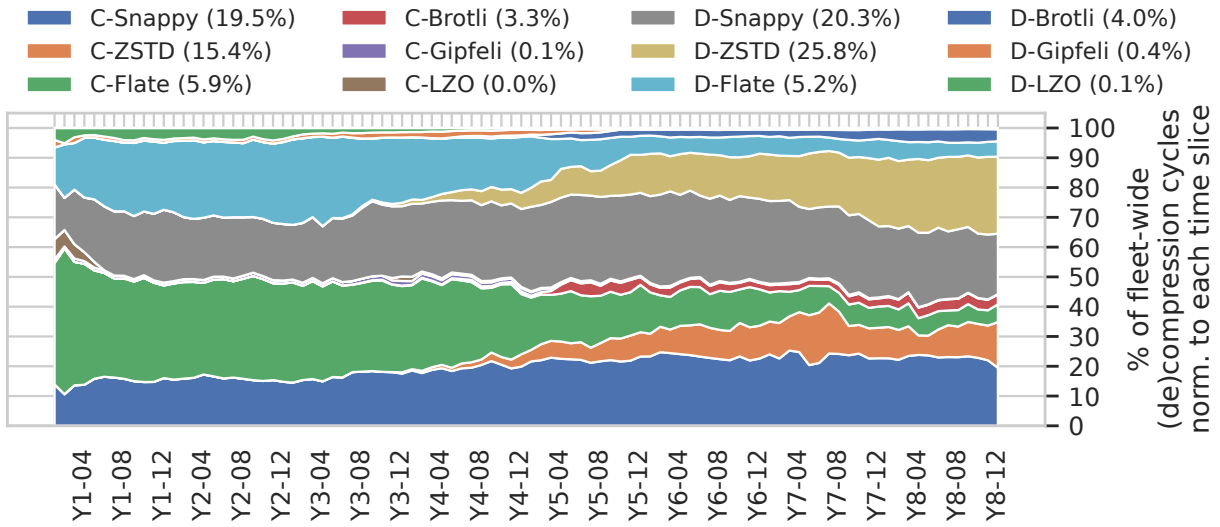


Figure 7.1: Percentage of (de)compression cycles in Google's fleet over several years, broken down by algorithm and normalized to each month. C = compression, D = decompression.

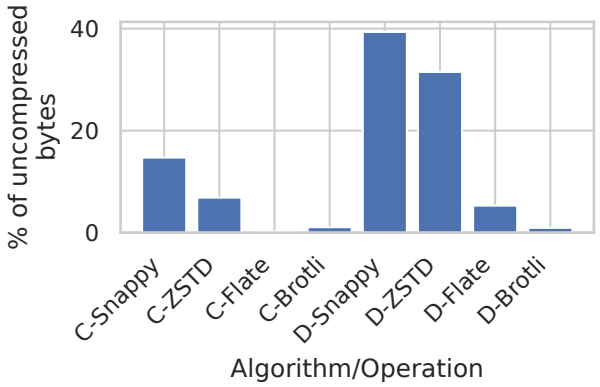
For large services, (de)compression can be a much greater proportion of total cycle consumption. We find that a total of sixteen services constitute around half of all fleet-wide cycles for Snappy and ZStd[3] (de)compression. Out of these, one service spends nearly 50% of its total cycles on (de)compression, another spends over 35%, and eight more spend between 10% and 25% of their cycles each on (de)compression. *Even ignoring potential growth in demand, these represent a significant acceleration opportunity at hyperscale*.

Can accelerators change WSC resource tradeoffs?

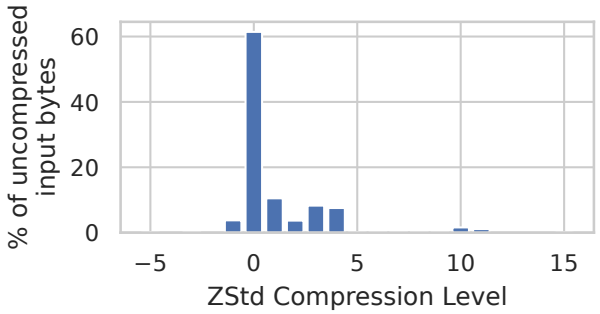
While reducing the existing CPU cycles spent on (de)compression is a useful goal, it is important to note that this usage is a function of the performance constraints imposed by current software libraries. When considering the introduction of specialized hardware, we must keep in mind that the accelerator is likely to change the "space" (storage/memory bytes, network bandwidth) vs. "time" (runtime, CPU cycles) trade-off involved in selecting a compression algorithm, that algorithm's parameters, or indeed, choosing to compress at all in a given situation.

In an ideal scenario, the accelerator would sufficiently reduce the performance overhead of "heavyweight" forms of (de)compression such that services can always choose them over "lightweight" techniques (or even no compression), and reduce storage, memory, and network bandwidth consumption for "free". To understand this opportunity, we must answer four key questions:

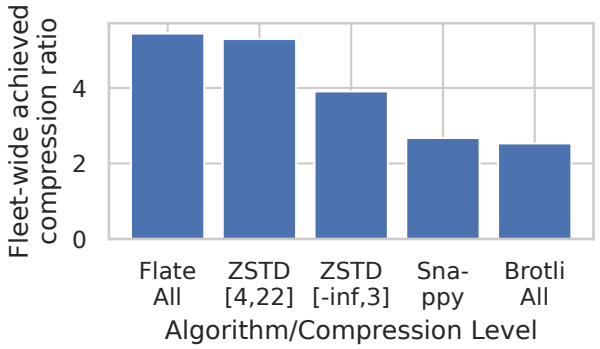
[3]In the rest of the chapter, we focus on Snappy and ZStd as dominant representatives of "lightweight" and "heavyweight" algorithms in the fleet respectively.



(a) Fleet-wide uncompressed bytes handled by (de)compression, broken down by algo.



(b) Fleet-wide ZStd compression level distribution.



(c) Aggregate fleet-wide compression ratios achieved, by algo./compression level pairs.

Figure 7.2: Google fleet-wide (de)compression algorithm breakdowns. C = compression, D = decompression.

Do existing services prefer to use heavyweight or lightweight algorithms?

Figure 7.1 shows a detailed breakdown of CPU time spent in the fleet on compression and decompression by algorithm, self-normalized to each month. In this sub-section, we focus only on the final time slice, which is summarized in the legend. In addition to cycle consumption, we would also like to understand the *amount of data* that each algorithm is invoked on. Figure 7.2a thus differentiates algorithms based on the number of uncompressed bytes they handle in the fleet (i.e., compression inputs and decompression outputs).

We find several interesting trends from this data. For compression, where the heavyweight vs. lightweight distinction is most significant, we see that slightly more *cycles*, 56%, are spent in heavyweight compression. However, from the perspective of *bytes* handled, the outcome is *reversed*: heavyweight compression only accounts for 36% of the total. This foreshadows the difference in cost-per-byte-compressed between heavyweight and lightweight compression, explored in greater detail in Chapter 7.3. In decompression, the CPU consumption imbalance between heavyweight and lightweight is far more stark, but the cost-per-byte is closer: heavyweight algorithms comprise 63% of fleet decompression *cycles*, while producing 49% of uncompressed *bytes*.

As an aside, Figure 7.2a also shows an interesting insight: on average, each byte that is compressed in the fleet is decompressed 3.3 times. So, despite a lower cost-per-byte, decompression remains a worthy target for hardware acceleration. Further, decompression is often more performance-sensitive, naturally appearing on client-visible read paths, rather than usually non-critical write paths.

Are heavyweight algorithms used to their full potential?

Generally, this requires supplying a larger *compression-level* argument to the algorithm, instructing it to spend more cycles improving the compression ratio. Consider ZStd compression, which currently supports levels from negative infinity to 22. Figure 7.2b shows the distribution of bytes passed to ZStd compression calls in the fleet, binned by the associated compression level specified by the caller. We find that even services that use ZStd tend to avoid high compression levels: 88% of bytes are compressed at level 3 (the default) or lower, while over 95% of bytes are compressed at level 5 or lower. Fewer than 0.002% of bytes are compressed at levels $\geq$12.

Combining the data in Figures 7.2a and 7.2b we glean a critical insight: over 95% of bytes compressed in the fleet are handled either by a lightweight algorithm (Snappy) or a heavyweight algorithm at low compression level (ZStd at level ≤ 3). This suggests that *there is significant opportunity for an accelerator that can achieve higher compression ratios within existing performance bounds* to produce significant savings in storage, network, and memory consumption.

Do high compression levels result in improved compression ratio?

Of course, the goal of using heavyweight algorithms configured to high compression levels is to achieve a better compression ratio. Therefore, we must understand whether this improvement is indeed notable.

Before we present this data, it is important to caveat that extrapolating from this data is generally difficult due to the highly data-dependent nature of both compression ratio and cycles-per-byte terms. While the data gives the reader an estimate of possible improvements and is valuable because it is based on large fleet byte volumes, a true comparison of algorithms/levels requires running the same sets of representative data through algorithms/levels of interest. We will address this in Chapter 7.4 when we construct our benchmark suites.

Figure 7.2c shows the aggregate fleet-wide compression ratio achieved by each compression algorithm (*total* uncompressed bytes divided by *total* compressed bytes). ZStd bins are further separated by the user-specified compression level. We can see that compression is clearly beneficial across the fleet, with no algorithm having an aggregate compression ratio less than 2. Furthermore, the data aligns with expectations from the taxonomy we established in Chapter 7.2. ZStd and Flate clearly belong in the heavyweight category, exceeding Snappy's compression ratio even at the lowest compression levels. Brotli results do not align with our taxonomy because most of its usage in the fleet is at low compression levels.

Quantitatively, we observe a favorable trend in the data to justify hardware acceleration. Services that use ZStd at a low compression level achieve a 1.46× improved compression ratio over services that use Snappy. Services that use ZStd at a high compression level achieve an additional 1.35× improved compression ratio over services that use it at a low compression level. It is also important to note that this is likely under-representing the potential of ZStd's highest compression-levels, since Figure 7.2b showed that the vast majority of bytes in the `[4, 22]` bin in Figure 7.2c are compressed at level 4, due to the aforementioned performance constraints.

In a hyperscale context, the corresponding reductions in demand for storage, network, and memory capacity that arise from these differences in compression ratio can translate to a further potential savings of hundreds of millions of dollars across the industry [24, 198, 113], in addition to the savings from CPU-cycle reduction/offloading: most hyperscaler customers are big-data companies who spend as much on storage as compute [178], while memory has been shown to be 50% of WSC TCO [31], and providing sufficient network bandwidth at low cost is a perpetual concern for hyperscalers [193].

What is the cycle cost in software of using heavyweight algorithms at high compression level in the fleet? Is hardware acceleration necessary?

Given the marked difference in achieved compression ratio using different algorithms/levels, one might ask: why not simply migrate to heavyweight algorithms at high compression levels in software?

To answer this question, we collect data on the aggregate cost-per-byte observed in the fleet for each algorithm, operation, and compression level of interest thus far. We find that our taxonomy from Chapter 7.2 is largely validated: both heavyweight compression and decompression are more expensive per-byte than lightweight compression and decompression respectively. We also find that services that use ZStd compression at lower compression levels pay 1.55× the cost-per-byte for compression as compared to those that use Snappy, and services that use ZStd compression at higher compression levels over lower compression levels pay an additional 2.39× cost-per-byte.

Extrapolating from this data (and keeping in mind caveats about the data-dependent nature of compression), if a service spends 25% of its cycles on Snappy compression (e.g., the services described in Chapter 7.3), switching to the highest ZStd levels would result in a 67% increase in the service's cycle consumption, a non-starter. There is also a significant additional cost for decompression, when the data is accessed later; ZStd decompression is 1.63× more costly than Snappy decompression, partially due to the entropy decoding.

Altogether, this profiling data suggests that there is significant headroom for services to achieve improved compression ratios for the deployed algorithms, but the cost of these algorithms in software is too high for services to adopt them. This suggests that *hardware-accelerated compression has the opportunity to save not only CPU cycles, but also to save storage/memory/network resources by changing the trade-off space between performance and compression ratio.*

Algorithm evolution vs. hardware accelerator design cycles

Even when hardware acceleration is well-motivated by projected resource savings, a significant roadblock to adoption is the opportunity cost of "ossification" of logic in hardware, since hardware design cycles are significantly longer than software development cycles. However, given the need for long-term stability of compression algorithms (e.g. for data written to cold storage), significant algorithm change generally only occurs when a completely new algorithm is adopted by a service. Referring back to Figure 7.1, we can observe the introduction of the ZStd algorithm in Google's fleet, which took roughly a year from being introduced to consuming 10% of fleet (de)compression cycles. While this broadly aligns with hardware design cycles, starting a design from scratch and deploying it in this timeframe would be challenging.

This suggests that *an agile hardware development approach is necessary*, with early hardware/software co-design with algorithm developers and utilization of (de)compression accelerator *generators* that provide high-performance primitives that are common across multiple algorithms, alleviating the need to write entire accelerators from scratch. For example, transitioning from Flate to ZStd would mostly entail adding an FSE module. This methodology is explored in Chapter 7.5.

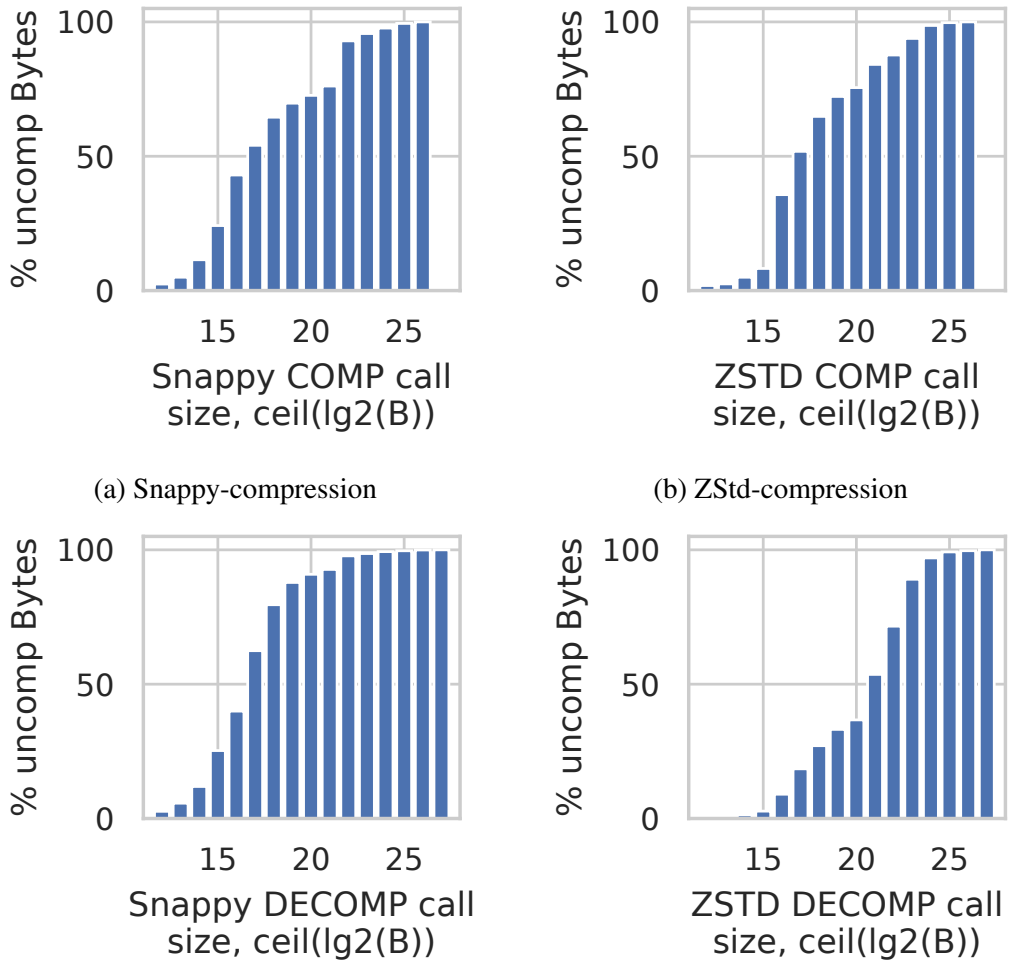
When compared to other datacenter taxes, (de)compression also has a qualitative advantage when considering hardware acceleration feasibility: the user API for compression and decompression has been essentially unchanged since the first compression tools were created—a stateless, buffer-in, buffer-out API, sometimes with a separate dictionary, and a streaming equivalent.

(De)Compression Accelerator Placement

In this section, we discuss the factors impacting an important choice in CDPU design: where to place it in the system: on-die, on a PCIe-attached device, or on a chiplet.

(De)compression call granularity

The granularity of offloaded work—in this case, the number of bytes to be (de)compressed—is a key factor in determining placement, since any overhead per accelerator invocation is only



(a) Snappy-compression

(b) ZStd-compression

(c) Snappy-decompression

(d) ZStd-decompression

Figure 7.3: Cumulative call-size distributions for Snappy/ZStd (de)compression. The x-axis bins calls by $log2(callsize)$, using uncompressed sizes. The y-axis is weighted by call size.

amortized over each payload size.

Figures 7.3a and 7.3b show the cumulative distribution of call granularities for Snappy and ZStd compression. Snappy's distribution is slightly more biased towards smaller calls: 24% of bytes compressed are from calls of size 32 KiB or smaller; for ZStd only 8% fall in this group. Interestingly, the distributions align near the median, with the 50th percentile of uncompressed bytes falling between 64 and 128 KiB calls in both. For ZStd, much of this jump comes from the `(32 KiB, 64 KiB]` bin, which represents 28% of bytes compressed. Apart from the 16.8% of bytes compressed by Snappy in the `(2 MiB, 4 MiB)` bin, both distributions increase uniformly until reaching a maximum size of 64 MiB.

Figures 7.3c and 7.3d show the corresponding data for decompression. Immediately, we see that Snappy's decompression distribution is slightly more biased towards smaller calls than its compression distribution, with 62% of bytes handled in calls smaller than 128 KiB and 80% of bytes handled in calls smaller than 256 KiB. On the other hand, the ZStd decompression distribution shifts drastically towards larger sizes, with the median size between 1 MiB and 2 MiB, rather than between 64 KiB and 128 KiB as for compression.

A back of the envelope projection of accelerator performance ranges shows that these distributions are insufficiently skewed to immediately fix accelerator placement. In contrast, if hypothetically most calls were 32 MB, a PCIe-attached accelerator would be a natural choice. As we will see in Chapter 7.6, both call sizes and various accelerator tuning parameters play important roles in determining accelerator placement; a comprehensive design-space exploration will be necessary to make a final determination.

Interaction with Related Accelerators

With increasing hardware specialization, we envision a future where our (de)compression accelerator is invoked in conjunction with related accelerators (e.g., a hardware protocol buffer (de)serializer [113, 105, 170]) as part of a larger data-access operation. While the hardware benefits of such a system are self-evident, the corresponding software services and libraries need to be architected appropriately as well.

Figure 7.4 shows fleet (de)compression cycles classified by the codebase (e.g. a library) that directly called the (de)compression operation. Note that 49% of cycles are derived from "file formats". Upon closer examination, we notice that even if these formats are internally "serializing and compressing protobufs" before writing to file, there are often small, unrelated book-keeping operations between the two accelerated operations. Services may also expect to pass in a sequence of serialized protobufs that are accumulated and compressed periodically. Handling these in hardware introduces significant complexity due to file-format specific logic and the need to track outstanding state, and can also limit file-format evolution.

This argues for *placing both accelerators close to the CPU cores*, utilizing the CPU caches or even main memory as the intermediate storage, allowing the general-purpose cores to sequence data movement between them in the normal course of program execution, without undue communication overhead. If the accelerators are far away, for example across PCIe, the operation would incur substantial offload overhead multiple times, making the use of each accelerator less attrac-

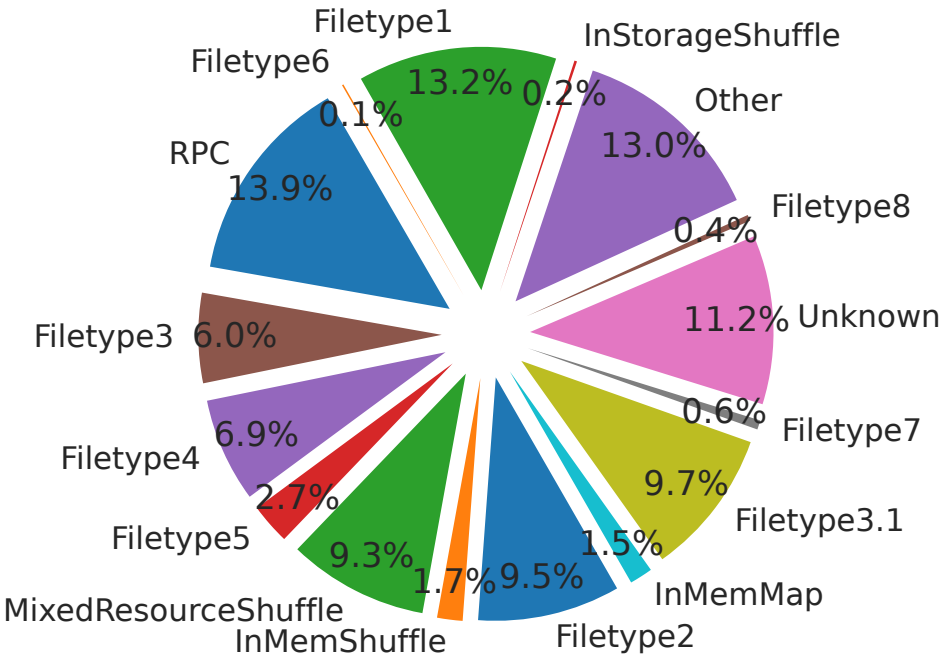


Figure 7.4: Percent of Google fleet-wide (de)compression cycles by the library that led to the (de)compression call.

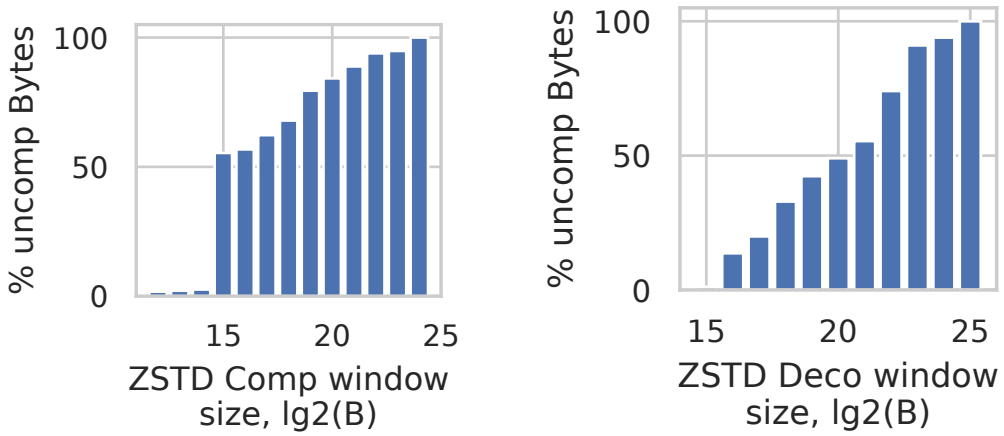
tive. In the long run, the potential performance gains may justify additional software engineering effort in file formats to enable the exploitation of sequences of hardware accelerators; this is left to future work.

Window Size Requirements

A compression algorithm's window size determines the amount of recent history the algorithm will keep when searching for matches during LZ77-style de-duplication. Correspondingly, during decompression, the window size represents the maximum offset into the recently produced output from which a copy command can read data.

Our first algorithm of interest, Snappy, has a *fixed* window size of 64 KiB for compression and decompression [197]. For ZStd compression, Figure 7.5a shows the per-call fleet-wide window-size distribution. We see that slightly over 50% of bytes compressed by ZStd use a window size of 32 KiB or less. However, the upper 50% of the distribution quickly grows, with a 75th percentile between 512 KiB and 1 MiB and tails as high as 16 MiB. The distribution for ZStd decompression is shown in Figure 7.5b, with a median of 1 MiB.

This parameter can affect accelerator design and performance. For compression, the window is commonly kept in SRAM, registers, or even expensive CAM structures. For decompression, the window is commonly kept in SRAM. However, beyond for example 32 KiB, on-chip storage can



(a) ZStd-compression

(b) ZStd-decompression

Figure 7.5: Window size distributions for ZStd (de)compression in Google's fleet. The x-axis bins calls by $log2(windowsize)$. The y-axis is weighted by call size.

become prohibitively expensive. Notably, the existing state-of-the-art compression accelerator for a heavyweight algorithm, IBM's z15 compression accelerator [3], offers a window size of 32 KiB, meaning it would not be able to handle 50% of these compression calls in Google's fleet.

This further argues for a near-core accelerator with access to the memory hierarchy, which would allow the accelerator to "fall back" to accessing the history from the L2 cache or main memory. This design space is further explored in Chapter 7.6.

Do existing open-source compression benchmarks represent hyperscale requirements?

Several of our analyses thus far have motivated the need to perform a design-space exploration of (de)compression acceleration within the context of a complete system. However, performing such an exploration requires representative (de)compression benchmarks used as input to the accelerators.

Many benchmark suites exist that aim to provide a standard set of input files to evaluate compression algorithms. The most well-known of these is Silesia [58], which, for example, is used to provide the "default" results in the `README`s of both ZStd and `lzbench`, a common compression benchmarking tool. Other commonly used benchmarks (e.g., in [3]) include Canterbury [19], Calgary [32], and several benchmarks included with Snappy (we will refer to the collection as SnappyFiles) [196].

Unfortunately, we find that they are not representative of Google's fleet usage of (de)compression. As one dimension of comparison, we can bin these open benchmarks by call size as we did for fleet-wide compression calls in Figure 7.3. Figure 7.6 shows this call-size distribution

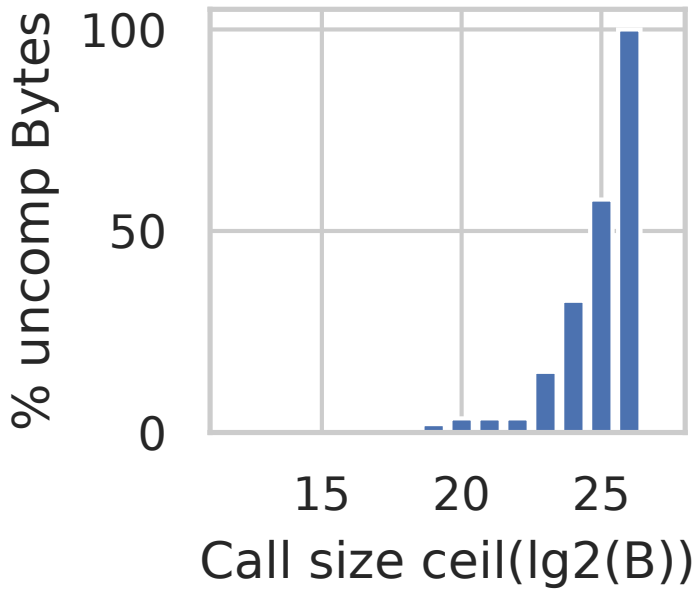


Figure 7.6: Call size distribution from four popular open-source compression benchmarks.

for open-source benchmarks, which we can see is vastly different from the fleet distribution. For example, the median call sizes of the distributions differ by an astounding 256×. More work is clearly needed to realistically evaluate compression in a hyperscale fleet. We will describe the construction of representative benchmarks in Chapter 7.4.

Key Cloud Provider Fleet Profiling Lessons for Hyperscale CDPUs

Before continuing, we summarize the key profiling insights gleaned thus far and highlight the important questions that remain:

1. Significant headroom exists in fleet compression usage for accelerators that improve compression ratio vs. compute tradeoffs:

    a) Lightweight algorithms dominate compression usage, handling 64% of compressed bytes.

    b) Heavyweight algorithms are primarily used at lower compression levels: 88% of bytes compressed with ZStd are handled at level 3 (the default) or lower.

    c) Services using heavyweight algorithms at high levels achieve higher compression ratios (1.35-1.97×), but at a significantly higher cost-per-byte (1.55-3.70×).

    d) For many services, this increased CPU cost is untenable, presenting an opportunity for accelerators that achieve higher compression ratios within service performance bounds.

2. Change in (de)compression algorithm usage in Google's fleet over time (e.g., ZStd's 0% $\rightarrow$ 10% of fleet (de)compression cycles in 1 year) aligns with agile hardware design cycles and motivates a re-usable CDPU *generator* over point designs.

3. Fleet call sizes are not sufficiently biased towards small/large calls to immediately determine accelerator placement.

    a) Instead, understanding placement requires design-space exploration of an implementation running representative benchmarks.

4. Accelerator chaining between serialization and compression, which could ease placement requirements, is non-trivial.

    a) At a minimum, chaining will require re-architecting file format libraries, which are responsible for invoking 49.2% of fleet (de)compression cycles, and the ability to maintain multiple contexts in the accelerator.

    b) On the other hand, both of these concerns can be avoided while maintaining most chaining benefits if the accelerator is placed close to the CPU, with direct access to caches or main memory.

5. History window sizes in the fleet are also insufficiently biased to make a clear recommendation for on-accelerator history window sizing.

    a) Like accelerator placement, this will require design-space exploration of an accelerator implementation.

6. Existing open-source (de)compression benchmarks used by prior work do not represent hyperscale compression usage.

    a) For example, call-size distributions differ greatly between open-source benchmarks and Google's fleet, even at a high-level; the median call size in popular open-source benchmarks is 256$\times$ the fleet's median call size.

While hyperscale fleet profiling has provided several insights about CDPU design requirements, a few critical questions remain that are difficult to explore without a concrete implementation evaluated in the context of a complete system. In the rest of this chapter, we will build a parameterized CDPU generator and a hyperscale-representative (de)compression benchmark suite, then answer the open CDPU design questions by performing an extensive design-space exploration.

7.4 Building open-source hyperscale-representative (de)compression benchmarks

To produce (de)compression benchmarks representative of Google's fleet requirements, while preserving privacy in Google's datasets, we build an open-source (de)compression benchmark gen-

erator that produces representative benchmarks from summary statistics about a private data set. By supplying this generator with profiles of Google's fleet, we produce the open-source *HyperCompressBench*, a benchmark suite representative of (de)compression requirements in Google's fleet.

The generator starts by breaking all files from the Silesia, Canterbury, Calgary, and SnappyFiles benchmarks into fixed-size chunks. Each chunk is individually run through all combinations of supported algorithms and parameters (window size, compression level) to obtain a compression ratio for that chunk for each algorithm/parameters pair. This data is stored in lookup tables indexed by the compression ratio.

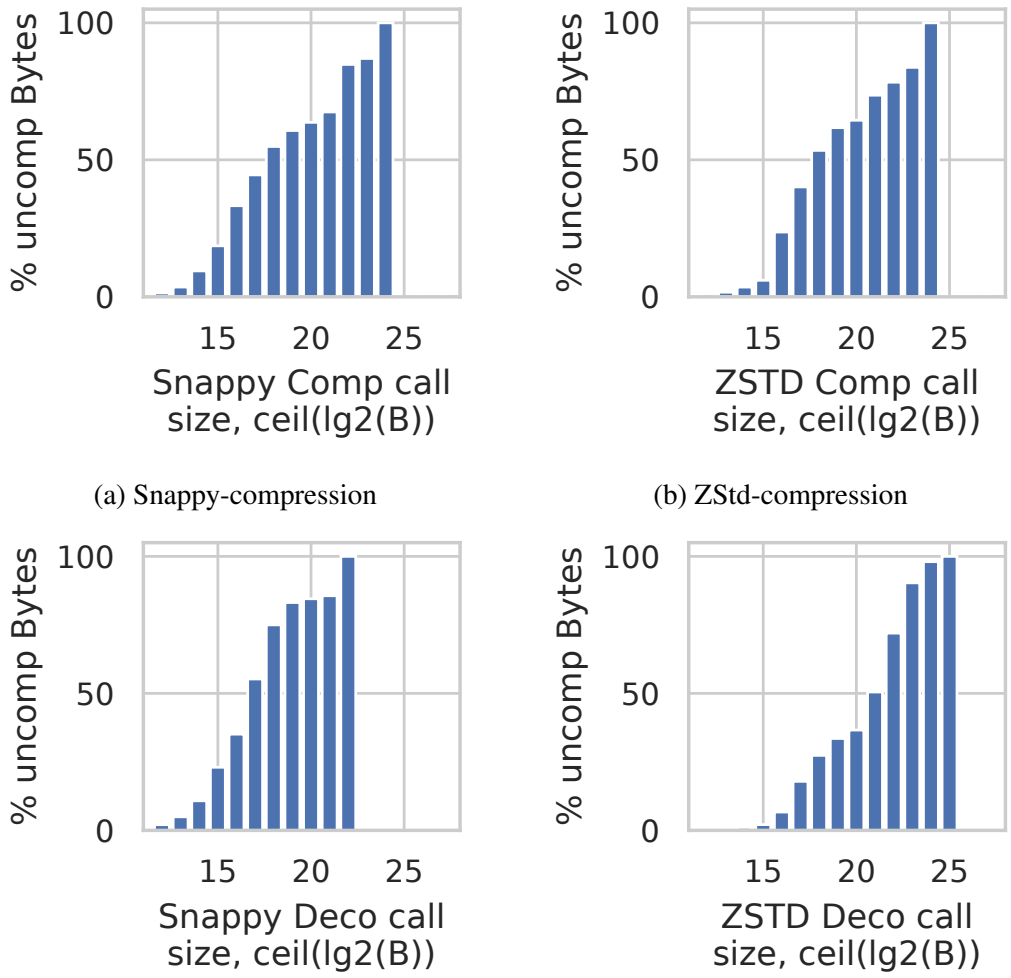
The generator then ingests metrics such as call size, compression ratio, window size, and compression level from the aforementioned fleet profiling data, constructs distributions from these metrics, and samples from the distributions to produce a set of target parameters for a single benchmark file.

For each such set of target parameters, the generator walks through the lookup table, greedily selecting chunks with the closest compression ratio and adding them to the output file until the target call size is reached. At various points during this process, the generator evaluates the file assembled so far and adjusts the target ratio accordingly. To avoid pathological sequences, random shuffles are introduced both within the lookup table and the output. The completed file is saved along with the parameters (level and window size) that should be applied when it is used. This process is repeated until we have a sufficient number of benchmark files to represent the overall distribution of calls across various dimensions. We find around 8,000 to 10,000 files to be a suitable number for this work.

The entire process is repeated for each algorithm/operation pair of interest, in our case, (Snappy and ZStd) $\times$ (Compress and Decompress). In the rest of the chapter, we refer to this suite of around 35,000 generated files as *HyperCompressBench*.

HyperCompressBench validation

We validate the suite across the swath of previously discussed metrics. For example, consider the distributions for call size, shown in Figure 7.7. We can see that these line-up very well with the fleet distributions from Figure 7.3 and preserve the shape of each algorithm/operation pair's unique distribution, in stark contrast to the existing open-source benchmark suite call-size distributions. Between each pair of distributions, the only significant difference is in the largest size bins—this is because these call sizes represent an extremely small proportion of uncompressed fleet bytes and thus are unlikely to be included in an 8,000 to 10,000 benchmark sample. Comparing compression ratios, we find that on average for each suite, achieved compression ratios are within 5%-10% of fleet compression ratios. While elided due to space constraints, the distributions for compression level and window sizes are also extremely similar to the fleet distributions shown in Figures 7.2b and 7.5 respectively.



(a) Snappy-compression

(b) ZStd-compression

(c) Snappy-decompression

(d) ZStd-decompression

Figure 7.7: Call-size distributions for HyperCompressBench.

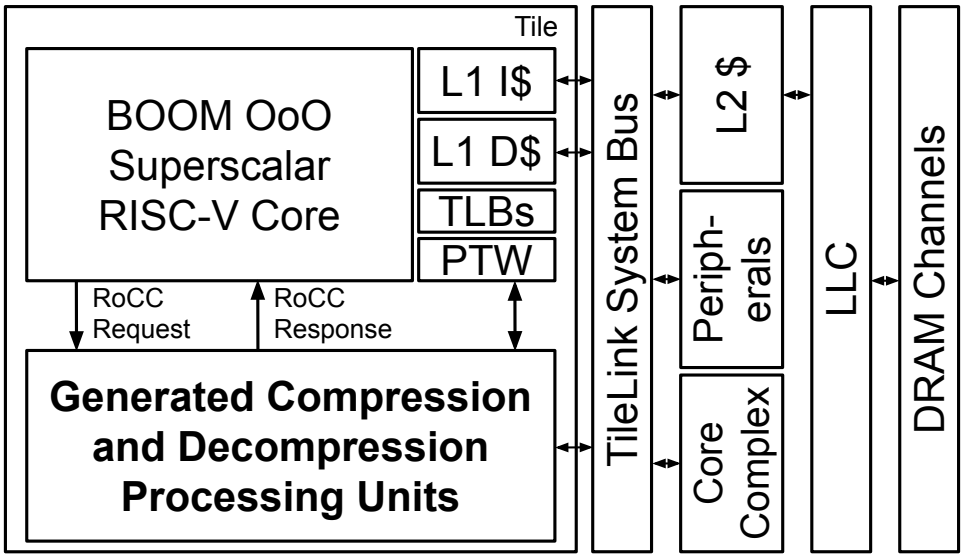


Figure 7.8: Top-level RISC-V SoC block diagram with CDPUs.

7.5 A parameterized generator for compression and decompression processing units (CDPUs)

Our open-source CDPU generator is implemented in Chisel RTL [25] and incorporated into the Chipyard RISC-V SoC generator ecosystem [14]. Figure 7.8 shows the overall architecture of the accelerated SoC, which is configured to use BOOM, an OoO superscalar RISC-V core with performance comparable to ARM A72-like cores [230].

The generated accelerators receive commands directly from the BOOM application core in the SoC via the RoCC interface [22], which allows the CPU to directly dispatch custom RISC-V instructions in its instruction stream to the accelerator within a few cycles. These *RoCC instructions* [22] can supply two 64-bit register values from the core to the accelerator. The accelerator accesses the same unified main memory space as the CPU through the 256 bit-wide TileLink-based NoC [97] and can issue memory requests with virtual addressing. As shown in Figure 7.8, all memory accesses made by the accelerator go through the L2 and LLC, which are shared with the application cores in the system.

Figures 7.9 and 7.10 show the block diagrams of complete decompressors and compressors respectively. Both handle the Snappy and ZStd algorithms. In these diagrams, components used by both algorithms are shown with a solid outline, components used only by Snappy with a dotted outline, and components used only by ZStd with a dashed outline. In the following subsections, we will outline the generator's library of reusable components used to build the aforementioned compressors and decompressors and give an overview of high-level parameters that can be modified.

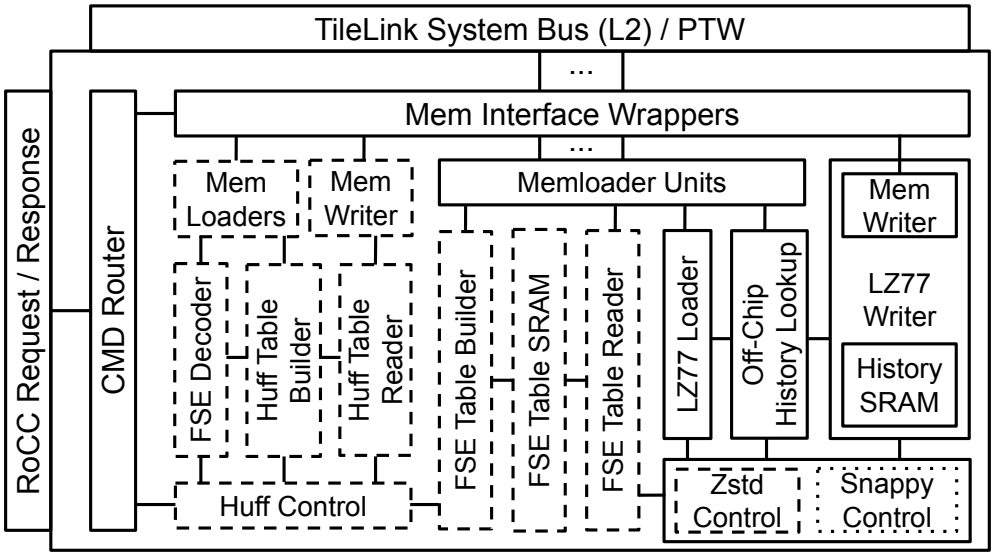


Figure 7.9: Block diagram for CDPU decompressor with support for Snappy and ZStd.

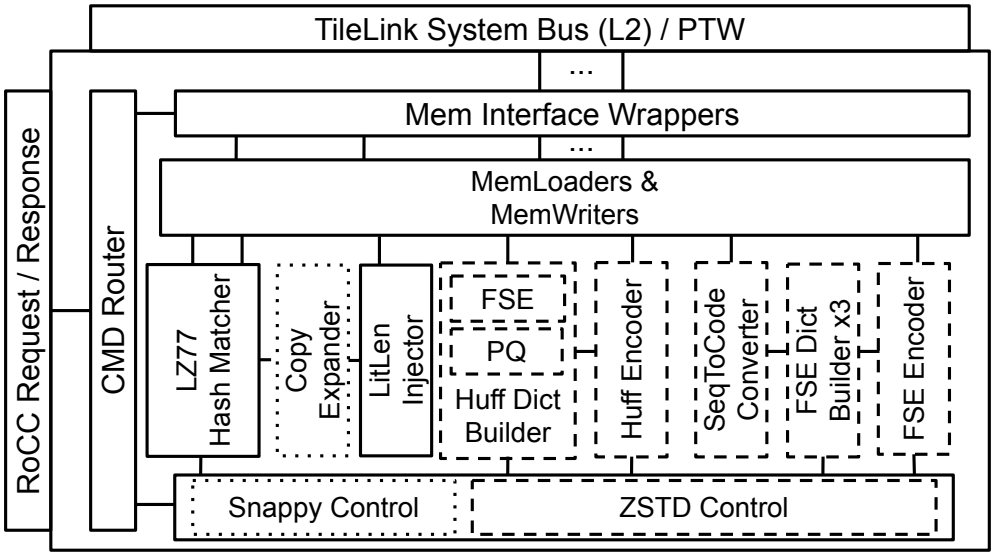


Figure 7.10: Block diagram for CDPU compressor with support for Snappy and ZStd.

System Interface Blocks

Our generator uses three types of blocks to interface accelerators to the rest of the system. *Memloaders* support streaming from the L2 cache, *Memwriters* support streaming to the L2 cache, and *CommandRouters* dispatch incoming commands to the appropriate sub-blocks. These are visible in both Figures 7.9 and 7.10.

LZ77 Decoder

The LZ77 Loader, Off-Chip History Lookup, and LZ77 Writer in Figure 7.9 comprise the LZ77 Decoder. This unit consumes sequences of `(offset, length, literal)` from a compressed input and produces the final stream of decompressed output data. The block primarily consists of a history window SRAM used to lookup matches based on offset and length, with the ability to fall back to making memory requests for matches that are further away than the configured size of the history SRAM.

Huffman Expander

The Huff Table Builder, Reader, and Control in Figure 7.9 comprise the Huffman Expander. Decoding Huffman-encoded streams is inherently serial because the starting position of a code cannot be known before decoding the previous code. The Huffman expander performs speculative decoding by issuing decode-table look-ups for a configurable number of starting bit positions, similar to the IBM z15 decompressor [3].

Finite-State Entropy (FSE) Expander

The FSE Expander (consisting of FSE Table Builder, SRAM, and Reader in Figure 7.9) first builds a decode table based on the normalized count statistics of each symbol by reading the input file stream. Then, the FSE expander reads the table to produce the decoded symbol, which is the sum of bits from the input file stream and the base value. The base value, number of bits to read from the input, and the next table entry to read are indicated in the table entry.

LZ77 Encoder

The LZ77 encoder (consisting of the LZ77 Hash Matcher and LitLen Injector blocks in Figure 7.10) performs streaming dictionary encoding of raw input data and produces output in the common `(offset, length, literal)` format. It primarily consists of a configurable hash table SRAM and a history window SRAM. This unit iterates over the data, checking the hash table for matches in the history and then checking the history buffer to find the extent of the match. If no match is available, the data is emitted as a literal.

Huffman Compressor

The Huffman compressor consists of two main modules, the Huffman dictionary builder and the Huffman encoder (Figure 7.10). The dictionary builder collects symbol statistics and writes the dictionary into memory. The encoder performs compression by performing look-ups into the dictionary builder.

Finite-State Entropy (FSE) Compressor

The FSE compressor is shown as part of Figure 7.10 and consists of three separate dictionary builders for each of literal length, match length, and offset and an FSE encoder that performs dictionary lookups to perform compression. The input stream is passed to the combinational SeqToCodeConverter which feeds the dictionary builders with the correct inputs while the encoder consumes the raw input stream.

Parameterization

Our framework supports two parameterization methods:

1. Runtime configurable (`RunT`): These are parameters that can be changed after hardware is built, either for programmability or for rapid design space exploration.
2. Compile-time configurable (`CompileT`): These are traditional hardware parameters that are fixed when the design is compiled.

The parameters available in our framework include:

CDPU-wide parameters:

1. Accelerator placement (`CompileT`), including:
    a) Near-core RoCC/on-NoC; no latency injection
    b) Chiplet; 25ns latency injection
    c) PCIeLocalCache: PCIe+DDIO, assuming PCIe card has large SRAM cache and on-board DRAM; 200ns latency injection (measurements from [158]) for raw input + final output, no latency injection for intermediate reads/writes
    d) PCIeNoCache: PCIe+DDIO, assuming PCIe card does not have on-board cache/DRAM; 200ns latency injection for all requests
2. Algorithm support (`RunT` & `CompileT`)

LZ77 decoder parameters:

3. History Window Size (`RunT` & `CompileT`)

LZ77 encoder parameters:

4. History Window Size (`RunT` & `CompileT`)
5. Hash-table number of entries (`RunT` & `CompileT`)
6. Hash-table associativity (`RunT` & `CompileT`)
7. Hash-table contents (`CompileT`)
8. Hash Function (`CompileT`)

Huffman expander parameters:

9. Number of speculations (`CompileT`)

Huffman compressor parameters:

10. Number of Bytes per cycle to collect symbol stats (`CompileT`)

FSE compressor parameters:

11. Number of Bytes per cycle to collect symbol stats (`CompileT`)
12. Max accuracy of FSE compression tables (`CompileT`)

7.6 CDPU design space exploration

Evaluation Methodology

We perform design-space exploration (DSE) of our accelerated systems implemented in RTL running HyperCompressBench using FireSim [118], which provides high-performance, deterministic, and cycle-exact[4] modeling of designs, while providing cycle-accurate abstract models for I/O, including DRAM [34].

Each benchmark is run on two systems: a single-core RISC-V system with CDPUs attached, modeled at 2 GHz core/CDPU frequency, and one core (2 HT) of a Xeon E5-2686 v4-based server, running at 2.3 GHz base/2.7 GHz turbo.

Performance results for our accelerated systems are reported by measuring end-to-end operation time from the perspective of software (i.e. the time taken by an entire compression or decompression call, without overlapping requests). Performance results for the Xeon are collected using `lzbench` [194], a standard tool for in-memory (de)compression algorithm benchmarking. In HyperCompressBench, a suite's aggregate performance metric is the total amount of time required to

[4]All components of the RISC-V SoC written in RTL, including our accelerator design, are modeled bit-by-bit and cycle-by-cycle exactly as they would perform in silicon taped-out using the same RTL.

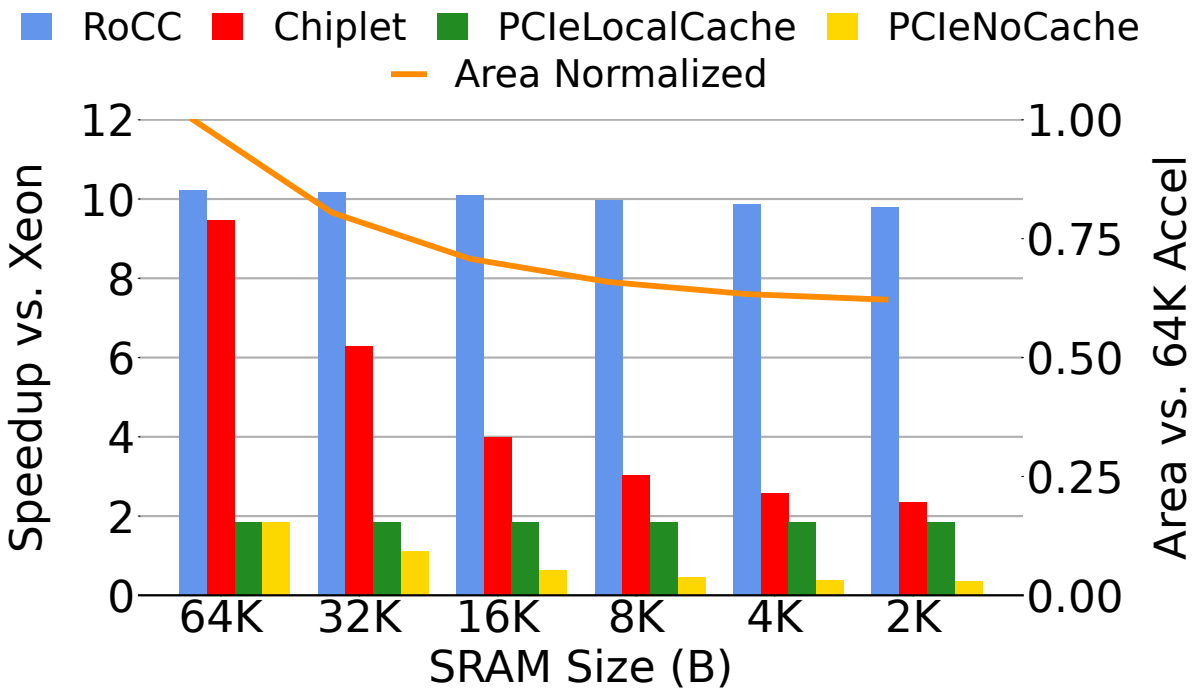


Figure 7.11: CDPU speedup running Snappy Decompression on HyperCompressBench across accelerator placements and History SRAM Sizes. Area is normalized vs. the 64KB history SRAM accelerator.

(de)compress each benchmark file in the suite. Lastly, we report ASIC area estimates by pushing designs through synthesis [214] for a commercial 16nm-class process.

Snappy Decompressor

Figure 7.11 shows speedup and area results from a CDPU generated for Snappy Decompression, configured with a range of on-accelerator history windows (given on the x-axis) and in a variety of placements in the system. In this design, offsets beyond the on-accelerator SRAM fall back to the L2 cache. We see that the CDPU placed near-core (RoCC) with the largest on-accelerator window size (equal to Snappy's SW maximum of 64 KB), achieves the highest speedup; it is over 10× faster than the Xeon (11.4 GB/s accelerated vs. 1.1 GB/s Xeon), while consuming 0.431mm^2 of silicon area in 16nm. As a comparison, this is less than 2.4% of the area of a single modern Xeon Core Tile (17.98mm^2 in 14nm, reported in [220]). If we instead shrink the on-CDPU history to 2 KB, we find a potentially more fruitful design point: we can achieve a 38% reduction in area for only a 4.3% reduction in speedup (i.e., 9.8× speedup vs. Xeon while consuming 1.5% of the area).

As discussed in Chapter 7.5, we also model integrating the CDPU over PCIe+DDIO and re-run the sweep of on-accelerator SRAM size, which is shown by the "PCIeNoCache" series in

Figure 7.11. Even with a 64K SRAM (no off-accelerator history lookups), we see that even the cost of loading/writing input/output data once over PCIe results in a significant (5.6×) slowdown vs. the near-core CDPU, due to the large number of small decompressions in the fleet (Fig. 7.3c).

The increased latency of PCIe also means that the accelerator cannot take advantage of the same performance vs. history SRAM-size tradeoff as the near-core accelerator: the PCIe-attached 32K SRAM design loses most of the performance advantage of the already degraded PCIe 64K design, and performance only degrades further from there. The "PCIeLocalCache" series in Figure 7.11 somewhat mitigates this by modeling a shared on-die SRAM cache and local DRAM attached to the PCIe card. In this situation, we can see that the SRAM optimization continues to work, albeit with an identical starting speedup (at the 64K size) as "PCIeNoCache".

Chiplet integration techniques and new protocols like CXL, UCIe, CCIX, and CAPI offer a new "intermediate" placement option for accelerators; accelerators can be manufactured in a separate die, reducing integration cost, while still remaining on the same package as the core. As discussed in Chapter 7.5, we can model this integration technique in our framework. The results of running the Snappy decompressor in this placement are shown in the "Chiplet" series in Figure 7.11. Considering the configuration with 64K history size, we can see that Chiplet integration is an attractive solution for a Snappy accelerator; it still achieves a 9.5× speedup vs. the Xeon, despite the added latency. However, we can see that performance suffers as more requests are forced to cross the Chiplet interconnect; at the smallest history window sizes, speedups drop such that they are on par with PCIe-based integration.

Snappy Compressor

Figure 7.12 shows speedup, compression ratio, and area results for the Snappy Compress accelerator, covering a range of on-accelerator history windows (on the x-axis). Area results are normalized to the largest version of the accelerator, which has a 64K History SRAM and 2^{14} hash table entries ("64K14HT" on plots). This design consumes 0.851 mm^2 in a 16nm process or about 4.7% the area of a Xeon Core [220]. Reducing the history SRAM size restricts the maximum matching offset that can be identified, and large offset matching does not fall back to the L2 cache since history checking is necessarily serial in compression. Interestingly, the 64 KB SRAM design achieves a 1.1% *higher* compression ratio than Snappy SW. This is because the software implements a skipping mechanism that avoids hash-table lookups when data appears incompressible to save cycles. In a hardware implementation, this optimization is not useful. Therefore, the accelerator has more "chances" to find a match than SW. As the SRAM size is reduced, we do see a drop-off in the achieved compression ratio as compared to software, ranging from an 8% loss at 2 KB (with 20% area savings) to a 0.5% loss at 32 KB (with 10% area savings).

We also see that across the swath of history window sizes, the accelerator achieves significant speedup compared to the Xeon. For example, the 64 KB configuration achieves over 16× speedup compared to the Xeon (5.84 GB/s accel. vs. 0.36 GB/s Xeon). The various smaller configurations achieve between 14.8× and 15.5× speedup, losing performance only because of the increased amount of data they must write due to the lower achieved compression ratio.

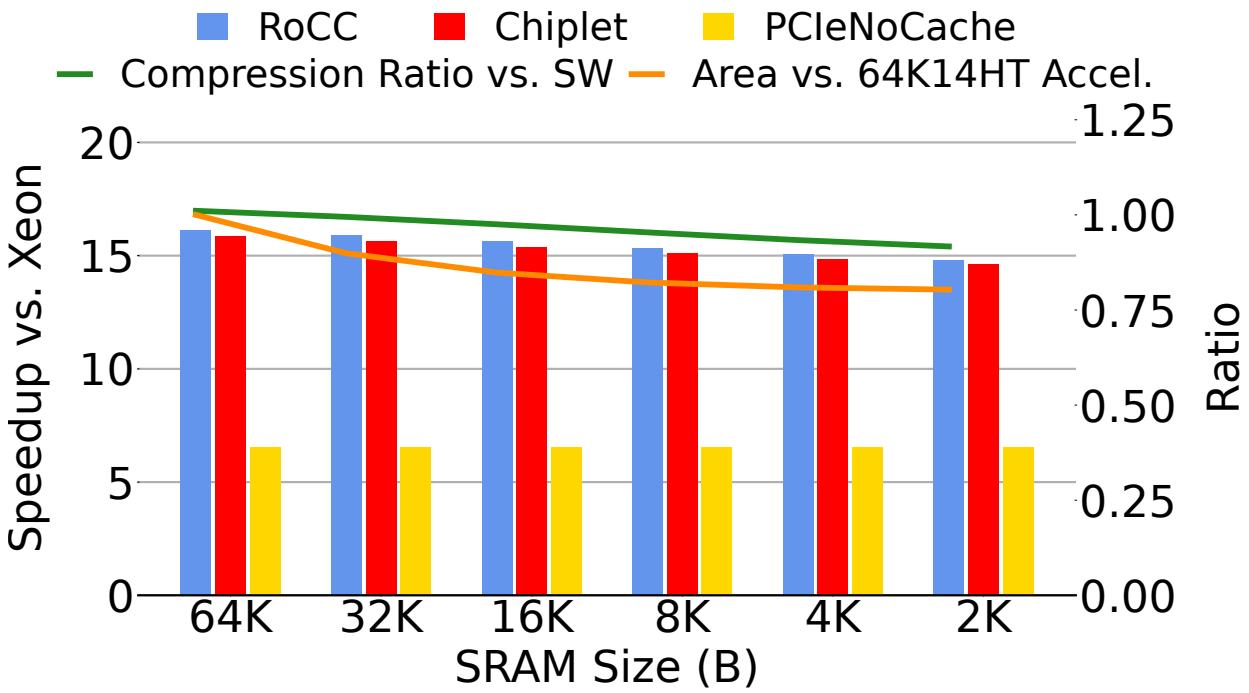


Figure 7.12: CDPU speedup/area running Snappy Compression on HyperCompressBench across CDPU placements and History SRAM Sizes. Area is norm-ed vs. the 64K history SRAM and 2^{14} hash table entry Snappy CDPU.

Figure 7.12 also shows various compression accelerator placements. We see again that a Chiplet-integrated design performs very well, achieving less than 1.7% loss of speedup vs. the near core design across the swath of SRAM sizes. PCIe again struggles, but fares much better than in the decompression case, with speedups shrinking to around 6.6×. Note that PCIeNoCache and PCIeLocalCache are identical for compression, given that there are no intermediate data accesses.

Given that Snappy is a lightweight algorithm, we can ask an interesting question: how small of a Snappy accelerator can we build while still achieving meaningful compression and high performance? In Figure 7.12 we can see that reducing the history window size to 2K for compression can result in negligible loss of speedup and a small, but potentially tolerable 8% loss in compression ratio, while reducing accelerator area by 20%. Figure 7.13 shows the results of tuning another design knob: the number of hash table entries. Reducing the number of entries increases the likelihood of collisions and reduces the chance of finding optimal matches in the history window. However, we can see that reducing the number of hash table entries can provide drastic area wins: a snappy compression accelerator with 2^9 hash table entries and a 2K history SRAM consumes only 34% of the area of the full-size design (and only 1.6% of the area of a Xeon Core), with a negligible loss of speedup and while only increasing compression ratio loss by 3% compared to the 2K history, 2^{14} hash table entry design.

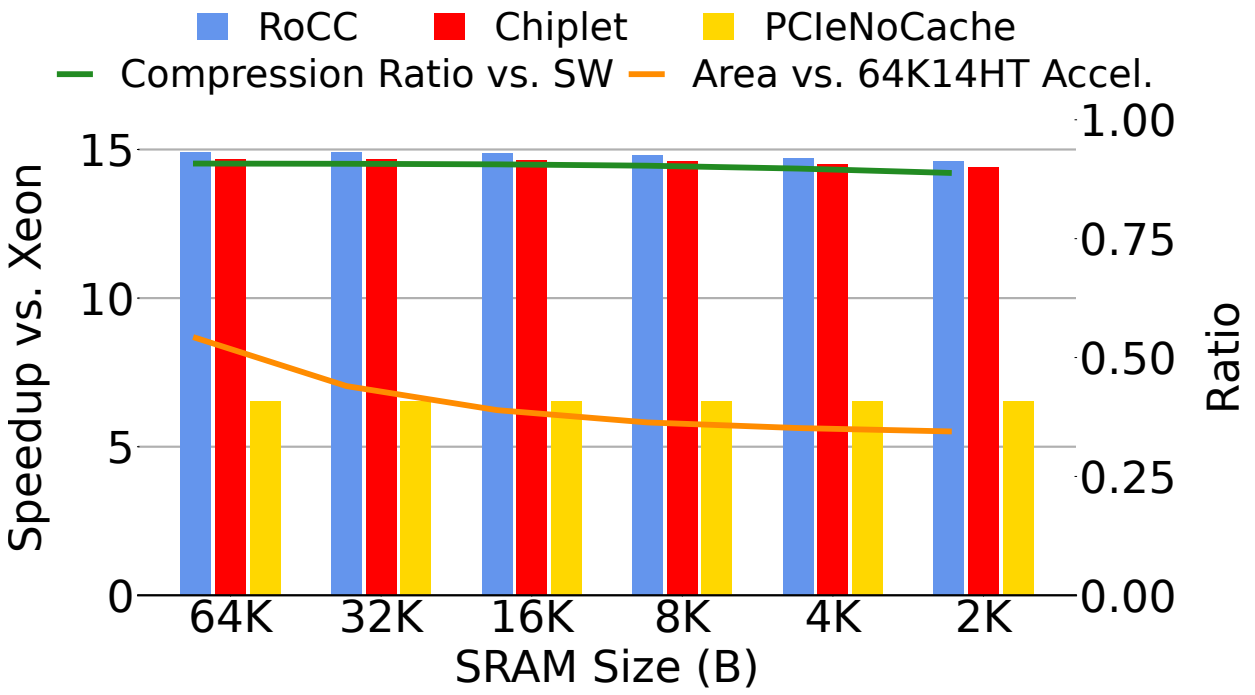


Figure 7.13: CDPU speedup/area running Snappy Compression on HyperCompressBench across CDPU placements and History SRAM Sizes, with only 2^9 Hash Table Entries. Area is norm-ed vs. the 64K history SRAM and 2^{14} hash table entry Snappy CDPU.

ZStd Decompressor

Figure 7.14 shows speedup and area results from a CDPU generated for ZStd Decompression, configured with a range of on-accelerator history windows (given on the x-axis) and in a variety of placements in the system. The largest design in this plot (64K SRAM) achieves 4.2× speedup vs. the Xeon (3.95 GB/s accelerated vs. 0.94 GB/s Xeon).

We can see that overall performance is reduced compared to the Snappy accelerator. While this is not directly comparable since the Snappy/ZStd suites in HyperCompressBench are different, we can broadly see the cost of the additional entropy decoding steps on the accelerator's performance, especially since the LZ77 decoding block is re-used between Snappy and ZStd accelerators. This added cost attenuates both the area savings and performance impact of reducing history SRAM compared to the Snappy decompressor; the overall savings moving from the 64K SRAM design (1.9 mm^2 in 16nm) to the 2K SRAM design of the ZStd compressor is only 8.6%.

An additional parameter that can be swept in the ZStd decompressor as compared to Snappy is the amount of speculation allowed in the Huffman Decoder. All results in Figure 14 used a speculation of 16. To better understand the design space, we explored two additional speculation design points: 32 (similar to IBM z15) and 4 (as a minimum reasonable design point), while keep-

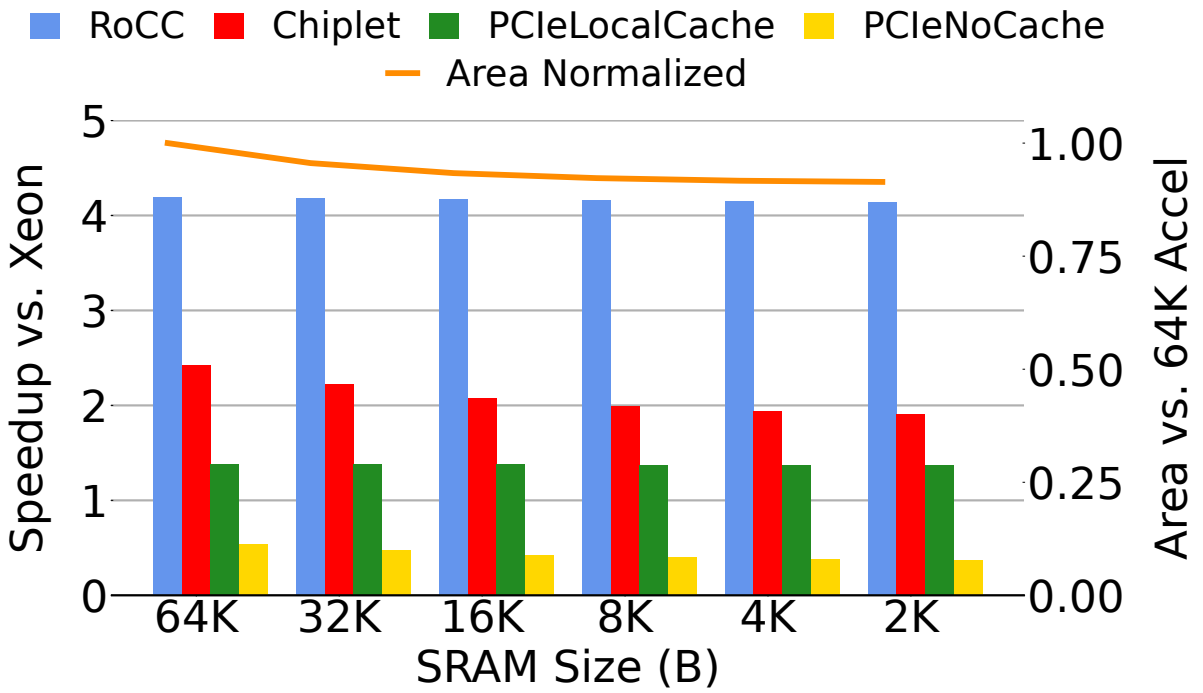


Figure 7.14: CDPU speedup running ZStd Decompression on HyperCompressBench across accelerator placements and History SRAM Sizes. Area is normalized vs. the 64KB history SRAM accelerator.

ing history SRAM size fixed at 64K. The 32-speculation design increases speedup over Xeon to 5.64×, while requiring an additional 18% area as compared to the 16-speculation design. The 4-speculation design reduces speedup over Xeon to 2.11×, while requiring 10% less area as compared to the 16-speculation design. As we can see, for the ZStd decompressor, tuning the speculation amount produces a much larger swing in design quality-of-result than history SRAM size.

ZStd Compressor

Figure 7.15 shows speedup, compression ratio, and area results for the ZStd Compress accelerator, covering a range of on-accelerator history windows (on the x-axis). Area results are normalized to the largest version of the accelerator, which has a 64K History SRAM and 2^{14} hash table entries ("64K14HT" on plots). This design consumes 3.48mm^2 in a 16nm process. As this accelerator re-uses the LZ77 encoder block from the Snappy accelerator, restricting history SRAM size similarly restricts the maximum matching offset that can be identified. Looking first at compression ratio, we see that the accelerator achieves only 84% of the compression ratio of software, likely primarily due to the fact that we are re-using the LZ77 encoder block as configured for Snappy. We leave exploring more complicated LZ77 encoding techniques to future work. With the caveat

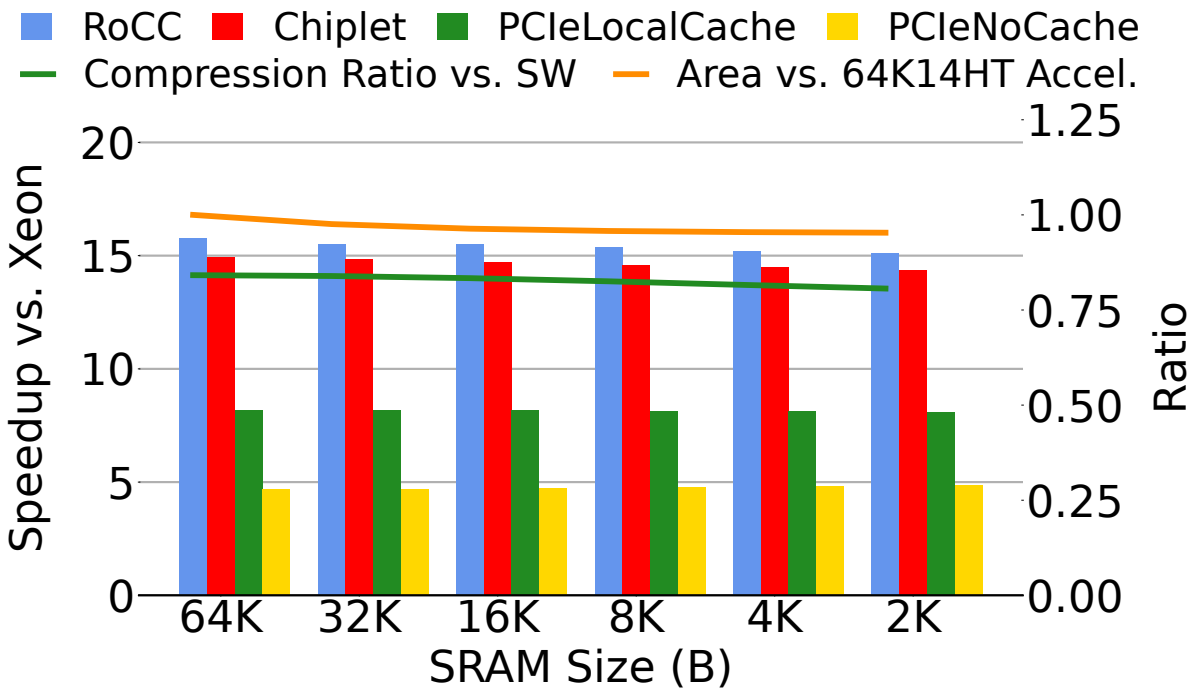


Figure 7.15: CDPU speedup/area running ZStd Compression on HyperCompressBench across CDPU placements and History SRAM Sizes. Area is norm-ed vs. the 64K hist. and 2^{14} hash table entry ZStd CDPU.

that compression ratio is reduced, the largest configuration of accelerator achieves a 15.8x speedup compared to the Xeon (3.5 GB/s accelerated vs. 0.22 GB/s Xeon).

Key Implementation-Based Design-Space Exploration Lessons for Hyperscale CDPUs

Our design space exploration shows the importance of focusing not only on the microarchitectural design of CDPUs, but also their high-level parameters. By tuning these high-level parameters in the previous section, we observed for example, 46× differences in speedups and 66% savings in silicon area. Here, we summarize our key findings:

1. Decompression accelerator feasibility is very heavily affected by accelerator placement. Given data sizes observed in Google's fleet, near-core accelerators (10× speedup for Snappy, 4× speedup for ZStd) perform over 3 to 5.6 *times* better than PCIe attached accelerators (1.8× speedup for Snappy, 1.4× speedup for ZStd). Chiplets offer a reasonable middle ground for Snappy, with our chiplet-integrated accelerator (9.5× speedup) performing only 1.1× worse than the near-core accelerator.

2. In contrast, compression is less sensitive to accelerator placement; we observe over 6.6× speedup (Snappy) or 8.2× speedup (ZStd) in the PCIe attached cases. However, the biggest performance gains are still seen for near-core and chiplet-integrated designs (around 15 to 16× speedup for both Snappy and ZStd).

3. Snappy decompression accelerator area is dominated by history size, which also affects speedup (but not compression ratio). Given data characteristics in Google's fleet, a 38% silicon area savings can be achieved by slightly sacrificing speedup (9.8× vs. 10× speedup).

4. ZStd decompression accelerator area is dominated by varying the amount of speculation in the Huffman stage. Given data characteristics in Google's fleet, there is a 31% silicon area cost increase between speculation amounts of 4 and 32, but this comes with a significant corresponding improvement in speedup (2.1× vs. 5.6× speedup).

5. Snappy compression accelerator area is dominated by history buffer size and hash table size. When both are reduced, a negligible sacrifice in speedup and a 12% sacrifice in compression ratio can result in reducing accelerator silicon area by 66%.

7.7 Related work

A few prior studies have presented (de)compression metrics as part of broader hyperscaler fleet characterizations [110, 198, 106]. Our study is the first to take a fleet-wide, multi-year deep-dive into (de)compression usage at a major cloud provider by profiling Google's fleet and derives several novel insights for CDPU design. Furthermore, we use the insights gained to build a parameterized generator for CDPUs that supports hyperscale use-cases, and translate our profiling data into open-source, hyperscaler-representative (de)compression benchmarks that can be used by the community.

Many prior studies have explored implementing hardware accelerators for lossless block-level (de)compression, both in academia [176, 47], industrial research [185, 74], and commercial products [7, 186, 171, 101, 172, 210, 3]. However, all of these studies only explore a single point in the design space; they focus on a single algorithm (usually Flate or ZStd), in a single placement (PCIe, NoC-attached, on-chipset, etc.), and sometimes only a single direction (decompress or compress). Furthermore, these studies usually run existing open-source benchmarks, which, as shown in Chapter 7.4, are not representative of hyperscale workloads. To our knowledge, we are the first study to build a highly parameterized CDPU *generator* that supports multiple algorithms using a common set of high-performance, re-usable primitives. Our generator integrates into a RISC-V SoC framework that allows for rapid evaluation of CDPUs across system placements and configuration parameters. Furthermore, we evaluate our generated designs with HyperCompressBench, a (de)compression benchmark that is representative of hyperscale workloads.

However, for our design space evaluations to produce realistic results, it is important to contextualize and validate our observed results with those published in prior studies. To that end, we compare against the current state of the art, the NXU accelerator for the IBM POWER9 and

z15 [3]. While the NXU study does not provide a directly comparable performance result using open-source benchmarks, we can extrapolate from its performance vs. data size plots and the size distribution of input files in HyperCompressBench. This calculation projects performance of the NXU on HyperCompressBench of 5.6 to 7.1 GB/s for compression and 6.7 to 7.7 GB/s for decompression. Our results for compression (5.8 GB/s Snappy, 3.5 GB/s ZStd) and decompression (11.4 GB/s Snappy, 5.3 GB/s ZStd) are comparable, given our RISC-V SoC's weaker memory system and algorithmic differences. In area terms, our academic prototype is similar, but could benefit from greater tuning/engineering effort, with our design consuming around 1.3 mm^2 (Snappy) or 5.7 mm^2 (ZStd) in a 16nm process, while the IBM NXU consumes around 3.5 mm^2 in the GF14 process (extrapolated from [3, 219]).

To our knowledge, the state-of-the-art *open-source* compression and decompression implementation is Project Zipline [172, 210, 48, 53] from Microsoft, which has also been fabricated in the Corsica ASIC [211, 48]. The ASIC version of this design is limited to 25 Gbps for single requests (3.125 GB/s) [48]. Also in contrast, our design is heavily parameterized, supporting several compile-time and run-time configurable parameters, and is integrated into a complete system for evaluation.

FPGAs have also been proposed as a host platform for compression and decompression accelerators constructed using handwritten RTL or high-level synthesis tools [74, 176, 47, 130]. Unfortunately, FPGAs as a basis technology are insufficiently performant to support (de)compression as compared to ASIC designs—our generated accelerators are significantly faster than the state-of-the-art handwritten [47] and HLS-generated [130] FPGA-hosted implementations. In Chapter 7.3, we also demonstrated that the flexibility of FPGAs is unnecessary for the pace of (de)compression algorithm evolution in WSCs. Lastly, the Corsica ASIC's compression engine has also been shown to achieve improved performance over FPGA-hosted solutions [48].

Several interesting industrial products are also on the horizon, including NVIDIA's DPU [162], Intel's IPU [99], and Intel Sapphire Rapids/QAT [100]. At time of writing, little commercial benchmarking data is available publicly for these systems.

7.8 Conclusion

In this work, we presented a detailed fleet-wide characterization of (de)compression usage at a major cloud provider by profiling Google's datacenter fleet. We showed that (de)compression consumes significant fleet CPU cycles, even though services under-utilize the most aggressive forms of compression, presenting an opportunity for hardware acceleration to save resources beyond merely CPU cycles.

We then presented the first end-to-end design/evaluation framework for CDPUs, including: 1. An open-source RTL-based CDPU generator that supports many run-time and compile-time parameters. 2. Integration into an open-source RISC-V SoC for rapid performance and silicon area evaluation with varying CDPU placements and configurations. 3. An open-source (de)compression benchmark, HyperCompressBench, that represents (de)compression usage in Google's fleet.

While a large body of prior work has improved the microarchitectural state-of-the-art for CDPUs supporting various algorithms in fixed contexts [176, 47, 185, 74, 7, 186, 171, 101, 172, 210, 3, 48, 211, 53, 130, 162, 99, 100], we found that higher-level design parameters like accelerator placement, hash table sizing, history window sizes, and more are as critical when considering the feasibility of CDPU integration, but were previously not well-studied in the literature.

Using our CDPU design framework, we performed an extensive design space exploration running HyperCompressBench. Our design-space exploration spanned a 46× range in accelerator speedup, 3× range in silicon area (for a single pipeline), and explored a variety of accelerator integration techniques to better understand optimal CDPU designs for hyperscale contexts. Our final hyperscale-optimized accelerator instances are up to 10× to 16× faster than a single Xeon core, while consuming a small fraction (as little as 2.4% to 4.7%) of the area.

7.9 Retrospective

This work has influenced product design at Google and various silicon vendors. This work underwent the ISCA conference's artifact evaluation process and received all available badges, including results being reproduced.

Chapter 8

Conclusion

In this dissertation, we bridged the gap between agile hardware-software co-design methodology and the world's largest compute platforms. With FireSim and Chipyard, we enabled small teams to rapidly co-design hardware and software for these hyperscale cloud systems (Figure 8.1, Theme 1) and demonstrated the potential of these methodologies by architecting and building a cloud-optimized server system-on-chip, Hyperscale SoC (Figure 8.1, Theme 2).

As there is no sign of the exponential demand for cloud compute slowing, there remains much work to do in improving the performance, efficiency, sustainability, and total-cost-of-ownership of these massive compute platforms. Below, we summarize important next-steps and trends that must

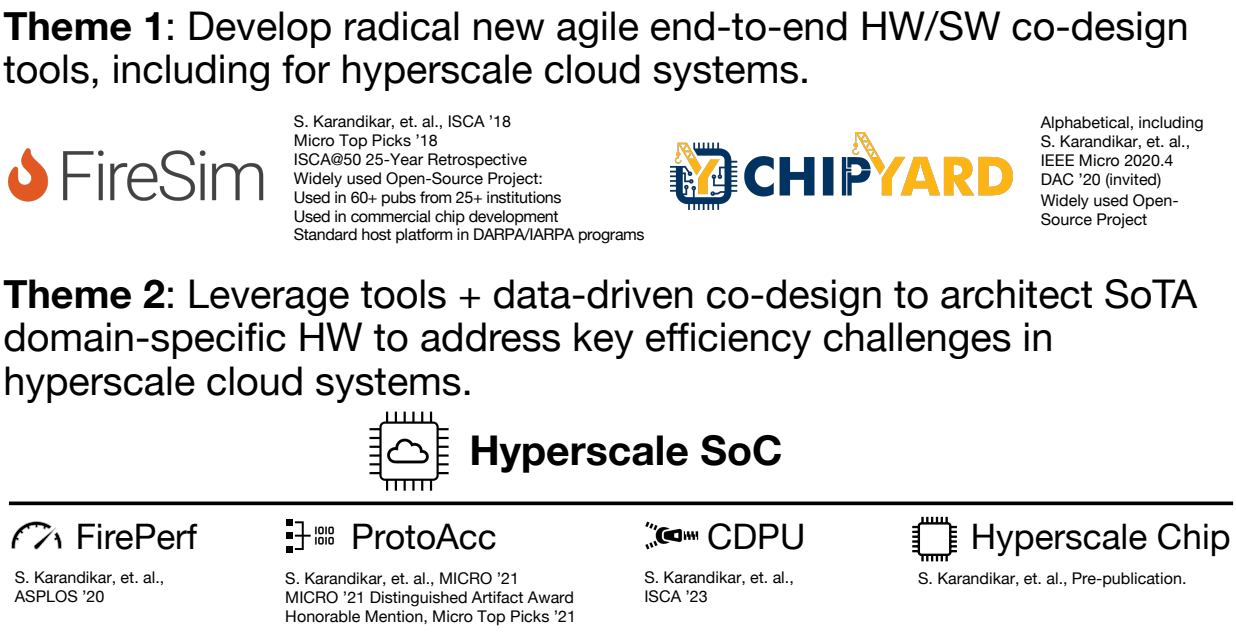


Figure 8.1: A review of overarching themes of this dissertation.

be accounted for when designing next-generation hyperscale systems, looking from the perspective of the key themes highlighted in this dissertation.

8.1 Open problems in agile design methodologies for hyperscale systems

Unsurprisingly, technology trends are continuing to push the scale of individual server designs as well as machine-learning training and inference platforms. While device scaling continues, albeit at a slower pace than in the past, the new name-of-the-game is the construction of massive system-in-package (SiP) designs, building on this dissertation's focus on systems-on-chip (SoCs). These SiP designs consist of many large compute, memory, and I/O dies packaged on a substrate that provides high-bandwidth and low-latency communication between them. This opens up numerous scalability challenges for both the generation, simulation, verification, and tape-out of these large systems, especially when working in small, agile teams. At a macro scale, inter-system network bandwidths and latencies continue to improve both for general-purpose server interconnection and the construction of machine-learning "supercomputers", increasing the challenge of modeling and co-designing large scale-out systems. From a methodological perspective, machine learning is a relatively new tool in our toolbox that can be used to further improve the agility of these flows. ML can already perform well on tasks such as assisting humans with RTL implementation, but continued efforts are needed to build a unified ecosystem that can help realize the larger-scale potential of machine learning for hardware design in agile teams.

8.2 Open problems in specialization for hyperscale cloud datacenters

The hyperscale system-on-chip design proposed and implemented in this dissertation is just the start. As a first step, while we enabled the acceleration of critical datacenter tax operations in our server design, it is important now to take a step back and focus on the integration of large collections of these accelerator designs into accelerator complexes. These accelerator complexes need to enable, for example, the chaining of datacenter tax accelerators into larger operations such as remote procedure call and storage processing without CPU intervention. When designing these complexes, there are important questions about partitioning functionality, understanding system balance requirements, and maintaining flexibility and programmability that must be addressed. Given the shift towards SiP designs discussed earlier, a new question also arises: how do these kinds of accelerators fit into the network-on-package topology? As we did for individual datacenter taxes in this dissertation, new fleet-wide profiling mechanisms need to be developed and deployed to better understand the interactions between frequently exercised code that may be amenable to acceleration and the rest of the systems software stack. Similarly, as system scale increases, it

becomes critical to not only build representative benchmarks for individual datacenter taxes, but to build and share workload mixes that represent everything happening on a server at a given time.

From a higher-level perspective, researchers need to continue to understand the limits of specialization in hyperscale contexts. While the datacenter taxes represent key "heavy-hitter" operations that are good candidates for acceleration, continuing to think in this fashion quickly leads to a long tail of acceleration candidates that are unlikely to bear fruit. Using machine-learning platforms as a key example, focusing on more vertically optimized approaches to acceleration and specialization may be fruitful, but again require careful analysis to understand whether hardware development timelines can keep pace with the demands on the evolution of service functionality.

8.3 Parting words

Given hardware industry trends, these two themes/open-problems will continue to go hand-in-hand: Greater specialization will motivate agility in hardware design flows and greater agility in hardware design flows will enable easier specialization. Whoever can deploy the *most effective* specialized systems the *fastest* will be able to drive the greatest societal impact.

Bibliography

[1] https://github.com/sifive/block-inclusivecache-sifive.

[2] https://github.com/sifive/sifive-blocks.

[3] Bulent Abali et al. "Data Compression Accelerator on IBM POWER9 and z15 Processors : Industrial Product". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 1–14. DOI: 10.1109/ISCA45697.2020.00012.

[4] Andreas Abel et al. "A Profiling-Based Benchmark Suite for Warehouse-Scale Computers". In: *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2024, pp. 325–327. DOI: 10.1109/ISPASS61541.2024.00046.

[5] Heather Adkins et al. *Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems*. O'Reilly Media, 2020. ISBN: 9781492083122.

[6] B. Agrawal et al. "Addressing the Challenges of Synchronization/Communication and Debugging Support in Hardware/Software Cosimulation". In: *21st International Conference on VLSI Design (VLSID 2008)*. Jan. 2008, pp. 354–361. DOI: 10.1109/VLSI.2008.74.

[7] *AHA374 / AHA378 PCI Express Compression and Decompression Accelerator Card*. http://www.aha.com/Uploads/aha374-378_brief_rev_c1.pdf. 2022.

[8] Jyrki Alakuijala and Zoltan Szabadka. *Brotli Compressed Data Format*. RFC 7932. July 2016. DOI: 10.17487/RFC7932. URL: https://www.rfc-editor.org/info/rfc7932.

[9] Jyrki Alakuijala et al. *Comparison of Brotli, Deflate, Zopfli, LZMA, LZHAM and Bzip2 Compression Algorithms*. https://cran.r-project.org/web/packages/brotli/vignettes/brotli-2015-09-22.pdf. 2015.

[10] Alibaba. *Instance generations and type families*. https://www.alibabacloud.com/help/doc-detail/25378.htm. 2017.

[11] Amazon Web Services. *Amazon EC2 F1 Instances*. https://aws.amazon.com/ec2/instance-types/f1/. 2017.

[12] Amazon Web Services. *AWS Cloud Credits for Research*. https://aws.amazon.com/research-credits/. 2017.

[13] Amazon Web Services. *Official repository of the AWS EC2 FPGA Hardware and Software Development Kit - GitHub*. https://github.com/aws/aws-fpga. 2017.

[14] Alon Amid et al. “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs”. In: *IEEE Micro* 40.4 (July 2020), pp. 10–21. ISSN: 0272-1732. DOI: 10.1109/MM.2020.2996616. URL: https://doi.org/10.1109/MM.2020.2996616.

[15] Alon Amid et al. “Vertically Integrated Computing Labs Using Open-Source Hardware Generators and Cloud-Hosted FPGAs”. In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2021, pp. 1–5. DOI: 10.1109/ISCAS51556.2021.9401515.

[16] Hari Angepat et al. “FPGA-Accelerated Simulation of Computer Systems”. In: *Synthesis Lectures on Computer Architecture* 9.2 (2014), pp. 1–80. DOI: 10.2200/S00586ED1V01Y201407CAC029. eprint: https://doi.org/10.2200/S00586ED1V01Y201407CAC029.

[17] *Apache Thrift*. https://thrift.apache.org/.

[18] Michael Armbrust et al. *Above the Clouds: A Berkeley View of Cloud Computing*. Tech. rep. UCB/EECS-2009-28. EECS Department, University of California, Berkeley, Feb. 2009.

[19] R. Arnold and T. Bell. “A corpus for the evaluation of lossless compression algorithms”. In: *Proceedings DCC ’97. Data Compression Conference*. 1997, pp. 201–210. DOI: 10.1109/DCC.1997.582019.

[20] Krste Asanović. “FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers”. In: FAST 2014.

[21] Krste Asanović and David A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Tech. rep. UCB/EECS-2014-146. EECS Department, University of California, Berkeley, Aug. 2014.

[22] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016.

[23] *AWS Nitro System*. https://aws.amazon.com/ec2/nitro/.

[24] Grant Ayers et al. “AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers”. In: *Proceedings of the 46th International Symposium on Computer Architecture*. ISCA ’19. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 462–473. ISBN: 9781450366694. DOI: 10.1145/3307650.3322234. URL: https://doi.org/10.1145/3307650.3322234.

[25] Jonathan Bachrach et al. “Chisel: Constructing hardware in a Scala embedded language”. In: *DAC Design Automation Conference 2012*. 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[26] Jonathan Balkind et al. “OpenPiton: An Open-Source Manycore Research Framework”. In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’16. Atlanta, Georgia, USA: ACM, 2016, pp. 217–232. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872414.

[27] Jeff Barr. *EC2 F1 Instances with FPGAs*. https://aws.amazon.com/blogs/aws/ec2-f1-instances-with-fpgas-now-generally-available/. 2017.

[28] Jeff Barr. *New C5n Instances with 100 Gbps Networking*. https://aws.amazon.com/blogs/aws/new-c5n-instances-with-100-gbps-networking/. 2018.

[29] Luiz Barroso et al. "Attack of the Killer Microseconds". In: *Commun. ACM* 60.4 (Mar. 2017), pp. 48–54. ISSN: 0001-0782. DOI: 10.1145/3015146. URL: http://doi.acm.org/10.1145/3015146.

[30] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition". In: *Synthesis Lectures on Computer Architecture* 8.3 (2013), pp. 1–154. DOI: 10.2200/S00516ED2V01Y201306CAC024. eprint: https://doi.org/10.2200/S00516ED2V01Y201306CAC024.

[31] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. "The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition". In: *Synthesis Lectures on Computer Architecture* 13.3 (2018), pp. i–189. DOI: 10.2200/S00874ED3V01Y201809CAC046. URL: https://doi.org/10.2200/S00874ED3V01Y201809CAC046.

[32] Timothy C. Bell, John G. Cleary, and I. H. Witten. *Text compression*. eng. Prentice Hall advanced reference series. Computer science. Englewood Cliffs, N.J: Prentice Hall, 1990. ISBN: 0139119914.

[33] David Biancolin et al. "Accessible, FPGA Resource-Optimized Simulation of Multiclock Systems in FireSim". In: *IEEE Micro* 41.4 (2021), pp. 58–66. DOI: 10.1109/MM.2021.3085537.

[34] David Biancolin et al. "FASED: FPGA-Accelerated Simulation and Evaluation of DRAM". In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '19. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 330–339. ISBN: 9781450361378. DOI: 10.1145/3289602.3293894. URL: https://doi.org/10.1145/3289602.3293894.

[35] N. L. Binkert et al. "Performance analysis of system overheads in TCP/IP workloads". In: *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. Sept. 2005, pp. 218–228. DOI: 10.1109/PACT.2005.35.

[36] N. L. Binkert et al. "The M5 Simulator: Modeling Networked Systems". In: *IEEE Micro* 26.4 (July 2006), pp. 52–60. ISSN: 1937-4143. DOI: 10.1109/MM.2006.82.

[37] Brendan Gregg. *Flame Graphs*. http://www.brendangregg.com/flamegraphs.html. 2019.

[38] Brendan Gregg. *FlameGraph: Stack trace visualizer*. https://github.com/brendangregg/FlameGraph. 2019.

[39] *Brotli compression format*. https://github.com/google/brotli.

[40] *Building Custom RISC-V SoCs in Chipyard*. https://fires.im/micro19-slides-pdf/03_building_custom_socs.pdf.

[41] D. C. Burger and D. A. Wood. "Accuracy vs. performance in parallel simulation of interconnection networks". In: *Proceedings of 9th International Parallel Processing Symposium*. Apr. 1995, pp. 22–31. DOI: 10.1109/IPPS.1995.395909.

[42] *C++ Arena Allocation Guide — Protocol Buffers — Google Developers*. https://developers.google.com/protocol-buffers/docs/reference/arenas.

[43] *Cap'n Proto*. https://capnproto.org/.

[44] Adrian Caulfield et al. "A Cloud-Scale Acceleration Architecture". In: *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Oct. 2016.

[45] Christopher Celio and UCB-BAR Contributors. *The Sodor Processor Collection*. https://github.com/ucb-bar/riscv-sodor. 2024.

[46] Christopher Celio, David A. Patterson, and Krste Asanović. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*. Tech. rep. UCB/EECS-2015-167. EECS Department, University of California, Berkeley, June 2015.

[47] Jianyu Chen, Maurice Daverveldt, and Zaid Al-Ars. "FPGA Acceleration of Zstd Compression Algorithm". In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2021, pp. 188–191. DOI: 10.1109/IPDPSW52791.2021.00035.

[48] Derek Chiou, Eric Chung, and Susan Carrie. *HotChips31 Tutorial: (Cloud) Acceleration at Microsoft*. https://old.hotchips.org/hc31/HC31_T2_Microsoft_CarrieChiouChung.pdf. 2019.

[49] Derek Chiou et al. "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators". In: *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 249–261. ISBN: 0-7695-3047-8. DOI: 10.1109/MICRO.2007.36. URL: http://dx.doi.org/10.1109/MICRO.2007.36.

[50] Eric S. Chung et al. "ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs". In: *ACM Trans. Reconfigurable Technol. Syst.* 2.2 (June 2009), 15:1–15:32. ISSN: 1936-7406. DOI: 10.1145/1534916.1534925. URL: http://doi.acm.org/10.1145/1534916.1534925.

[51] Yann Collet and Murray Kucherawy. *Zstandard Compression and the 'application/zstd' Media Type*. RFC 8878. Feb. 2021. DOI: 10.17487/RFC8878. URL: https://www.rfc-editor.org/info/rfc8878.

[52] Henry Cook, Wesley Terpstra, and Yunsup Lee. "Diplomatic Design Patterns: A TileLink Case Study". In: *First Workshop on Computer Architecture Research with RISC-V*. 2017.

[53] Microsoft Corporation and Broadcom Corporation. *Project Zipline Top Micro Architecture Specification*. https://github.com/opencomputeproject/Project-Zipline/blob/master/specs/Project_Zipline_Top_Micro_Architecture_Specification.docx. 2019.

[54] Daniel Dabbelt et al. "Vector Processors for Energy-Efficient Embedded Systems". In: *Proceedings of the Third ACM International Workshop on Many-Core Embedded Systems*. MES '16. Seoul, Republic of Korea: Association for Computing Machinery, 2016, pp. 10–16. ISBN: 9781450342629. DOI: 10.1145/2934495.2934497. URL: https://doi.org/10.1145/2934495.2934497.

[55] Our World in Data. *Computation used to train notable artificial intelligence systems, by domain*. https://ourworldindata.org/grapher/artificial-intelligence-training-computation. 2024.

[56] Arnaldo Carvalho De Melo. "The New Linux perf Tools". In: *Slides from Linux Kongress*. Vol. 18. 2010.

[57] S. De Pestel et al. "RPPM: Rapid Performance Prediction of Multithreaded Workloads on Multicore Processors". In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2019, pp. 257–267. DOI: 10.1109/ISPASS.2019.00038.

[58] Sebastian Deorowicz. *Silesia compression corpus*. https://sun.aei.polsl.pl//~sdeor/index.php?page=silesia.

[59] L. Peter Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. RFC 1951. May 1996. DOI: 10.17487/RFC1951. URL: https://www.rfc-editor.org/info/rfc1951.

[60] Jarek Duda et al. "The use of asymmetric numeral systems as an accurate replacement for Huffman coding". In: *2015 Picture Coding Symposium (PCS)*. 2015, pp. 65–69. DOI: 10.1109/PCS.2015.7170048.

[61] DWARF Debugging Information Format Committee. *DWARF Debugging Information Format Version 5*. Standard. Feb. 2017. URL: http://www.dwarfstd.org/doc/DWARF5.pdf.

[62] Lieven Eeckhout. "Computer Architecture Performance Evaluation Methods". In: *Synthesis Lectures on Computer Architecture* 5.1 (2010), pp. 1–145. DOI: 10.2200/S00273ED1V01Y201006CAC010. eprint: https://doi.org/10.2200/S00273ED1V01Y201006CAC010. URL: https://doi.org/10.2200/S00273ED1V01Y201006CAC010.

[63] *Encoding — Protocol Buffers — Google Developers*. https://developers.google.com/protocol-buffers/docs/encoding#signed_integers.

[64] ESnet/LBNL. *iPerf - The ultimate speed test tool for TCP, UDP and SCTP*. https://iperf.fr/. 2019.

[65] *Extensible Markup Language (XML)*. https://www.w3.org/XML/.

[66] Facebook. *Disaggregated Rack*. http://www.opencompute.org/wp/wp-content/uploads/2013/01/OCP_Summit_IV_Disaggregation_Jason_Taylor.pdf. 2013.

[67] M. Ferdman et al. "A Case for Specialized Processors for Scale-Out Workloads". In: *IEEE Micro* 34.3 (May 2014), pp. 31–42. ISSN: 0272-1732. DOI: 10.1109/MM.2014.41.

[68] *FiniteStateEntropy: New Generation Entropy coders*. https://github.com/Cyan4973/FiniteStateEntropy.

[69] *FireSim: Easy-to-use, Scalable, FPGA-accelerated Cycle-accurate Hardware Simulation in the Cloud*. https://github.com/firesim/firesim. 2019.

[70] Daniel Firestone et al. "Azure Accelerated Networking: SmartNICs in the Public Cloud". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 51–66. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/firestone.

[71] *Flatbuffers*. https://google.github.io/flatbuffers/.

[72] *FlexBuffers*. https://google.github.io/flatbuffers/flexbuffers.html.

[73] Jeremy Fowers et al. "A Configurable Cloud-Scale DNN Processor for Real-Time AI". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 1–14. DOI: 10.1109/ISCA.2018.00012.

[74] Jeremy Fowers et al. "A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs". In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 2015, pp. 52–59. DOI: 10.1109/FCCM.2015.46.

[75] Yu Gan et al. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 3–18. ISBN: 9781450362405. DOI: 10.1145/3297858.3304013. URL: https://doi.org/10.1145/3297858.3304013.

[76] Peter X. Gao et al. "Network Requirements for Resource Disaggregation". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 249–264. ISBN: 978-1-931971-33-1.

[77] *Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach $679 Billion in 2024*. https://www.gartner.com/en/newsroom/press-releases/11-13-2023-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-679-billion-in-20240. 2023.

[78] Hasan Genc et al. "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration". In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 769–774. DOI: 10.1109/DAC18074.2021.9586216.

[79] *Gipfeli, a high-speed compression library.* https://github.com/google/gipfeli.

[80] Abraham Gonzalez et al. “Profiling Hyperscale Big Data Processing”. In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA ’23. Orlando, FL, USA: Association for Computing Machinery, 2023. ISBN: 9798400700958. DOI: 10.1145/3579371.3589082. URL: https://doi.org/10.1145/3579371.3589082.

[81] Google. *HyperCompressBench*. https://github.com/google/HyperCompressBench. 2023.

[82] Google. *HyperProtoBench*. https://github.com/google/HyperProtoBench. 2021.

[83] Brendan Gregg. “The Flame Graph”. In: *Commun. ACM* 59.6 (May 2016), pp. 48–57. ISSN: 0001-0782. DOI: 10.1145/2909476. URL: http://doi.acm.org/10.1145/2909476.

[84] Juncheng Gu et al. “Efficient Memory Disaggregation with Infiniswap”. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 649–667. ISBN: 978-1-931971-37-9.

[85] Gagan Gupta et al. “Open-source Hardware: Opportunities and Challenges”. In: *CoRR* abs/1606.01980 (2016). arXiv: 1606.01980.

[86] Anthony Gutierrez et al. “Sources of error in full-system simulation”. In: *ISPASS*. IEEE Computer Society, 2014, pp. 13–22.

[87] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2017. ISBN: 9780128119068. URL: https://books.google.com/books?id=cM8mDwAAQBAJ.

[88] John Hennessy and David Patterson. “A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development”. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 27–29. DOI: 10.1109/ISCA.2018.00011.

[89] Sara Hooker. “The hardware lottery”. In: *Commun. ACM* 64.12 (Nov. 2021), pp. 58–65. ISSN: 0001-0782. DOI: 10.1145/3467017. URL: https://doi.org/10.1145/3467017.

[90] HP Labs. *The Machine*. https://www.labs.hpe.com/the-machine. 2017.

[91] Qijing Huang et al. “Centrifuge: Evaluating full-system HLS-generated heterogenous-accelerator SoCs using FPGA-Acceleration”. In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2019, pp. 1–8. DOI: 10.1109/ICCAD45719.2019.8942048.

[92] Huawei. *High Throughput Computing Data Center Architecture - Thinking of Data Center 3.0*. www.huawei.com/ilink/en/download/HW_349607. 2014.

[93] Huawei. *Huawei Releases the New-Generation Intelligent Cloud Hardware Platform Atlas*. http://e.huawei.com/us/news/global/2017/201709061557. 2017.

[94] David A. Huffman. “A Method for the Construction of Minimum-Redundancy Codes”. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

[95] Stephen Ibanez et al. “The nanoPU: A Nanosecond Network Stack for Datacenters”. In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021, pp. 239–256. ISBN: 978-1-939133-22-9. URL: https://www.usenix.org/conference/osdi21/presentation/ibanez.

[96] IBISWorld. *Internet Traffic Volume*. https://www.ibisworld.com/us/bed/internet-traffic-volume/88089/. 2024.

[97] SiFive Inc. *SiFive TileLink Specification*. https://sifive.cdn.prismic.io/sifive%2Fcab05224-2df1-4af8-adee-8d9cba3378cd_tilelink-spec-1.8.0.pdf. 2019.

[98] Intel. *Intel Rack Scale Design*. https://www-ssl.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html. 2017.

[99] *Intel Infrastructure Processing Unit (Intel IPU)*. https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html. 2023.

[100] *Intel Launches 4th Gen Xeon Scalable ”Sapphire Rapids”*. https://www.phoronix.com/image-viewer.php?id=intel-xeon-sapphire-rapids-max&image=intel_sapphirerapids_8_lrg. 2023.

[101] *Intel QuickAssist Adapter 8950*. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/quickassist-adapter-8950-brief.pdf. 2015.

[102] *Introducing JSON*. https://www.json.org/json-en.html.

[103] A. Izraelevitz et al. “Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2017, pp. 209–216. DOI: 10.1109/ICCAD.2017.8203780.

[104] M. Jahre and L. Eeckhout. “GDP: Using Dataflow Properties to Accurately Estimate Interference Free Performance at Runtime”. In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2018, pp. 296–309. DOI: 10.1109/HPCA.2018.00034.

[105] Jaeyoung Jang et al. “A Specialized Architecture for Object Serialization with Applications to Big Data Analytics”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 322–334. DOI: 10.1109/ISCA45697.2020.00036.

[106] Geonhwa Jeong et al. “Characterization of Data Compression in Datacenters”. In: *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2023.

[107] Theo Jepsen et al. *From Sand to Flour: The Next Leap in Granular Computing with NanoSort*. 2022. arXiv: 2204.12615 [cs.DC]. URL: https://arxiv.org/abs/2204.12615.

[108] Jim Hogan. *Hogan compares Palladium, Veloce, EVE ZeBu, Aldec, Bluespec, Dini*. http://www.deepchip.com/items/0522-04.html. 2013.

[109] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: ACM, 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246.

[110] Svilen Kanev et al. "Profiling a Warehouse-scale Computer". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15. Portland, Oregon: ACM, 2015, pp. 158–169. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2750392. URL: http://doi.acm.org/10.1145/2749469.2750392.

[111] Sagar Karandikar. "QEMU Support for the RISC-V Instruction Set Architecture". In: *2016 KVM Forum*.

[112] Sagar Karandikar, Mia Champion, and Krste Asanović. "Bringing datacenter-scale hardware-software co-design to the cloud with FireSim and Amazon EC2 F1 instances". In: *AWS Compute Blog* 25 (2017).

[113] Sagar Karandikar et al. "A Hardware Accelerator for Protocol Buffers". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 462–478. ISBN: 9781450385572. DOI: 10.1145/3466752.3480051. URL: https://doi.org/10.1145/3466752.3480051.

[114] Sagar Karandikar et al. "CDPU: Co-designing Compression and Decompression Processing Units for Hyperscale Systems". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA '23. Orlando, FL, USA: Association for Computing Machinery, 2023. ISBN: 9798400700958. DOI: 10.1145/3579371.3589074. URL: https://doi.org/10.1145/3579371.3589074.

[115] Sagar Karandikar et al. "FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 715–731. ISBN: 9781450371025. DOI: 10.1145/3373376.3378455. URL: https://doi.org/10.1145/3373376.3378455.

[116] Sagar Karandikar et al. "FireSim: Cycle-accurate rack-scale system simulation using FPGAs in the public cloud". In: *7th RISC-V Workshop*. 2017.

[117] Sagar Karandikar et al. "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud". In: *IEEE Micro* 39.3 (May 2019), pp. 56–65. ISSN: 1937-4143. DOI: 10.1109/MM.2019.2910175.

[118] Sagar Karandikar et al. "FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud". In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA '18. Los Angeles, California: IEEE Press, 2018, pp. 29–42. ISBN: 978-1-5386-5984-7. DOI: 10.1109/ISCA.2018.00014. URL: https://doi.org/10.1109/ISCA.2018.00014.

[119] Sagar Karandikar et al. "RETROSPECTIVE: FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud". In: *ISCA@50 25-Year Retrospective: 1996-2020*. Ed. by José F. Martínez and Lizy K. John. ACM SIGARCH and IEEE TCCA, June 2023. URL: https://sites.coecis.cornell.edu/isca50retrospective/files/2023/06/Karandikar_2018_FireSim.pdf.

[120] Sagar Karandikar et al. "Using FireSim to Enable Agile End-to-End RISC-V Computer Architecture Research". In: *Third Workshop on Computer Architecture Research with RISC-V*. 2019.

[121] K. Katrinis et al. "Rack-scale disaggregated cloud data centers: The dReDBox project vision". In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2016, pp. 690–695.

[122] *Kendryte K210 Announcement*. https://cnrv.io/bi-week-rpts/2018-09-16. 2018.

[123] D. Kim et al. "DESSERT: Debugging RTL Effectively with State Snapshotting for Error Replays across Trillions of Cycles". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2018, pp. 76–764. DOI: 10.1109/FPL.2018.00021.

[124] Donggyu Kim et al. "Evaluation of RISC-V RTL with FPGA-Accelerated Simulation". In: *First Workshop on Computer Architecture Research with RISC-V*. 2017.

[125] Donggyu Kim et al. "Strober: Fast and Accurate Sample-based Energy Simulation for Arbitrary RTL". In: *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA '16. Seoul, Republic of Korea: IEEE Press, 2016, pp. 128–139. ISBN: 978-1-4673-8947-1. DOI: 10.1109/ISCA.2016.21.

[126] Hyong-youb Kim and Scott Rixner. *Performance characterization of the FreeBSD network stack*. Tech. rep. 2005.

[127] Seah Kim et al. "AuRORA: Virtualized Accelerator Orchestration for Multi-Tenant Workloads". In: *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2023, pp. 62–76.

[128] Joe Kiniry. "FireSim in High-Profile Action—FETT: DARPA's First Ever Bug Bounty Program". In: *2023 FireSim and Chipyard User/Developer Workshop*. Vancouver, BC Canada, 2023.

[129] Andres Lagar-Cavilla et al. "Software-Defined Far Memory in Warehouse-Scale Computers". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 317–330. ISBN: 9781450362405. DOI: 10.1145/3297858.3304053. URL: https://doi.org/10.1145/3297858.3304053.

[130] Morgan Ledwon, Bruce F. Cockburn, and Jie Han. "High-Throughput FPGA-Based Hardware Accelerators for Deflate Compression and Decompression Using High-Level Synthesis". In: *IEEE Access* 8 (2020), pp. 62207–62217. DOI: 10.1109/ACCESS.2020.2984191.

[131] Benjamin C. Lee. "Datacenter Design and Management: A Computer Architect's Perspective". In: *Synthesis Lectures on Computer Architecture* 11.1 (2016), pp. 1–121. DOI: 10.2200/S00693ED1V01Y201601CAC037. eprint: https://doi.org/10.2200/S00693ED1V01Y201601CAC037.

[132] Yongjun Lee et al. "A Fully Associative, Tagless DRAM Cache". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15. New York, NY, USA: ACM, 2015, pp. 211–222. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2750383.

[133] Yunsup Lee and Andrew Waterman. "Managing Chip Design Complexity in the Domain-Specific SoC Era". In: *2020 IEEE Symposium on VLSI Circuits*. 2020, pp. 1–2. DOI: 10.1109/VLSICircuits18222.2020.9162812.

[134] Yunsup Lee et al. "An Agile Approach to Building RISC-V Microprocessors". In: *IEEE Micro* 36.2 (2016), pp. 8–20. DOI: 10.1109/MM.2016.11.

[135] Yunsup Lee et al. *Hwacha Preliminary Evaluation Results, Version 3.8.1*. Tech. rep. UCB/EECS-2015-264. Dec. 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-264.html.

[136] Yunsup Lee et al. *The Hwacha Microarchitecture Manual, Version 3.8.1*. Tech. rep. UCB/EECS-2015-263. Dec. 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-263.html.

[137] Yunsup Lee et al. *The Hwacha Vector-Fetch Architecture Manual, Version 3.8.1*. Tech. rep. UCB/EECS-2015-262. Dec. 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-262.html.

[138] Rastislav Lenhardt and Jyrki Alakuijala. "Gipfeli - High Speed Compression Algorithm". In: *Proceedings of the 2012 Data Compression Conference*. DCC '12. USA: IEEE Computer Society, 2012, pp. 109–118. ISBN: 9780769546568. DOI: 10.1109/DCC.2012.19. URL: https://doi.org/10.1109/DCC.2012.19.

[139] Jacob Leverich and Christos Kozyrakis. "Reconciling High Server Utilization and Sub-millisecond Quality-of-service". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 4:1–4:14. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592821.

[140] Kevin Lim et al. "Disaggregated Memory for Expansion and Sharing in Blade Servers". In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA '09. Austin, TX, USA: ACM, 2009, pp. 267–278. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555789.

[141] S. H. Lim et al. "MDCSim: A multi-tier data center simulation, platform". In: *2009 IEEE International Conference on Cluster Computing and Workshops*. Aug. 2009, pp. 1–9. DOI: 10.1109/CLUSTR.2009.5289159.

[142] LLVM Project. *CIRCT: Circuit IR Compilers and Tools*. https://github.com/llvm/circt. 2024.

[143] lowRISC. *Ibex: An embedded 32 bit RISC-V CPU core*. https://ibex-core.readthedocs.io/en/latest/. 2024.

[144] Y. Lv et al. "CounterMiner: Mining Big Performance Data from Hardware Counters". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2018, pp. 613–626. DOI: 10.1109/MICRO.2018.00056.

[145] *LZO real-time data compression library*. http://www.oberhumer.com/opensource/lzo/.

[146] Ikuo Magaki et al. "ASIC Clouds: Specializing the Datacenter". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 178–190. DOI: 10.1109/ISCA.2016.25.

[147] Albert Magyar et al. "Golden Gate: Bridging The Resource-Efficiency Gap Between ASICs and FPGA Prototypes". In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2019, pp. 1–8. DOI: 10.1109/ICCAD45719.2019.8942087.

[148] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. "Effectiveness of Trace Sampling for Performance Debugging Tools". In: *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '93. Santa Clara, California, USA: ACM, 1993, pp. 248–259. ISBN: 0-89791-580-1. DOI: 10.1145/166955.167023. URL: http://doi.acm.org/10.1145/166955.167023.

[149] John D. McCalpin. "HPL and DGEMM Performance Variability on the Xeon Platinum 8160 Processor". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '18. Dallas, Texas: IEEE Press, 2018, 18:1–18:13. DOI: 10.1109/SC.2018.00021. URL: https://doi.org/10.1109/SC.2018.00021.

[150] David Meisner, Junjie Wu, and Thomas F. Wenisch. "BigHouse: A Simulation Infrastructure for Data Center Systems". In: *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*. ISPASS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 35–45. ISBN: 978-1-4673-1143-4. DOI: 10.1109/ISPASS.2012.6189204.

[151] Sergey Melnik et al. "Dremel: A Decade of Interactive SQL Analysis at Web Scale". In: *Proc. VLDB Endow.* 13.12 (Aug. 2020), pp. 3461–3472. ISSN: 2150-8097. DOI: 10.14778/3415478.3415568. URL: https://doi.org/10.14778/3415478.3415568.

[152] Sergey Melnik et al. "Dremel: Interactive Analysis of Web-Scale Datasets". In: *Proc. VLDB Endow.* 3.1–2 (Sept. 2010), pp. 330–339. ISSN: 2150-8097. DOI: 10.14778/1920841.1920886. URL: https://doi.org/10.14778/1920841.1920886.

[153] J. E. Miller et al. "Graphite: A distributed parallel simulator for multicores". In: *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. Jan. 2010, pp. 1–12. DOI: 10.1109/HPCA.2010.5416635.

[154] A. Mohammad et al. "dist-gem5: Distributed simulation of computer clusters". In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Apr. 2017, pp. 153–162. DOI: 10.1109/ISPASS.2017.7975287.

[155] Tipp Moseley, Neil Vachharajani, and William Jalby. "Hardware Performance Monitoring for the Rest of Us: A Position and Survey". In: *8th Network and Parallel Computing (NPC)*. Ed. by Erik Altman and Weisong Shi. Vol. LNCS-6985. Network and Parallel Computing. Part 8: Session 8: Microarchitecture. Changsha,, China: Springer, Oct. 2011, pp. 293–312. DOI: 10.1007/978-3-642-24403-2_23. URL: https://hal.inria.fr/hal-01593009.

[156] I. Moussa, T. Grellier, and G. Nguyen. "Exploring SW performance using SoC transaction-level modeling". In: *2003 Design, Automation and Test in Europe Conference and Exhibition*. Mar. 2003, 120–125 suppl. DOI: 10.1109/DATE.2003.1186682.

[157] *Network Maximum Transmission Unit (MTU) for Your EC2 Instance*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/network_mtu.html. 2019.

[158] Rolf Neugebauer et al. "Understanding PCIe Performance for End Host Networking". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. Budapest, Hungary: Association for Computing Machinery, 2018, pp. 327–341. ISBN: 9781450355674. DOI: 10.1145/3230543.3230560. URL: https://doi.org/10.1145/3230543.3230560.

[159] Dima Nikiforov et al. "RoSÉ: A Hardware-Software Co-Simulation Infrastructure Enabling Pre-Silicon Full-Stack Robotics SoC Evaluation". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA '23. Orlando, FL, USA: Association for Computing Machinery, 2023. ISBN: 9798400700958. DOI: 10.1145/3579371.3589099. URL: https://doi.org/10.1145/3579371.3589099.

[160] Stanko Novakovic et al. "Scale-out NUMA". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 3–18. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541965.

[161] NVIDIA. *NVIDIA Deep Learning Accelerator*. https://nvdla.org/. 2024.

[162] *NVIDIA BlueField Data Processing Units*. https://www.nvidia.com/en-us/networking/products/data-processing-unit/. 2023.

[163] U. Y. Ogras and R. Marculescu. "Analytical Router Modeling for Networks-on-Chip Performance Analysis". In: *2007 Design, Automation Test in Europe Conference Exhibition*. Apr. 2007, pp. 1–6. DOI: 10.1109/DATE.2007.364440.

[164] David Patterson et al. *The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink*. 2022. arXiv: 2204.05149 [cs.LG]. URL: https://arxiv.org/abs/2204.05149.

[165] M. Pellauer et al. "HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing". In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. Feb. 2011, pp. 406–417. DOI: 10.1109/HPCA.2011.5749747.

[166] Johan Peltenburg et al. "Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow". In: *29th International Conference on Field Programmable Logic and Applications*. DOI: 10.1109/FPL.2019.00051.

[167] Nathan Pemberton. "Enabling Efficient and Transparent Remote Memory Access in Disaggregated Datacenters". MA thesis. EECS Department, University of California, Berkeley, Dec. 2019. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-154.html.

[168] Nathan Pemberton and Alon Amid. "FireMarshal: Making HW/SW Co-Design Reproducible and Reliable". In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2021, pp. 299–309. DOI: 10.1109/ISPASS51385.2021.00052.

[169] Erez Perelman et al. "Using SimPoint for Accurate and Efficient Simulation". In: *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '03. San Diego, CA, USA: Association for Computing Machinery, 2003, pp. 318–319. ISBN: 1581136641. DOI: 10.1145/781027.781076. URL: https://doi.org/10.1145/781027.781076.

[170] Arash Pourhabibi et al. "Optimus Prime: Accelerating Data Transformation in Servers". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 1203–1216. ISBN: 9781450371025. DOI: 10.1145/3373376.3378501. URL: https://doi.org/10.1145/3373376.3378501.

[171] *Product Brief: Intel Atom C3000 Processor*. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/atom-c3000-family-brief.pdf. 2017.

[172] *Project Zipline*. https://github.com/opencomputeproject/Project-Zipline. 2021.

[173] *Protocol Buffers — Google Developers*. https://developers.google.com/protocol-buffers.

[174] Kishore Punniyamurthy, Behzad Boroujerdian, and Andreas Gerstlauer. "GATSim: Abstract Timing Simulation of GPUs". In: *Proceedings of the Conference on Design, Automation & Test in Europe*. DATE '17. Lausanne, Switzerland: European Design and Automation Association, 2017, pp. 43–48. URL: http://dl.acm.org/citation.cfm?id=3130379.3130390.

[175] Andrew Putnam et al. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. IEEE Press, June 2014, pp. 13–24. ISBN: 978-1-4799-4394-4.

[176] Weikang Qiao et al. "High-Throughput Lossless Compression on Tightly Coupled CPU-FPGA Platforms". In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2018, pp. 37–44. DOI: 10.1109/FCCM.2018.00015.

[177] Deepti Raghavan et al. "Breakfast of Champions: Towards Zero-Copy Serialization with NIC Scatter-Gather". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '21. Ann Arbor, Michigan: Association for Computing Machinery, 2021, pp. 199–205. ISBN: 9781450384384. DOI: 10.1145/3458336.3465287. URL: https://doi.org/10.1145/3458336.3465287.

[178] Parthasarathy Ranganathan et al. "Warehouse-Scale Video Acceleration: Co-Design and Deployment in the Wild". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 600–615. ISBN: 9781450383172. DOI: 10.1145/3445814.3446723. URL: https://doi.org/10.1145/3445814.3446723.

[179] Steven K. Reinhardt et al. "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers". In: *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '93. Santa Clara, California, USA: ACM, 1993, pp. 48–60. ISBN: 0-89791-580-1. DOI: 10.1145/166955.166979. URL: http://doi.acm.org/10.1145/166955.166979.

[180] Gang Ren et al. "Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers". In: *IEEE Micro* (2010), pp. 65–79. URL: http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.68.

[181] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. "DRAMSim2: A Cycle Accurate Memory System Simulator". In: *IEEE Computer Architecture Letters* 10.1 (2011), pp. 16–19. DOI: 10.1109/L-CA.2011.4.

[182] Davide Rossi et al. "PULP: A parallel ultra low power platform for next generation IoT applications". In: *2015 IEEE Hot Chips 27 Symposium (HCS)*. 2015, pp. 1–39. DOI: 10.1109/HOTCHIPS.2015.7477325.

[183] J. A. Rowson. "Hardware/Software Co-Simulation". In: *31st Design Automation Conference*. June 1994, pp. 439–440. DOI: 10.1109/DAC.1994.204143.

[184] Stephen M. Rumble et al. "It's Time for Low Latency". In: *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*. HotOS'13. Napa, California: USENIX Association, 2011, pp. 11–11.

[185] Sudhir Satpathy et al. "A 1.4GHz 20.5Gbps GZIP decompression accelerator in 14nm CMOS featuring dual-path out-of-order speculative Huffman decoder and multi-write enabled register file array". In: *2019 Symposium on VLSI Circuits*. 2019, pp. C238–C239. DOI: 10.23919/VLSIC.2019.8777934.

[186] *Scaling Acceleration Capacity from 5 to 50 Gbps and Beyond with Intel QuickAssist Technology*. https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/scaling-acceleration-capacity-brief.pdf. 2013.

[187] Jürgen Schnerr et al. "High-performance Timing Simulation of Embedded Software". In: *Proceedings of the 45th Annual Design Automation Conference*. DAC '08. Anaheim, California: ACM, 2008, pp. 290–295. ISBN: 978-1-60558-115-6. DOI: 10.1145/1391469.1391543. URL: http://doi.acm.org/10.1145/1391469.1391543.

[188] N. Sehatbakhsh et al. "Spectral profiling: Observer-effect-free profiling by monitoring EM emanations". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–11. DOI: 10.1109/MICRO.2016.7783762.

[189] SiFive. *block-inclusivecache-sifive*. https://github.com/sifive/block-inclusivecache-sifive. 2024.

[190] SiFive. *SiFive HiFive Unleashed Getting Started Guide*. https://sifive.cdn.prismic.io/sifive/fa3a584a-a02f-4fda-b758-a2def05f49f9_hifive-unleashed-getting-started-guide-v1p1.pdf. 2018.

[191] SiFive, Inc. *SiFive TileLink Specification 1.7*. https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf. 2017.

[192] SiFive, Inc. *SiFive TileLink Specification 1.9.3*. https://www.sifive.com/document-file/tilelink-spec-1.9.3. 2023.

[193] Arjun Singh et al. "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network". In: *Commun. ACM* 59.9 (Aug. 2016), pp. 88–97. ISSN: 0001-0782. DOI: 10.1145/2975159. URL: https://doi.org/10.1145/2975159.

[194] Przemyslaw Skibinski. *lzbench*. https://github.com/inikep/lzbench. 2022.

[195] *Snappy compressed format description*. https://github.com/google/snappy/blob/main/format_description.txt. 2011.

[196] *Snappy Testdata*. https://github.com/google/snappy/tree/main/testdata.

[197] *Snappy: A fast compressor/decompressor*. https://github.com/google/snappy.

[198] Akshitha Sriraman and Abhishek Dhanotia. "Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 733–750. ISBN: 9781450371025. URL: https://doi.org/10.1145/3373376.3378450.

[199] *strace: strace is a diagnostic, debugging and instructional userspace utility for Linux*. https://github.com/strace/strace. 2019.

[200] Chen Sun et al. "Single-chip microprocessor that communicates directly using light". In: *Nature* 528.7583 (Dec. 2015), pp. 534–538. ISSN: 0028-0836.

[201] Zhangxi Tan. "Using FPGAs to Simulate Novel Datacenter Network Architectures At Scale". PhD thesis. EECS Department, University of California, Berkeley, June 2013.

[202] Zhangxi Tan et al. "A Case for FAME: FPGA Architecture Model Execution". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA '10. Saint-Malo, France: ACM, 2010, pp. 290–301. ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1815999.

[203] Zhangxi Tan et al. "DIABLO: A Warehouse-Scale Computer Network Simulator Using FPGAs". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: ACM, 2015, pp. 207–221. ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694362.

[204] Zhangxi Tan et al. "RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors". In: *Proceedings of the 47th Design Automation Conference*. DAC '10. Anaheim, California: ACM, 2010, pp. 463–468. ISBN: 978-1-4503-0002-5. DOI: 10.1145/1837274.1837390. URL: http://doi.acm.org/10.1145/1837274.1837390.

[205] "Tape-Out Course: Silicon in a Semester [Society News]". In: *IEEE Solid-State Circuits Magazine* 14.2 (2022), pp. 66–75. DOI: 10.1109/MSSC.2022.3163619.

[206] Willy Tarreau and Dave Rodgman. *LZO stream format as understood by Linux's LZO decompressor*. https://www.kernel.org/doc/Documentation/lzo.txt. 2018.

[207] Petroc Taylor. *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025*. https://www.statista.com/statistics/871513/worldwide-data-created/. 2023.

[208] UCB-BAR. *MMIO-Based FFT Generator for Chipyard*. https://github.com/ucb-bar/FFTGenerator. 2024.

[209] *Updating a Message Type — Language Guide — Protocol Buffers — Google Developers*. https://developers.google.com/protocol-buffers/docs/proto#updating.

[210] Kushagra Vaid. *Hardware innovation for data growth challenges at cloud-scale*. https://azure.microsoft.com/en-us/blog/hardware-innovation-for-data-growth-challenges-at-cloud-scale/. 2019.

[211] Kushagra Vaid. *Improved cloud service performance through ASIC acceleration*. https://azure.microsoft.com/en-us/blog/improved-cloud-service-performance-through-asic-acceleration/. 2019.

[212] VexiiRiscv. *Introduction - VexiiRiscv Documentation*. https://spinalhdl.github.io/VexiiRiscv-RTD/master/VexiiRiscv/Introduction/. 2024.

[213] Stavros Volos et al. "Fat Caches for Scale-Out Servers". In: *IEEE Micro* 37.2 (Mar. 2017), pp. 90–103. ISSN: 0272-1732. DOI: 10.1109/MM.2017.32.

[214] Edward Wang et al. "A Methodology for Reusable Physical Design". In: *2020 21st International Symposium on Quality Electronic Design (ISQED)*. 2020, pp. 243–249. DOI: 10.1109/ISQED48828.2020.9136999.

[215] John Wawrzynek et al. *RAMP: A Research Accelerator for Multiple Processors*. Tech. rep. UCB/EECS-2006-158. EECS Department, University of California, Berkeley, Nov. 2006.

[216] Sewook Wee et al. "A Practical FPGA-based Framework for Novel CMP Research". In: *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Program mable Gate Arrays*. FPGA '07. Monterey, California, USA: ACM, 2007, pp. 116–125. ISBN: 978-1-59593-600-4. DOI: 10.1145/1216919.1216936. URL: http://doi.acm.org/10.1145/1216919.1216936.

[217] Johannes Weiner et al. "TMO: Transparent Memory Offloading in Datacenters". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2022. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 609–621. ISBN: 9781450392051. DOI: 10.1145/3503222.3507731. URL: https://doi.org/10.1145/3503222.3507731.

[218] Joonho Whangbo et al. "FireAxe: Partitioned FPGA-Accelerated Simulation of Large-Scale RTL Designs". In: *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 2024, pp. 501–515. DOI: 10.1109/ISCA59077.2024.00044.

[219] Wikichip. *POWER9 - Microarchitectures - IBM*. https://en.wikichip.org/wiki/ibm/microarchitectures/power9. 2023.

[220] Wikichip. *Skylake (server) - Microarchitectures - Intel*. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server). 2023.

[221] Adam Wolnikowski et al. "Zerializer: Towards Zero-Copy Serialization". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '21. Ann Arbor, Michigan: Association for Computing Machinery, 2021, pp. 206–212. ISBN: 9781450384384. DOI: 10.1145/3458336.3465283. URL: https://doi.org/10.1145/3458336.3465283.

[222] G. Wu et al. "GPGPU performance and power estimation using machine learning". In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2015, pp. 564–576. DOI: 10.1109/HPCA.2015.7056063.

[223] Roland E. Wunderlich et al. “SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling”. In: vol. 31. 2. New York, NY, USA: Association for Computing Machinery, May 2003, pp. 84–97. DOI: 10.1145/871656.859629. URL: https://doi.org/10.1145/871656.859629.

[224] *YAML: YAML Ain’t Markup Language*. https://yaml.org/.

[225] T. Yoshino et al. “Performance optimization of TCP/IP over 10 Gigabit Ethernet by precise instrumentation”. In: *SC ’08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. Nov. 2008, pp. 1–12. DOI: 10.1109/SC.2008.5215913.

[226] F. Zaruba and L. Benini. “The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (Nov. 2019), pp. 2629–2640. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2926114.

[227] Sizhuo Zhang, Hari Angepat, and Derek Chiou. “HGum: Messaging framework for hardware accelerators”. In: *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2017, pp. 1–8. DOI: 10.1109/RECONFIG.2017.8279799.

[228] Jerry Zhao et al. “COBRA: A Framework for Evaluating Compositions of Hardware Branch Predictors”. In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2021, pp. 310–320. DOI: 10.1109/ISPASS51385.2021.00053.

[229] Jerry Zhao et al. “Constellation: An Open-Source SoC-Capable NoC Generator”. In: *2022 15th IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc)*. 2022, pp. 1–7. DOI: 10.1109/NoCArc57472.2022.9911299.

[230] Jerry Zhao et al. “SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine”. In: (May 2020).

[231] Xinnian Zheng, Lizy K. John, and Andreas Gerstlauer. “Accurate Phase-level Cross-platform Power and Performance Estimation”. In: *Proceedings of the 53rd Annual Design Automation Conference*. DAC ’16. Austin, Texas: ACM, 2016, 4:1–4:6. ISBN: 978-1-4503-4236-0. DOI: 10.1145/2897937.2897977. URL: http://doi.acm.org/10.1145/2897937.2897977.

[232] Pin Zhou et al. “iWatcher: Efficient Architectural Support for Software Debugging”. In: *Proceedings of the 31st Annual International Symposium on Computer Architecture*. ISCA ’04. München, Germany: IEEE Computer Society, 2004, pp. 224–. ISBN: 0-7695-2143-6. URL: http://dl.acm.org/citation.cfm?id=998680.1006720.

[233] Jacob Ziv and Abraham Lempel. “A universal algorithm for sequential data compression”. In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343. DOI: 10.1109/TIT.1977.1055714.

[234] *zlib Home Site - A Massively Spiffy Yet Delicately Unobtrusive Compression Library*. https://www.zlib.net/.

[235] *Zstandard - Real-time data compression algorithm*. `https://facebook.github.io/zstd/`.

Appendix A

Artifact appendix: FirePerf

This artifact appendix describes how to reproduce results demonstrated in Chapter 5 by running FirePerf/FireSim simulations on Amazon EC2 F1 instances.

A.1 Artifact checklist (meta-information)

- **Run-time environment:** AWS FPGA Developer AMI 1.6.0
- **Hardware:** AWS EC2 Instances (`c5.4xlarge`/`f1.4xlarge`)
- **Metrics:** Bandwidth Results (Gbits/s)
- **Output:** Bandwidth Results, TraceRV Flame Graphs, AutoCounter Queue Occupancy Graphs
- **Experiments:** `iperf3` benchmarks
- **How much disk space required?:** 75GB (on EC2 instance)
- **How much time is needed to prepare workflow?:** 1.5 hours (scripted installation)
- **How much time is needed to complete experiments?:** 1 hour
- **Publicly available?:** Yes
- **Code licenses:** Several, see download
- **Archived:** `https://doi.org/10.5281/zenodo.3561040`

A.2 Description

How delivered.

A version of FirePerf that reproduces the results in this chapter is available openly on Zenodo; its use is described in this artifact appendix. FirePerf is also open-sourced within the FireSim

project. Those intending to use the FirePerf tools for their own designs should use FireSim directly: `https://github.com/firesim/firesim`.

Hardware dependencies.

One `c5.4xlarge` instance (also referred to as the "manager" instance) and at least one `f1.4xlarge` instance on Amazon EC2 are required.

Software dependencies.

Installing `mosh` (`https://mosh.org/`) on your local machine is highly recommended for reliable access to EC2 instances.

A.3 Installation

First, follow the instructions on the FireSim website to create a manager instance on AWS: `https://docs.fires.im/en/1.6.0/Initial-Setup/index.html`. You must complete up to and including Section 2.3.1.2, "Key Setup, Part 2", with the following changes in Section 2.3.1:

1. When instructed to launch a `c4.4xlarge` instance, choose a `c5.4xlarge` instead.

2. When instructed to copy a long script into the text box in "Advanced Details," instead copy only the following script:

```
#!/bin/bash
sudo yum install -y mosh
echo "PS1='\u@\H:\w\\$ '" >> /home/centos/.bashrc
```

3. When entering the root EBS volume size, use 1000GB rather than 300GB.

At this point, you should have a manager instance setup, with an IP address and key. Use either `ssh` or `mosh` to login to the instance. From this point forward, all commands should be run on the manager instance. Begin by pulling the FirePerf codebase from Zenodo onto the instance, like so:

```
# Enter the wget as a single line:
$ wget -O fireperf.zip
      https://zenodo.org/record/3561041/files/fireperf.zip
$ unzip fireperf.zip
```

Next, from the home directory, run the following to install some basic dependencies:

```
$ ./fireperf/scripts/machine-launch-script.sh
```

This step should take around 3 minutes. At the end, the script will print:

```
Setup complete. You should log out and log back in now.
```

At this point, you need to log out and log back into the manager instance. Once logged into the manager again, run:

```
$ cd fireperf
$ ./scripts/first-clone-setup-fast.sh
```

This step should take around 1 hour. Upon successful completion, it will print:

```
first-clone-setup-fast.sh complete.
```

To finish-up the installation process, run the following:

```
$ source sourceme-f1-manager.sh
$ cd deploy
$ firesim managerinit
```

A.4 Experiment workflow

For rapid exploration, we provide a short workload that runs a FirePerf/FireSim simulation for the baseline, single-core, networked result in our work (the 1.67 Gbit/s number in Table 5.1), incorporating both TraceRV/Flame Graph construction and AutoCounter logging. The included infrastructure will automatically dispatch simulations to an `f1.4xlarge` instance, which has two Xilinx FPGAs. The target design consists of two nodes simulated by FireSim (one on each FPGA), each with a NIC, that communicate through a simulated two-port Ethernet switch. To run the experiment, first we build target-software for the simulation, which should take approximately 2 minutes:

```
$ cd workloads
$ make iperf3-trigger-slowcopy-cover
```

Next, we must make a change to the `run-ae-short.sh` script to support running on your own EC2 instances. Open `run-ae-short.sh` (located in the `workloads` directory you are currently in) in a text editor. You will need to uncomment line 40 and comment out line 41, so that they look like so:

```
    runserialwithlaunch $1
    #runserialnolaunch $1
```

Now, we can run the simulation:

```
$ ./run-ae-short.sh
```

This process will take around 1 hour. Upon completion, it will print:

```
AE run complete.
```

A.5 Evaluation and expected result

Results from the run of `run-ae-short.sh` will be located in a subdirectory in `~/fireperf/deploy/results-workload/` on your manager instance. The subdirectory name will look like the following, but adjusted for the date/time at which you run the workload:
`2020-01-18--06-00-00-iperf3-trigger-slowcopy-cover-singlecore`

We will refer to this subdirectory as your "*AE Subdirectory*" in the rest of this document. Once in this subdirectory, there are three results we are interested in looking at, described in the following subsections.

Overall performance result.

Within your "*AE Subdirectory*", open `iperf3-client/uartlog` in a text editor and search for "Gbits". You will be taken to a section of the console output from the simulated system that is produced by `iperf3` running on the system. The number we are interested in will be listed in the "Bitrate" column, with "sender" written in the same row. This is the workload used to produce the Linux 5.3-rc4, Single-core, Networked number in Table 5.1 (with a result of 1.67 Gbits/s).

Generated Flame Graph.

The generated flame graph from this run (constructed with FirePerf/TraceRV trace collection) is available in `iperf3-server/TRACEFILE0.svg` in your "*AE Subdirectory*". This flame graph should be very similar to that shown in Figure 5.3 , since it is generated from your run of the same workload.

AutoCounter output.

As this workload was running, AutoCounter results were also being collected from the simulations. The scripts post-process these into a graph that will be similar to Figure 5.5, but will contain a single bar. Your generated version of this graph will be located in `iperf3-client/nic_queue_send_req.pdf`, relative to your "*AE Subdirectory*". You should expect a single bar at queue-occupancy 1, with around 900000 cycles., close to the Baseline bar in Figure 5.5.

A.6 Experiment customization

FirePerf is heavily customizable as a result of being integrated into the FireSim environment, which itself provides a vast number of configuration options. FirePerf's TraceRV/stack unwinding

feature works with any RISC-V processor simulated in FireSim that can pipe out its PC trace to the top level. FirePerf's AutoCounter feature is more general—it can be added to any Chisel design simulated in FireSim. The FireSim documentation describes the extensive customization options available: `https://docs.fires.im`

We provide pre-built FPGA images for the designs in this work, encoded in the configuration files included in the artifact. Regenerating the supplied FPGA images is also possible, by running `firesim buildafi` in `~/fireperf/deploy/`.

Extended Benchmarks.

We also include a script in the artifact to reproduce the other FirePerf workloads in this chapter, covering Tables 5.1, 5.2, 5.3, and 5.4 and Figures 5.2, 5.3, 5.7, and 5.5.

Before running this script, it is first required to run the following in `~/fireperf/deploy/workloads` to build all target-software images:

```
$ make all-iperf3
```

Next, we must make a change to the `run-all-iperf3.sh` script to support running on your own EC2 instances. Open `run-all-iperf3.sh` (located in the workloads directory you are currently in) in a text editor. You will need to uncomment line 39 and comment out line 41, so that lines 39-41 look like so:

```
    runparallelwithlaunch $1
    #runserialwithlaunch $1
    #runserialnolaunch $1
```

Next, ensure that your results directory (`~/fireperf/deploy/results-workload/`) does not contain existing results from a previous run. Finally, to start all 21 simulations, run the following:

```
$ cd fireperf-dataprocess
$ ./full-runner.sh
```

Once this script completes, final results will be located in: `~/fireperf/deploy/workloads/fireperf-dataprocess/02_covered/`. This directory contains the following:

- `generated-tables/`: Contains LaTeX source for Tables 5.1, 5.2, 5.3, and 5.4, populated with bandwidth results from this run.
- `generated-flamegraphs/`: Contains newly generated PDFs for the flame graphs in this chapter: Figures 5.2, 5.3, and 5.7.
- `nic_queue_send_req.pdf`: A newly generated version of Figure 5.5.

A.7 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

Appendix B

Artifact appendix: A hardware accelerator for protocol buffers

This artifact appendix describes how to reproduce the protobuf accelerator evaluation results in Chapter 6. As in Chapter 6.5, we will use FireSim FPGA-accelerated simulations to cycle-exactly simulate the entire RISC-V SoC containing the protobuf accelerator. We will boot Linux on this system and run both microbenchmarks and HyperProtoBench to collect accelerator performance metrics.

B.1 Artifact checklist (meta-information)

- **Run-time environment:** AWS FPGA Developer AMI 1.6.1.
- **Hardware:** AWS EC2 instances: 1× `c5.9xlarge`, 1× `f1.16xlarge`, 1× `m4.large`.
- **Metrics:** Protobuf serialization/deserialization throughput (Gbits/s).
- **Output:** Serialization/deserialization performance plots.
- **Experiments:** FireSim simulations of protobuf accelerator incorporated into a RISC-V SoC, running serialization/deserialization microbenchmarks and HyperProtoBench.
- **How much disk space is required?:** 200 GB (on EC2 instance).
- **How much time is needed to prepare workflow?:** 2 hours (scripted installation).
- **How much time is needed to complete experiments?:** 3.5 hours (scripted run).
- **Publicly available:** Yes.
- **Code licenses:** Several, see download.
- **Archived:** `https://doi.org/10.5281/zenodo.5433464`, `https://doi.org/10.5281/zenodo.5433448`, `https://doi.org/10.5281/zenodo.5433434`, `https://doi.org/10.5281/zenodo.5433410`, and `https://doi.org/10.5281/zenodo.5433364`.

B.2 Description

How to access

The artifact consists of five git repositories preserved on Zenodo:

1. `firesim-protoacc-ae`: Top-level FireSim simulation environment. (`https://doi.org/10.5281/zenodo.5433464`)

2. `chipyard-protoacc-ae`: Chipyard RISC-V SoC generation environment. (`https://doi.org/10.5281/zenodo.5433448`)

3. `protoacc-ae`: Protobuf accelerator design, software, and scripts. (`https://doi.org/10.5281/zenodo.5433434`)

4. `protobuf-library-for-accel-ae`: Fork of protobuf library modified for accelerator support. (`https://doi.org/10.5281/zenodo.5433410`)

5. `HyperProtoBench`: Protobuf serialization/deserialization benchmarks representative of key serialization-framework user services at scale, open-sourced for this work. This is a fork of our upstream release (`https://github.com/google/HyperProtoBench`) customized for accelerator benchmarking. (`https://doi.org/10.5281/zenodo.5433364`)

Users need not download the latter four repositories manually—they will be obtained automatically from Zenodo when the first repository is set up in the next section.

Hardware dependencies

One AWS EC2 `c5.9xlarge` instance (also referred to as the "manager" instance), one `f1.16xlarge` instance, and one `m4.large` instance are required. The latter two will be launched automatically by FireSim's manager.

To optionally run FPGA builds (see Appendix B.6), two additional `z1d.6xlarges` are required, however we provide pre-built FPGA images to avoid the long latency (˜10 hours) of this process.

Software dependencies

Installing `mosh` (`https://mosh.org/`) on your local machine is highly recommended for reliable access to EC2 instances. All other requirements are automatically installed by scripts in the following sections.

B.3 Installation

First, follow the instructions on the FireSim website[1] to create a manager instance on EC2. You must complete up to and including "Section 2.3.1.2: Key Setup, Part 2", with the following changes in "Section 2.3.1":

1. When instructed to launch a `c5.4xlarge` instance, choose a `c5.9xlarge` instead.
2. When entering the root EBS volume size, use 1000GB rather than 300GB.

Once you have completed up to and including "Section 2.3.1.2" in the FireSim docs, you should have a manager instance set up, with an IP address and key. Use either `ssh` or `mosh` to login to the instance.

From this point forward, all commands should be run on the manager instance.

Begin by downloading the top-level repository from Zenodo, like so:

```
$ cd ~/
# Enter as a single line:
$ wget -O firesim-protoacc-ae.zip https://zenodo.org/
       record/5433465/files/firesim-protoacc-ae.zip
$ unzip firesim-protoacc-ae.zip
```

Next, run the following, which will initialize all dependencies and run basic FireSim and Chipyard setup steps (RISC-V toolchain installation, matching host toolchain installation, etc.):

```
$ cd firesim-protoacc-ae
$ ./scripts/first-clone-setup-fast.sh
```

This step should take around 1.5 hours. Upon successful completion, it will print:

```
first-clone-setup-fast.sh complete.
```

Once this is complete, run:

```
$ source sourceme-f1-manager.sh
```

Sourcing this file will have set up your environment to run the protobuf accelerator simulations.

Finally, in the FireSim docs, follow the steps in (only) "Section 2.3.3: Completing Setup Using the Manager"[2]. Once you have completed this, your manager instance is fully set up to run protobuf accelerator simulations.

[1] `https://docs.fires.im/en/1.12.0/Initial-Setup/index.html`

[2] `https://docs.fires.im/en/1.12.0/Initial-Setup/Setting-up-your-Manager-Instance.html#completing-setup-using-the-manager`

B.4 Experiment workflow

Now that our environment is set up, we will run the full artifact evaluation script, which does the following:

1. On the manager instance, build the FireSim host-side drivers required to drive the FPGA simulation.
2. On the manager instance, build our modified protobuf library, cross-compile all benchmarks we will run, and construct a Buildroot-based Linux distribution containing these benchmarks, which will be booted on the accelerated system.
3. For isolated Xeon runs, launch an `m4.large`, run benchmarks on it and collect results, and terminate the `m4.large`.
4. Run FireSim simulations, repeat the following for the three classes of benchmarks (accelerated serialization, accelerated deserialization, and plain BOOM):
    a) Launch an `f1.16xlarge` instance.
    b) Copy all simulation infrastructure to the F1 instance.
    c) Run the set of benchmarks on 6 or 7 simulated systems in parallel (one `f1.16xlarge` has 8 FPGAs).
    d) Copy results back to the manager instance.
    e) Terminate the `f1.16xlarge` instance.
5. On the manager instance, re-generate the accelerator performance plots in this chapter, with data collected from your runs.

Note that this script will *not* rebuild FPGA images for the system by default, since each build takes around 10 hours. We instead provide pre-built images by default (see `config_hwdb.ini` in `$PROTOACC_FSIM`). If you would like to build your own images, see Appendix B.6, then return here.

Now, let's run the aforementioned full artifact evaluation script:

```
$ cd $PROTOACC_FSIM
$ ./run-ae-full.sh
```

This will take around 3.5 hours. When complete, it will print:

```
run-ae-full.sh complete.
```

The FireSim manager will have automatically terminated any instances it launched during this process, but please confirm in your AWS EC2 management console that no instances remain besides the manager.

B.5 Evaluation and expected results

Next, we will step through the plots generated from your run of `run-ae-full.sh` in the previous section.

Microbenchmark results

Results from your run will be located in the `$UBENCH_RESULTS` directory:

1. Figure 6.11a: `nonalloc.pdf`
2. Figure 6.11c: `allocd.pdf`
3. Figure 6.11b: `nonalloc-serializer.pdf`
4. Figure 6.11d: `allocd-serializer.pdf`
5. Final speedup results: at the end of `process.py.log` and `process-serialize.py.log`

HyperProtoBench results

Results from your run will be located in the `$HYPER_RESULTS` directory:

1. Figure 6.12: `hyper-des.pdf`
2. Figure 6.13: `hyper-ser.pdf`
3. Final speedup results for serialization and deserialization: near the end of the `SPEEDUPS` file

Once your evaluation is complete, manually terminate your manager instance in the EC2 management console and confirm that no other instances from the evaluation process are left running.

B.6 Experiment customization

Customizing the design

Since the protobuf accelerator is written in Chisel RTL, incorporated into the Chipyard RISC-V SoC generator ecosystem, and modeled at high-performance using FireSim, it can be experimented with in a wide-variety of contexts, including in multi-core systems, attached to in-order processors (instead of the superscalar OoO BOOM used here), and with different memory hierarchy configurations, to name a few. These parameters are too numerous to list here; see the FireSim docs[3], Chipyard docs[4], and tutorial slides[5] for these configuration options.

[3] `https://docs.fires.im/en/1.10.0/`
[4] `https://chipyard.readthedocs.io/en/1.3.0/`
[5] `https://fires.im/isca-2021-tutorial/`

The protobuf accelerator RTL is located in the `$PROTOACC_SRC` directory and can be customized and improved as necessary.

Rebuilding FPGA images

We provide pre-built FPGA images for the designs in this work (generated from the included RTL), encoded in the configuration files in the artifact.

Regenerating the supplied FPGA images can also be done by modifying the S3 bucket name in `$PROTOACC_FSIM/config_build.ini` to an unused bucket name (that the manager will create), then running `./buildafi.sh` in the `$PROTOACC_FSIM` directory. This will take around 10 hours, require two `z1d.6xlarge` instances, generate two new AGFIs (i.e., FPGA bitstreams on EC2 F1), and place their `config_hwdb.ini` entry in `$BUILT_HWDB_ENTRIES/[config name]`. To use the new AGFI, replace the existing entry in the `config_hwdb.ini` file in `$PROTOACC_FSIM` (or, for a new config, add it). If generating your own FPGA images, you must also set the correct value for `customruntimeconfig` in the `config_hwdb.ini` entry to obtain correct memory system performance:

```
customruntimeconfig=2GHz-runtime-conf-32MBLLC-qc.conf
```

When an FPGA build completes, the FireSim manager will automatically terminate the instances it launched during the build process, but please confirm in your AWS EC2 management console that no instances remain besides the manager. More details about the FireSim FPGA build process can be found in the FireSim docs[6]. Note that many of the FireSim manager build configuration files are in a non-standard location to simplify scripting for artifact evaluation. Open `buildafi.sh` to see their locations.

B.7 Methodology

Submission, reviewing and badging methodology:

- `https://www.acm.org/publications/policies/artifact-review-badging`
- `http://cTuning.org/ae/submission-20201122.html`
- `http://cTuning.org/ae/reviewing-20201122.html`

[6] `https://docs.fires.im/en/1.10.0/Building-a-FireSim-AFI.html`

Appendix C

Artifact appendix: CDPU

This artifact appendix describes how to reproduce the CDPU Design Space Exploration results from Chapter 7. As in Chapter 7.6, we will use FireSim FPGA-accelerated simulations to cycle-exactly simulate the entire RISC-V SoC containing the RTL implementations of CDPUs (compression and decompression accelerators). We will run HyperCompressBench, the benchmark suite we created from fleet-wide profiling at Google, on both a Xeon system (for the baseline) and our RISC-V SoC augmented with CDPUs. We will sweep the design parameters explored in Chapter 7.6 to collect accelerator performance metrics and reproduce the accelerator trade-offs and insights discussed earlier.

C.1 Artifact checklist (meta-information)

- **Run-time environment:** AWS FPGA Developer AMI 1.12.1.
- **Hardware:** AWS EC2 instances: 1× `c5.9xlarge`, 16× `f1.2xlarge`, 1× `m4.large`.
- **Metrics:** Compression and decompression throughput (GB/s), compression ratio.
- **Output:** Compression and decompression performance and compression ratio plots. HyperCompressBench call-size distribution plots. Re-generation of chapter text that contains data.
- **Experiments:** FireSim simulations of compression and decompression accelerators incorporated into a RISC-V SoC, running HyperCompressBench.
- **How much disk space is required?:** 2000GB (on EC2 instance).
- **How much time is needed to prepare workflow?:** 1 hour (scripted installation).
- **How much time is needed to complete experiments?:** 6 hours for Snappy, 110 hours for ZStd (both fully automated).
- **Publicly available:** Yes.
- **Code licenses:** Several, see download.

- **Archived:**
  - `https://doi.org/10.5281/zenodo.7812634`
  - `https://doi.org/10.5281/zenodo.7812577`
  - `https://doi.org/10.5281/zenodo.7812573`
  - `https://doi.org/10.5281/zenodo.7812563`

C.2 Description

How to access

The artifact consists of four git repositories preserved on Zenodo.

1. `chipyard-compress-acc-ae`: Chipyard RISC-V SoC generation environment, customized for CDPU evaluation. Zenodo: `https://doi.org/10.5281/zenodo.7812634`

2. `firesim-compress-acc-ae`: FireSim simulation environment, customized for CDPU evaluation. Zenodo: `https://doi.org/10.5281/zenodo.7812577`

3. `compress-acc-ae`: Compression and decompression accelerator implementation (RTL), software, and scripts. Zenodo: `https://doi.org/10.5281/zenodo.7812573`

4. `HyperCompressBench`: Compression and decompression benchmarks representative of compression and decompression usage in Google's datacenter fleet, created and open-sourced for this work. Zenodo: `https://doi.org/10.5281/zenodo.7812563`

Users need not download the latter three repositories manually—they will be obtained automatically from Zenodo when the first repository is set up in the next section.

Hardware dependencies

One AWS EC2 `c5.9xlarge` instance (also referred to as the "manager" instance), 16 `f1.2xlarge` instances, and one `m4.large` instance are required. The latter two instance types will be launched automatically by FireSim's manager.

To optionally run FPGA builds (see Appendix C.6), seven additional `z1d.6xlarges` are required, however we provide pre-built FPGA images to avoid the long latency ($\approx$ 18 hours) of this process.

Software dependencies

Installing `mosh` (`https://mosh.org/`) on your local machine is highly recommended for reliable access to EC2 instances. All other requirements are automatically installed by scripts in the following sections.

C.3 Installation

First, follow the instructions on the FireSim website[1] to create a manager instance on EC2. You must complete up to and including "Section 1.3.1.2: Key Setup, Part 2", with the following changes in "Section 1.3.1":

1. When instructed to launch a `c5.4xlarge` instance, choose a `c5.9xlarge` instead.
2. When entering the root EBS volume size, use 2000GB rather than 300GB.

Once you have completed up to and including "Section 1.3.1.2" in the FireSim docs, you should have a manager instance set up, with an IP address and key. Use either `ssh` or `mosh` to login to the instance.

```
# Option 1: USE SSH
$ ssh -i KEY.pem centos@IP_ADDR
# Option 2: USE MOSH
$ mosh --ssh="ssh -i KEY.pem" centos@IP_ADDR
```

From this point forward, all commands should be run on the manager instance.

If using `ssh`, be sure to start `screen` or `tmux` on the manager so that the artifact continues running even if your network connection is interrupted.

Begin by fetching the top-level repository from Zenodo, like so:

```
$ cd $HOME
# Enter as a single line:
$ curl -Ls -w %{url_effective} -o a https://doi.org/
       10.5281/zenodo.7812634 > DL_url
$ wget $(cat DL_url)/files/chipyard-compress-acc-ae.zip
$ unzip chipyard-compress-acc-ae.zip
```

Next, run the following, which will initialize all dependencies and run basic Chipyard and FireSim setup steps (RISC-V toolchain installation, host toolchain installation, etc.):

```
$ cd chipyard-compress-acc-ae
$ ./scripts/first-clone-setup-fast.sh
```

This step should take around 45 minutes. Upon successful completion, it will print:

```
first-clone-setup-fast.sh complete.
```

[1] `https://docs.fires.im/en/1.15.2/Initial-Setup/index.html`

Once this is complete, run:

```
$ source env.sh
$ cd sims/firesim
$ source sourceme-f1-manager.sh
```

Finally, run the following to finish setting up FireSim. You can enter your email address when prompted if you plan to run the optional FPGA builds in Appendix C.6, otherwise just hit Enter.

```
$ firesim managerinit
```

Now, your manager instance is fully set up to run CDPU sims.

C.4 Experiment workflow

Now that our manager is set up, we will run the full artifact evaluation script, which will automatically do the following:

1. On the manager instance, extract HyperCompressBench and compile RISC-V/CDPU benchmarks for the ≈35,000 benchmark files we need.
2. For isolated Xeon baseline runs, launch an `m4.large`, run HyperCompressBench on it using `lzbench`, collect results, and terminate the `m4.large`.
3. On the manager instance, build FireSim host-side drivers required to drive each FPGA-accelerated simulation.
4. Launch sixteen `f1.2xlarge` instances, which provide a total of sixteen FPGAs to run simulations on in parallel.
5. Run FireSim simulations, repeating the following for the 16 workloads of interest:
    a) Copy all simulation infrastructure to the F1 instances.
    b) Run the set of benchmarks on 16 simulated systems in parallel (one `f1.2xlarge` has 1 FPGA).
    c) Copy results back to the manager instance.
6. Terminate the sixteen `f1.2xlarge` instances.
7. On the manager, re-generate accelerator performance plots from the chapter (and sections of the chapter text that use this data), using data collected from your runs.

Note that this script will *not* rebuild FPGA images for the system by default, since each build takes around 18 hours. We instead provide pre-built images by default (see `$COMPRESSACC_FSIM/config_others/config_hwdb.yaml`). If you would like to build your own images, see Appendix C.6, then return here.

Now, run the aforementioned full artifact evaluation script:

```
$ cd $COMPRESSACC_FSIM
$ ./run-ae-full.sh
```

This takes around 6 hours for Snappy and 110 hours for ZStd. When complete, it will print:

```
run-ae-full.sh complete.
```

The FireSim manager will automatically terminate any instances it launched during this process.

C.5 Evaluation and expected results

Next, we will step through the plots generated from your run of `run-ae-full.sh` in the previous section. The following results generated from your run will be located in the `$HYPER_RESULTS` directory:

1. Figures 7.7a, 7.7b, 7.7c, and 7.7d:
   `[Snappy,ZSTD]-[C,D]-callsizes.pdf`
2. Figure 7.11: `snappy-decompression.pdf`
3. Figure 7.12: `snappy-compression-ht14.pdf`
4. Figure 7.13: `snappy-compression-ht9.pdf`
5. Figure 7.14: `zstd-decompression.pdf`
6. Figure 7.15: `zstd-compression-ht14.pdf`
7. `FINAL_TEXT_SUMMARIES.txt` contains chapter text re-generated with data obtained from these simulations. This excludes the single ZStd-Decomp-32spec data point, which requires ≈100 additional machine-days of software simulation.
8. Raw results are located in the five `*.csv` files

C.6 Experiment customization

Customizing the design

Since the compression and decompression accelerators are written in Chisel RTL, incorporated into the Chipyard RISC-V SoC generator ecosystem, and modeled at high-performance using FireSim, they can be experimented with in a wide-variety of contexts, including in multi-core systems, attached to various kinds of processors, and with different memory hierarchy configurations, to name a few. These parameters are too numerous to list here; see the FireSim docs[2], Chipyard docs[3], and tutorial slides[4] for these configuration options.

The compression and decompression accelerator RTL is located in the `$COMPRESSACC_SRC` directory and can be customized as necessary, including using the runtime and compile-time configurable parameters outlined in Chapter 7.5, several of which we swept in this artifact evaluation.

Rebuilding FPGA images

We provide pre-built FPGA images for designs in this chapter (generated from the included RTL), encoded in the configuration files in the artifact.

Rebuilding the supplied FPGA images can also be done by running `./buildafi.sh` in the `$COMPRESSACC_FSIM` directory. This will take around 18 hours, require seven `z1d.6xlarge` instances, generate seven new AGFIs (i.e., FPGA bitstreams on EC2 F1), and place their `config_hwdb.yaml` entries in `$BUILT_HWDB_ENTRIES/[config name]`. To use the new AGFIs, replace existing entries in the `$COMPRESSACC_FSIM/config_others/config_hwdb.yaml` file (or, for a new config, add it).

When an FPGA build completes, the FireSim manager will automatically terminate the instances it launched during the build process. More details about the FireSim FPGA build process can be found in the FireSim docs[5]. Note that many of the FireSim manager build configuration files are in a non-standard location to simplify scripting for artifact evaluation. Open `buildafi.sh` to see their locations.

C.7 Methodology

Submission, reviewing and badging methodology:

- `https://www.acm.org/publications/policies/artifact-review-badging`
- `http://cTuning.org/ae/submission-20201122.html`
- `http://cTuning.org/ae/reviewing-20201122.html`

[2] `https://docs.fires.im/en/1.15.2/`
[3] `https://chipyard.readthedocs.io/en/1.8.1/`
[4] `https://fires.im/asplos-2023-tutorial/`
[5] `https://docs.fires.im/en/1.15.2/Building-a-FireSim-AFI.html`