FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design

Sagar Karandikar
University of California, Berkeley
sagark@eecs.berkeley.edu

Albert Ou
University of California, Berkeley
aou@eecs.berkeley.edu

Alon Amid
University of California, Berkeley
alonamid@eecs.berkeley.edu

Howard Mao
University of California, Berkeley
zhemao@eecs.berkeley.edu

Randy Katz
University of California, Berkeley
randy@eecs.berkeley.edu

Borivoje Nikolić
University of California, Berkeley
bora@eecs.berkeley.edu

Krste Asanović
University of California, Berkeley
krste@eecs.berkeley.edu

Abstract

Achieving high-performance when developing specialized hardware/software systems requires understanding and improving 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. In this work, we enable agile full-system performance optimization for hardware/software systems with FirePerf, a set of novel out-of-band system-level performance profiling capabilities integrated into the open-source FireSim FPGA-accelerated hardware simulation platform. Using out-of-band call stack reconstruction and automatic performance counter insertion, FirePerf enables introspecting into hardware and software at appropriate abstraction levels to rapidly identify opportunities for software optimization and hardware specialization, without disrupting end-to-end system behavior like traditional profiling tools. We demonstrate the capabilities of FirePerf with a case study that optimizes the hardware/software stack of an open-source RISC-V SoC with an Ethernet NIC to achieve 8× end-to-end improvement in achievable bandwidth for networking applications running on Linux. We also deploy a RISC-V Linux kernel optimization discovered with FirePerf on commercial RISC-V silicon, resulting in up to 1.72× improvement in network performance.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03.
https://doi.org/10.1145/3373376.3378455

CCS Concepts. • **General and reference → Performance**; • **Hardware → Simulation and emulation**; **Hardware-software codesign**; • **Computing methodologies → Simulation environments**; • **Networks → Network performance analysis**; • **Software and its engineering → Operating systems**.

Keywords. performance profiling; hardware/software co-design; FPGA-accelerated simulation; network performance optimization; agile hardware

ACM Reference Format:
Sagar Karandikar, Albert Ou, Alon Amid, Howard Mao, Randy Katz, Borivoje Nikolić, and Krste Asanović. 2020. 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), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3373376.3378455

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 [9]. 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` [18] and `strace` [4] are

common tools of the trade for software systems developers, but have the potential to perturb the system being evaluated (as demonstrated in Section 4.6). 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 [34]. 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. [29]) 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 [29]), 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× 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.

2 Background

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.

2.1 Target System Design: FireChip SoC

In the case study in Section 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 [6], which is written in parameterized Chisel RTL [7] 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 [46] and the Kendryte K210 SoC [1]. 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.

2.2 FireSim

FireSim [2, 29, 30] is an open-source, FPGA-accelerated, cycle-exact hardware simulation platform that enables automatically modeling RTL designs (including SoCs like FireChip) on FPGAs at 10s to 100s of MHz. To provide a reproducible, deterministic, and accurate simulation, FireSim also provides standard I/O models, such as DRAM [10], disk, UART, and Ethernet network models. In our case study, we will

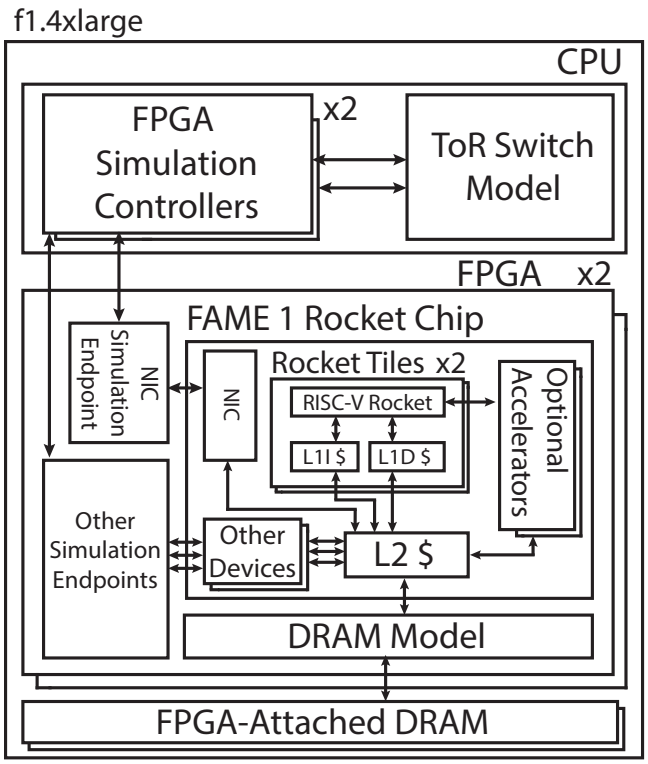

Figure 1. FireSim simulation of a networked 2-node, dual-core FireChip configuration on one AWS `f1.4xlarge` instance with two FPGAs, which will form the basis of the system we will instrument, analyze, and improve.

use FireSim's 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 [31], 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 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.

3 FirePerf

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.

3.1 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 [13, 14, 25] (e.g. Figure 2). These cycle-exact flame graphs allow users to see complete end-to-end Linux kernel (or other software) behavior on simulated SoCs.

3.1.1 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 [29] 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 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 [15]. 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 paper, 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.

3.1.2 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 [20] 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.

3.1.3 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 [13, 14]. This produces a flame graph, a kind of histogram that shows the fraction of total time spent in different parts of the stack trace [25].

An example flame graph generated from FirePerf data is shown in Figure 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.

3.2 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 high 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 [37]. 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 [34]. 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 [26] 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 [31] 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).

4 Using FirePerf to Optimize Linux Network 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× 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.

techniques — Accelerating accelerator adoption.

4.1 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 Section 2.1. 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 [10], and a 200 Gbit/s Ethernet NIC. Each design boots Linux and is capable of running complete Linux-based applications.

4.2 The `iperf3` benchmark

Our driving benchmark is `iperf3` [22], 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 paper, 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 × 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 a `end-trigger` binary, which issue the special instructions described earlier that allow starting/stopping tracing from within the simulation.

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

4.3 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 [46], 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`.

4.4 Baseline Performance Results

Prior work [29] 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.

4.4.1 `iperf3` results on baseline hardware/software configurations. Table 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

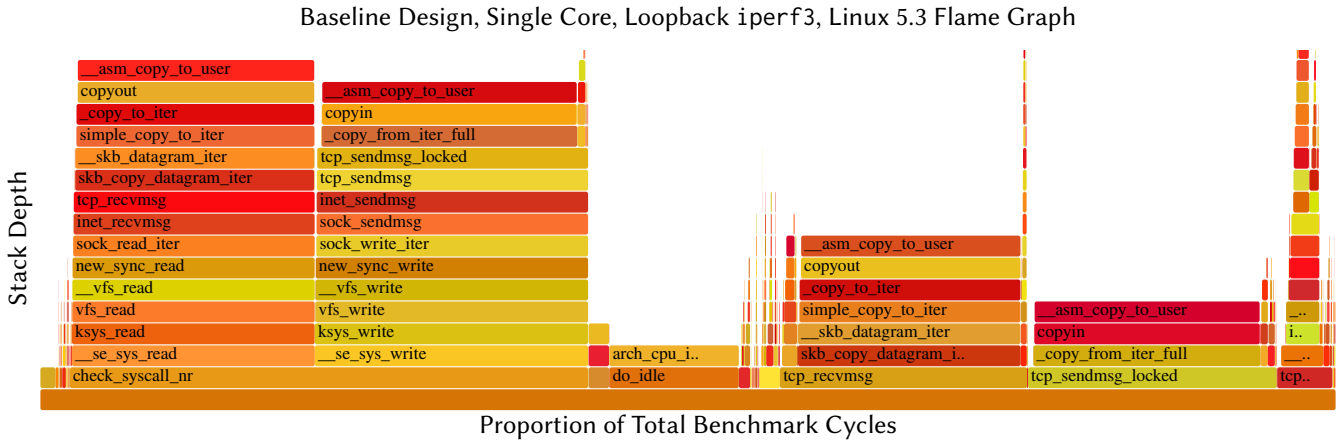

Figure 2. Flame graph[1] for Baseline, Single Core, Loopback on Linux 5.3. This flame graph shows that as a percentage of overall time spent in the workload, the `__asm_copy_{to,from}_user` function in Linux dominates runtime. Furthermore, the userspace `iperf3` code consumes a negligible amount of time (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.

worthwhile—we see performance improvements or similar performance across-the-board. Consequently, going forward we will only analyze the system running Linux 5.3.

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 Section 4.5. 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 2 and 3—we will explore these in-depth in later sections.

4.4.2 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 4 identifies key parts of the NIC microarchitecture. To understand NIC behavior, we add counters with AutoCounter to track several of these potential bottlenecks:

- 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 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

[1] *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.

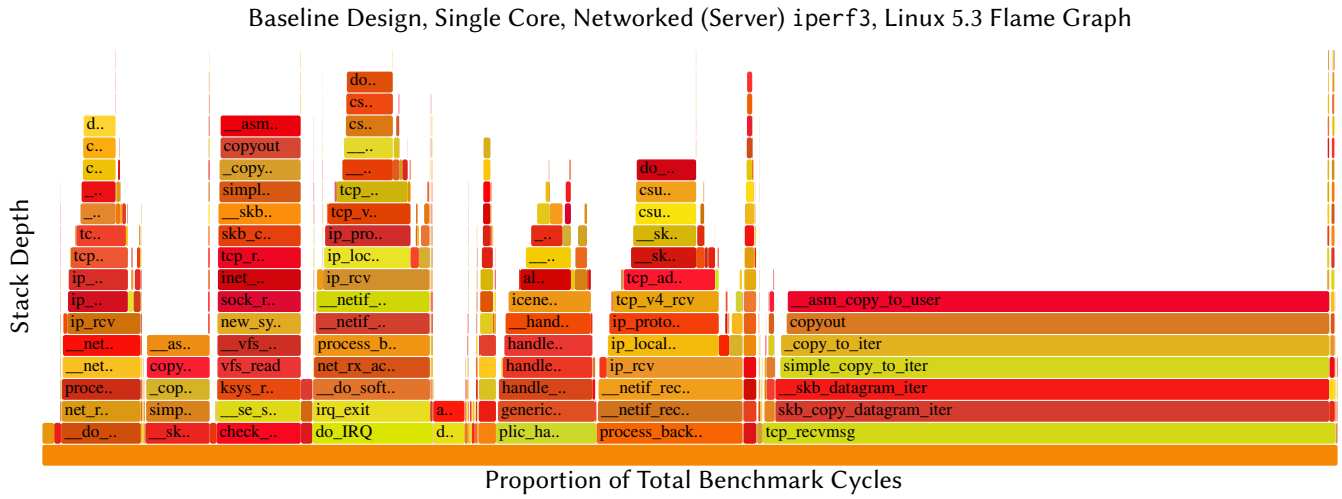

Figure 3. Flame graph for Baseline, Single Core, Networked on Linux 5.3 (server-side). This flame graph shows that while not as severe as the loopback case, in the networked case `__asm_copy_{to,from}_user` still dominates runtime as a percentage of overall time spent during the workload. Due to space 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.

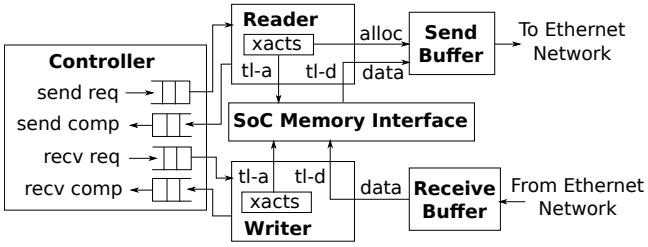

Figure 4. FireChip NIC microarchitecture.

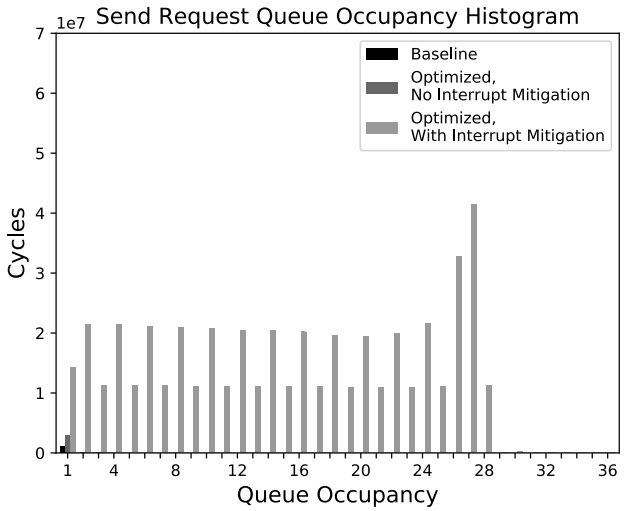

Figure 5. NIC send request queue occupancy analysis collected via AutoCounter performance counter instrumentation.

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 to demonstrate that our software improvements are improving hardware utilization of the NIC.

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

As shown in Table 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 2 and 3 show that one particular function, `__asm_copy_to_user`[2], 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[3] in the Linux kernel that implements user-space to kernel-space memory copies and vice-versa.

[2]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.
[3]https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/riscv/lib/uaccess.S?h=v5.3-rc4

techniques — Accelerating accelerator adoption.

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

```
# 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 6. Realignment code for optimized `__asm_copy_{to, from}_user` implementation.

Naturally, if this system is to be optimized, significant improvements need to be made within this function.

4.5.1 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 underutilizing 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 timing conditions. In Section 4.6, 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 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 Section 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 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 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.

4.5.2 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 [33], 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 3 shows the significant speedups achieved with the integration of Hwacha into

techniques — Accelerating accelerator adoption.

	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 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

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()` [28], 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.

4.6 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 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 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.

4.7 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 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 4.

4.8 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 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 4. The "Optimized, With Interrupt Mitigation" bars in Figure 5 represent NIC queue occupancy once interrupt mitigation is enabled. We see a significant increase in queue occupancy which manifests as improved network performance.

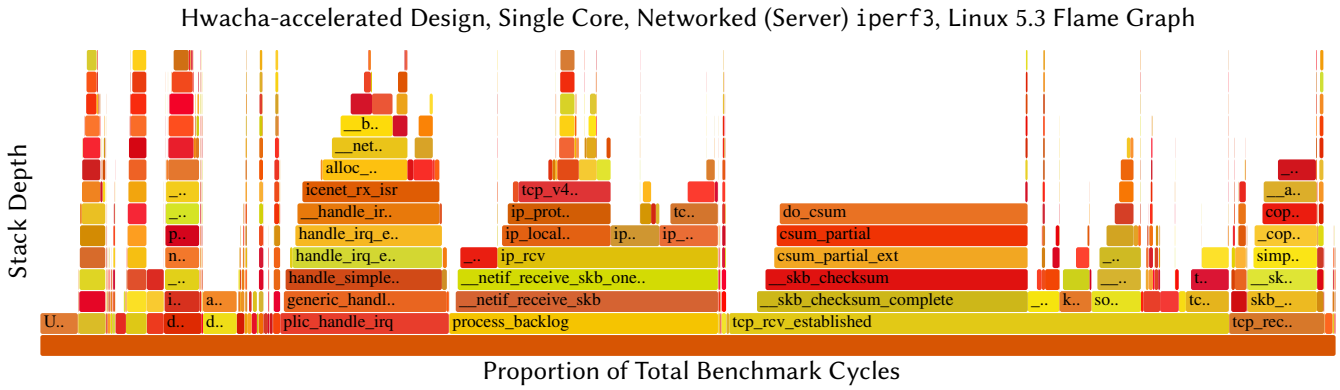

Figure 7. Flame graph for Hwacha-accelerated, Single Core, Networked on Linux 5.3 (server-side). This flame graph shows that a previously-insignificant software routine consumes a significant number of cycles in the networked case once kernel-userspace copies are accelerated: `do_csum`. Due to space 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.

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.[4] 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.

4.9 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 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 [3]. 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 4. In combination with earlier improvements, the system is now capable of 17.5 Gbit/s on `iperf3`.

4.10 Final performance results

Table 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 [8], 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

[4] On RISC-V, this would likely involve extending the Supervisor Binary Interface (SBI) to expose some functionality of high-privilege M-mode timer interrupts, a potentially invasive change.

techniques — Accelerating accelerator adoption.

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

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 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 same networked mode as our case study. Table 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.

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 [11, 19, 27, 38, 39, 42, 44, 49, 52]. In particular, [12] 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 [5, 43], rather than application performance analysis.

Sampling-based methods have also been widely used for profiling [21, 35, 41, 50]. 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 [32] 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. [51] 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 [36].

Some methods to reduce the overhead of software profiling and debugging come in the form of architectural support for software debugging such as IWatcher [53]. 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 [45] at the cost of coarser granularity and non-negligible imprecision.

Prior FPGA-accelerated simulators [16, 17, 40, 47, 48] do not cycle-exactly 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 exceptions of Atlas [48], 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 [24], as well as specialization for network-centric scale-out processors [23].

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 paper also provides instructions for reproducing the experiments in this paper.

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 paper 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, FIRRTL 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 a standard `TracedInstruction` port in the SoC design.

Achieving *Introspection Parity* between FPGA and Software Simulation. Traditionally, FPGA-simulators and open-hardware 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-hardware 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.

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.

Acknowledgments

The information, data, or work presented herein was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0000849. Research was also partially funded by RISE Lab sponsor Amazon Web Services and ADEPT Lab industrial sponsors and affiliates Intel, Apple, Futurewei, Google, and Seagate. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

References

[1] 2018. Kendryte K210 Announcement. https://cnrv.io/bi-week-rpts/2018-09-16.
[2] 2019. FireSim: Easy-to-use, Scalable, FPGA-accelerated Cycle-accurate Hardware Simulation in the Cloud. https://github.com/firesim/firesim.

[3] 2019. Network Maximum Transmission Unit (MTU) for Your EC2 Instance. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/network_mtu.html.

[4] 2019. strace: strace is a diagnostic, debugging and instructional userspace utility for Linux. https://github.com/strace/strace.

[5] B. Agrawal, T. Sherwood, C. Shin, and S. Yoon. 2008. Addressing the Challenges of Synchronization/Communication and Debugging Support in Hardware/Software Cosimulation. In *21st International Conference on VLSI Design (VLSID 2008)*. 354–361. https://doi.org/10.1109/VLSI.2008.74

[6] Krste Asanović, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley.

[7] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. 1212–1221. https://doi.org/10.1145/2228360.2228584

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

[9] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (March 2017), 48–54. https://doi.org/10.1145/3015146

[10] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, and Krste Asanović. 2019. FASED: FPGA-Accelerated Simulation and Evaluation of DRAM. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'19)* (Seaside, CA, USA) *(FPGA '19)*. ACM, New York, NY, USA, 10. https://doi.org/10.1145/3289602.3293894

[11] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. 2006. The M5 Simulator: Modeling Networked Systems. *IEEE Micro* 26, 4 (July 2006), 52–60. https://doi.org/10.1109/MM.2006.82

[12] N. L. Binkert, L. R. Hsu, A. G. Saidi, R. G. Dreslinski, A. L. Schultz, and S. K. Reinhardt. 2005. Performance analysis of system overheads in TCP/IP workloads. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 218–228. https://doi.org/10.1109/PACT.2005.35

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

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

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

[16] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. 2007. 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)*. IEEE Computer Society, Washington, DC, USA, 249–261. https://doi.org/10.1109/MICRO.2007.36

[17] Eric S. Chung, Michael K. Papamichael, Eriko Nurvitadhi, James C. Hoe, Ken Mai, and Babak Falsafi. 2009. ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 2, 2, Article 15 (June 2009), 32 pages. https://doi.org/10.1145/1534916.1534925

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

[19] S. De Pestel, S. Van den Steen, S. Akram, and L. Eeckhout. 2019. RPPM: Rapid Performance Prediction of Multithreaded Workloads on Multicore Processors. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 257–267. https://doi.org/10.1109/ISPASS.2019.00038

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

[21] Lieven Eeckhout. 2010. Computer Architecture Performance Evaluation Methods. *Synthesis Lectures on Computer Architecture* 5, 1 (2010), 1–145. https://doi.org/10.2200/S00273ED1V01Y201006CAC010 arXiv:https://doi.org/10.2200/S00273ED1V01Y201006CAC010

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

[23] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. 2014. A Case for Specialized Processors for Scale-Out Workloads. *IEEE Micro* 34, 3 (May 2014), 31–42. https://doi.org/10.1109/MM.2014.41

[24] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[25] Brendan Gregg. 2016. The Flame Graph. *Commun. ACM* 59, 6 (May 2016), 48–57. https://doi.org/10.1145/2909476

[26] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 209–216. https://doi.org/10.1109/ICCAD.2017.8203780

[27] M. Jahre and L. Eeckhout. 2018. GDP: Using Dataflow Properties to Accurately Estimate Interference-Free Performance at Runtime. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 296–309. https://doi.org/10.1109/HPCA.2018.00034

[28] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. ACM, New York, NY, USA, 158–169. https://doi.org/10.1145/2749469.2750392

[29] 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 Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. 2018. FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 29–42. https://doi.org/10.1109/ISCA.2018.00014

[30] 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 Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. 2019. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the

Public Cloud. *IEEE Micro* 39, 3 (May 2019), 56–65. https://doi.org/10.1109/MM.2019.2910175

[31] D. Kim, C. Celio, S. Karandikar, D. Biancolin, J. Bachrach, and K. Asanović. 2018. 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)*. 76–764. https://doi.org/10.1109/FPL.2018.00021

[32] Hyong-youb Kim and Scott Rixner. 2005. *Performance characterization of the FreeBSD network stack*. Technical Report.

[33] Yunsup Lee, Colin Schmidt, Albert Ou, Andrew Waterman, and Krste Asanović. 2015. *The Hwacha Vector-Fetch Architecture Manual, Version 3.8.1*. Technical Report UCB/EECS-2015-262. EECS Department, University of California, Berkeley.

[34] Y. Lv, B. Sun, Q. Luo, J. Wang, Z. Yu, and X. Qian. 2018. CounterMiner: Mining Big Performance Data from Hardware Counters. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 613–626. https://doi.org/10.1109/MICRO.2018.00056

[35] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. 1993. Effectiveness of Trace Sampling for Performance Debugging Tools. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Santa Clara, California, USA) *(SIGMETRICS '93)*. ACM, New York, NY, USA, 248–259. https://doi.org/10.1145/166955.167023

[36] John D. McCalpin. 2018. 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* (Dallas, Texas) *(SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 18, 13 pages. https://doi.org/10.1109/SC.2018.00021

[37] Tipp Moseley, Neil Vachharajani, and William Jalby. 2011. Hardware Performance Monitoring for the Rest of Us: A Position and Survey. In *8th Network and Parallel Computing (NPC) (Network and Parallel Computing)*, Erik Altman and Weisong Shi (Eds.), Vol. LNCS-6985. Springer, Changsha,, China, 293–312. https://doi.org/10.1007/978-3-642-24403-2_23 Part 8: Session 8: Microarchitecture.

[38] I. Moussa, T. Grellier, and G. Nguyen. 2003. Exploring SW performance using SoC transaction-level modeling. In *2003 Design, Automation and Test in Europe Conference and Exhibition*. 120–125 suppl. https://doi.org/10.1109/DATE.2003.1186682

[39] U. Y. Ogras and R. Marculescu. 2007. Analytical Router Modeling for Networks-on-Chip Performance Analysis. In *2007 Design, Automation Test in Europe Conference Exhibition*. 1–6. https://doi.org/10.1109/DATE.2007.364440

[40] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer. 2011. HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 406–417. https://doi.org/10.1109/HPCA.2011.5749747

[41] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for Accurate and Efficient Simulation. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (San Diego, CA, USA) *(SIGMETRICS '03)*. Association for Computing Machinery, New York, NY, USA, 318–319. https://doi.org/10.1145/781027.781076

[42] Kishore Punniyamurthy, Behzad Boroujerdian, and Andreas Gerstlauer. 2017. GATSim: Abstract Timing Simulation of GPUs. In *Proceedings of the Conference on Design, Automation & Test in Europe* (Lausanne, Switzerland) *(DATE '17)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 43–48. http://dl.acm.org/citation.cfm?id=3130379.3130390

[43] J. A. Rowson. 1994. Hardware/Software Co-Simulation. In *31st Design Automation Conference*. 439–440. https://doi.org/10.1109/DAC.1994.204143

[44] Jürgen Schnerr, Oliver Bringmann, Alexander Viehl, and Wolfgang Rosenstiel. 2008. High-performance Timing Simulation of Embedded Software. In *Proceedings of the 45th Annual Design Automation Conference* (Anaheim, California) *(DAC '08)*. ACM, New York, NY, USA, 290–295. https://doi.org/10.1145/1391469.1391543

[45] N. Sehatbakhsh, A. Nazari, A. Zajic, and M. Prvulovic. 2016. Spectral profiling: Observer-effect-free profiling by monitoring EM emanations. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–11. https://doi.org/10.1109/MICRO.2016.7783762

[46] SiFive. 2018. SiFive HiFive Unleashed Getting Started Guide. https://sifive.cdn.prismic.io/sifive/fa3a584a-a02f-4fda-b758-a2def05f49f9_hifive-unleashed-getting-started-guide-v1p1.pdf.

[47] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. 2010. RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors. In *Proceedings of the 47th Design Automation Conference* (Anaheim, California) *(DAC '10)*. ACM, New York, NY, USA, 463–468. https://doi.org/10.1145/1837274.1837390

[48] Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Tesylar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. 2007. A Practical FPGA-based Framework for Novel CMP Research. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '07)*. ACM, New York, NY, USA, 116–125. https://doi.org/10.1145/1216919.1216936

[49] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. 2015. GPGPU performance and power estimation using machine learning. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 564–576. https://doi.org/10.1109/HPCA.2015.7056063

[50] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. 2003. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. *SIGARCH Comput. Archit. News* 31, 2, 84–97. https://doi.org/10.1145/871656.859629

[51] T. Yoshino, Y. Sugawara, K. Inagami, J. Tamatsukuri, M. Inaba, and K. Hiraki. 2008. Performance optimization of TCP/IP over 10 Gigabit Ethernet by precise instrumentation. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 1–12. https://doi.org/10.1109/SC.2008.5215913

[52] Xinnian Zheng, Lizy K. John, and Andreas Gerstlauer. 2016. Accurate Phase-level Cross-platform Power and Performance Estimation. In *Proceedings of the 53rd Annual Design Automation Conference* (Austin, Texas) *(DAC '16)*. ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/2897937.2897977

[53] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. 2004. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture* (München, Germany) *(ISCA '04)*. IEEE Computer Society, Washington, DC, USA, 224–. http://dl.acm.org/citation.cfm?id=998680.1006720

techniques — Accelerating accelerator adoption.

A Artifact Appendix

A.1 Abstract

This artifact appendix describes how to reproduce results demonstrated earlier in this paper by running FirePerf/FireSim simulations on Amazon EC2 F1 instances.

A.2 Artifact check-list (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.3 Description

A.3.1 How delivered. A version of FirePerf that reproduces the results in this paper 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.

A.3.2 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.

A.3.3 Software dependencies. Installing `mosh` (https://mosh.org/) on your local machine is highly recommended for reliable access to EC2 instances.

A.4 Installation.

First, follow the instructions on the FireSim website (https://docs.fires.im/en/1.6.0/Initial-Setup/index.html) to create a manager instance on AWS. 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.5 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 paper (the 1.67 Gbit/s number in Table 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:

techniques — Accelerating accelerator adoption.

```
    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.6 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.

A.6.1 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 1 of the paper (with a result of 1.67 Gbits/s).

A.6.2 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 3 , since it is generated from your run of the same workload.

A.6.3 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 in the paper, 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.

A.7 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 paper, encoded in the configuration files included in the artifact. Regenerating the supplied FPGA images is also possible, by running `firesim buildafi` in `~/fireperf/deploy/`.

A.7.1 Extended Benchmarks. We also include a script in the artifact to reproduce the other FirePerf workloads in the paper, covering Tables 1, 2, 3, and 4 and Figures 2, 3, 7, and 5 from the paper.

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 1, 2, 3, and 4, populated with bandwidth results from this run.
- `generated-flamegraphs/`: Contains newly generated PDFs for the flame graphs in our paper: Figures 2, 3, and 7.
- `nic_queue_send_req.pdf`: A newly generated version of Figure 5.

A.8 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