Out-of-Band Userspace Profiling in FireSim

Reza Sajadiany
Department of EECS
UC Berkeley
rezasjd@berkeley.edu

Raghav Gupta
Department of EECS
UC Berkeley
raghavgupta@berkeley.edu

Alex Hao
Department of EECS
UC Berkeley
boyuhao@berkeley.edu

***Abstract*—FirePerf is a set of out-of-band hardware/software performance profiling tools integrated into FireSim, a hardware-accelerated cycle-accurate simulation platform running on cloud FPGAs.**

FirePerf provides high-fidelity introspection into the call stack and system behavior, through which it directly exposes bottlenecks and points of possible improvement (in both hardware and software), allowing one to quickly optimize programs without deep domain-specific knowledge of the underlying logic. FirePerf does not impose any profiler overhead on the target program and thus does not perturb the behavior of the target program. Combined with its high fidelity, it can show issues not discoverable by traditional software profilers. It also allows one to target specific behavior and not deal with the massive volumes of detailed data provided by waveforms. Another critical advantage of FirePerf is that it can be deployed early in the hardware development cycle to enable hardware improvements and not just software optimizations to improve the performance of target applications.

The original FirePerf infrastructure does not distinguish between different userspace applications, so it can only profile the kernel and system-level services like I/O. In this work, we extend FirePerf to profile userspace programs. This allows for a wide range of user applications to be profiled and optimized with FirePerf. It also allows studying boundary behavior between the Linux kernel and userspace applications.

I. Introduction

With the slowdown of Moore's Law and Dennard Scaling in silicon, hardware designers have been encouraged to focus on domain-specific architectures that interact with general-purpose CPUs. Designers spend countless simulation cycles in verifying and optimizing their designs to get better performance, power, and area metrics. The trade-offs to consider in iterative design, simulation, and verification cycles can completely change the behavior of the overall system in terms of performance, power consumption, and area in silicon. As design cycles consume long periods of the designer's time, early performance analysis of systems is imperative. Early design space exploration of the overall system, the software stack, and hardware performance is essential to converge to a highly optimized system.

FirePerf [11] is a performance analyzer implemented as part of the FireSim [10] toolchain that allows for analysis and optimizations at such granularity. FireSim maps a design onto cloud FPGAs and runs provided workloads on the design. The advantages of such simulation are the frequency of the simulation, the high visibility of signals, and the quick iteration of simulations. Being part of the infrastructure around the simulated design, FirePerf can gather real-time instruction traces as the simulation runs.

Prior to this work, FirePerf was only able to recognize kernel instructions running on the machine. Realizing the point at which the User-Kernel boundary is crossed can give us more information on the behavior of the software stack, the underlying hardware, and the system as a whole. The original implementation uses the TraceRV bridge to physically gather information such as the program counter (PC) and cycle count for each retiring instruction at the end of the pipeline, through a trace port exposed by the device under test. With PC and cycle count for each instruction, it uses the kernel's DWARF [1] to match the instruction to its corresponding function label. Labeled instructions are then analyzed to unwind the call stack. The unwound call stack is used to generate flame graphs [8] [9], which visualize call stacks as histograms of execution time.

Our work on extending FirePerf to userspace allows us to profile not only the kernel call stack, but userspace programs and kernel-user boundary crossings as well. This work gives a full view of the system behavior under different workloads and can be used to optimize both software and hardware.

Our contributions to FirePerf include:

- Restructuring the code to handle multiple user programs
- Enabling the tracing of pre-kernel machine mode instructions
- Augmenting the TraceRV bridge to gather minimum necessary information for userspace profiling
- Implementing multi-object TraceTrackers in order to support multiple programs
- Implementing a history-based matching algorithm to match instruction tokens to their corresponding executables

This paper is structured as follows: we address the motivation in Section II, background in Section III, related work in Section IV, design and implementation in Section V, results in Section VI, evaluations in Section VII, limitations in Section VIII, next steps in Section IX, and future work in Section X.

II. Motivation

Conventional software profilers, such as Strace [3] and KUtrace [13], are in-band, which means they run alongside the program they are profiling. This is often problematic as the profiler competes with other programs, including its target, over finite hardware resources. So, simply running a profiler can negatively impact the target's performance, leading to inaccurate results. This effect is especially significant when trying to profile, for instance, a CPU-intensive program where the performance almost entirely depends on CPU time, or a disk-intensive workload where available disk access bandwidth is the most crucial metric.

Moreover, since aforementioned profilers are in-band, they must run at the same pace as their target programs. If the profiler processes traces slower than they are produced, and it has no control over the generation of those traces, it will have to sacrifice tracing fidelity and skip some data to prevent overflowing the backlog of traces. Additionally, profilers are known to alter program behavior in more subtle ways, such as influencing the alignment of memory buffers. This can completely mask performance issues from profiling.

In-band profilers fall prey to the computer science equivalent of Heisenberg's famous Uncertainty Principle - the more we attempt to measure a system's execution, the lesser we know about its true behavior.

On the other hand, waveforms of simulated hardware are extremely fine-grained. That level of detail is simply infeasible for system-wide analysis.

To overcome these issues, the only solution is to go out-of-band. The idea is simple: to fully decouple the profiler from the target program, the target must be unaware of the existence of the profiler; to guarantee full fidelity, the profiler must be able to schedule the arrival of traces. In other words, the profiler must run on independent hardware but also maintain full control of the target program's execution at the same time.

However, out-of-band profilers are often infamously slow. The simplest way of building an out-of-band profiler is by attaching it to a software RTL simulator and profiling the simulated workload. In such a setup, a complex workload could take days to run, making it virtually useless in a production environment.

A fast, high-fidelity, and out-of-band profiler is needed. In the following sections, we present our solution to this problem.

III. Background

A. FireSim

FireSim is an open-source cycle-accurate FPGA-accelerated hardware simulation platform running on AWS F1 instances. FireSim wraps a given hardware RTL design in the FireSim RTL simulation infrastructure and emulates the entire package on FPGAs, which is supported by a manager system that monitors and controls the simulation process [10].

With this approach, we are able to run simulations at orders-of-magnitude higher frequencies than pure software simulations, such as Verilator or Verilog Compiler Simulator (VCS). Using FireSim, we can boot Linux and run user programs on custom hardware at an interactive speed.

As an inherent property of RTL simulators, FireSim offers full visibility into the hardware behavior for the purpose of debugging a hardware design. By making necessary changes to the infrastructure, we can let FireSim monitor any points of interest in the simulated hardware.

B. FirePerf

FirePerf is a high-fidelity performance profiling tool built on top of FireSim. It consists of TraceRV and AutoCounter, and the former is explained in greater depth in Section III.C.

FirePerf leverages useful features offered by FireSim, such as fast simulation speed, full visibility of hardware states, and total control over simulation flow. It allows us to monitor hardware behavior and gather data from FPGAs without perturbing the simulated workload, making it an out-of-band profiler. [11]

C. TraceRV

As a fundamental component of FirePerf, TraceRV consists of a bridge and a real-time call stack unwinder. The bridge exists at the boundary of hardware/software and is responsible for transmitting instruction traces from the FPGA simulation to software on the host system. The call stack unwinder, or the TraceTracker, runs on the host and processes collected traces on-the-fly to reconstruct the entire call stack, which is eventually used to generate a flame graph. [11]

D. Kernel Profiling

In the absence of Kernel Address Space Layout Randomization, the position of the kernel in virtual memory is known and fixed. However, the same is not true for userspace programs. Due to the challenges of userspace profiling, previous work on TraceRV can only be used to profile the kernel. In order to profile the stream of instructions running on the device under test, TraceRV uses PC and cycle count from the bridge. In its constructor, TraceRV first parses the kernel DWARF file and extracts necessary information such as the base address of the kernel executable and information per instruction as an array. During the simulation run, TraceRV creates a TraceTracker object that extracts instructions from the DWARF by subtracting the kernel base address from the collected PC (the addresses from the DWARF should match the virtual address from the TraceRV bridge). Indexing into the parsed array using this calculated address returns an "Instr" object corresponding to the instruction. This Intsr object provides all relevant metadata such as the name of the function that this instruction belongs to and whether this instruction is a call site or a function entry point. The TraceTracker then uses this information to re-create the call stack while the simulation is running. The product of the TraceTracker is written into a file for the flame graph generation flow. [11]

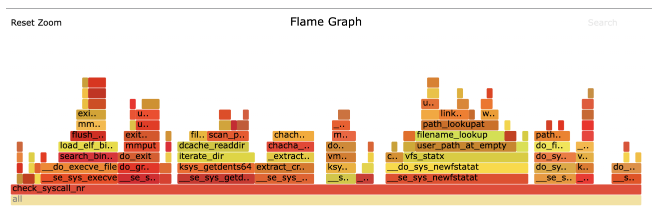

Fig. 1. Example Kernel flame graph. This is generated with the original implementation of FirePerf, and it shows part of the Linux booting process

E. flame graph Genereation

To obtain a hierarchical visualization of system behavior after profiling, TraceRV uses an open-source flame graph generation tool [8]. A flame graph represents a call stack as a histogram of execution time. It is stored as a Scalable Vector Graphic (svg) file. It is interactive when opened in a browser tab, allowing the user to zoom in and search for a particular function. Figure 1 shows an example kernel-specific flame graph.

IV. Related Work

The following tools are generally used to collect information required for flame graph generation.

A. Strace

Strace [3] is a Linux environment command-line tool that performs performance monitoring for system calls and outputs information for troubleshooting bottlenecks. Strace can be used to generate system call stacks. Each line in the output of Strace gives information on the system call function, the arguments passed in, and the return value. Strace has the ability to keep track of system calls from multiple threads at the same time. Strace is often used for profiling and measuring the system-level performance.

B. KUTrace

KUTrace [13] is a low-overhead Linux tracing tool used on commercial multi-processors. It focuses on the kernel-user boundary behavior by tracing every transition between kernel-mode and user-mode executions. It is commonly used to find bottlenecks in databases, file system transactions, and more. KUtrace supports traditional server architectures as well as ARM and RISC-V architectures. Although it profiles the processor dynamics, it does not show any fine-grained timings of either user or kernel routines.

C. Perf

Perf [12] is a Linux system performance analyzer that can trace and profile the kernel and user programs. It inserts trace points in code and inspects performance counters in the CPU hardware, such as instructions executed, cache misses, and branch mispredictions. Perf is easy to install and simple to use, but it only does light profiling: Perf does profiling by sampling at fixed intervals or at inserted triggers, so it is nowhere near full fidelity.

V. Design and Implementation

At a high level, the system is divided into a hardware side and a software side, connected by the TraceRV bridge, as shown in Figure 2. The hardware side involves a standard FireSim setup simulating an instrumented Rocket Chip [5] on AWS F1 FPGA instances. The chip runs a relatively-simple in-order RV64IMAFDC Rocket core; however, the implementation below would also work on out-of-order cores such as BOOM [7] because the mechanism only reuqires in-order commit of instructions. The software side includes TraceRV running on the host system, which assigns a stack unwinder, i.e. a TraceTracker, to each user program so it can be tracked independently. The bridge delivers instruction data from the simulation to TraceRV, which, through a novel matching algorithm, multiplexes the instruction to its corresponding TraceTracker. The output of each TraceTracker is sent to the flame graph generator for the final product.

In order to profile userspace programs, we would need more information from the execution stream to realize which executable an instruction packet corresponds to. As shown in Figure 5, the PC is in virtual memory space, and our end goal is mapping the PC back to the correct executable's correct instruction address. As we must preserve the out-of-order property, the major challenge here is to maximize the deduced data while minimizing perturbation in the OS kernel and the simulated core.

After much consideration, we decided to pursue completely out-of-band userspace profiling without any software instrumentation. We elected to only modify the Rocket Chip's trace port and the TraceRV bridge on the hardware RTL side, and do the majority of the work in software post-processing. The RTL changes made to the TraceRV bridge were done using the Chisel Hardware Description Language [6].

In the following subsections, we describe the design and implementation details, organized by various aspects of the system.

A. Introspection and Transmission

We posit that the following register values give us the minimum amount of information required to identify instructions in a deterministic manner, and justify our choices as follows:

PC: The PC is in virtual address space, as used internally by the core, and is already part of the bridge in the original FirePerf implementation. It is still used the same way to profile the kernel process, but we must map it back to user executables to profile userspace processes.

Cycle: The cycle count is the second of the two original members of the TraceRV bridge. It is used to record the execution time, in cycles, of matched functions to generate flame graphs at the end.

Instruction: The instruction bits are used for matching traces from the bridge to the exact instructions in source executables.

SATP: The SATP register, or Supervisor Address Translation and Protection register, holds the pointer to the base page table for each process, as described in the RISC-V Privileged

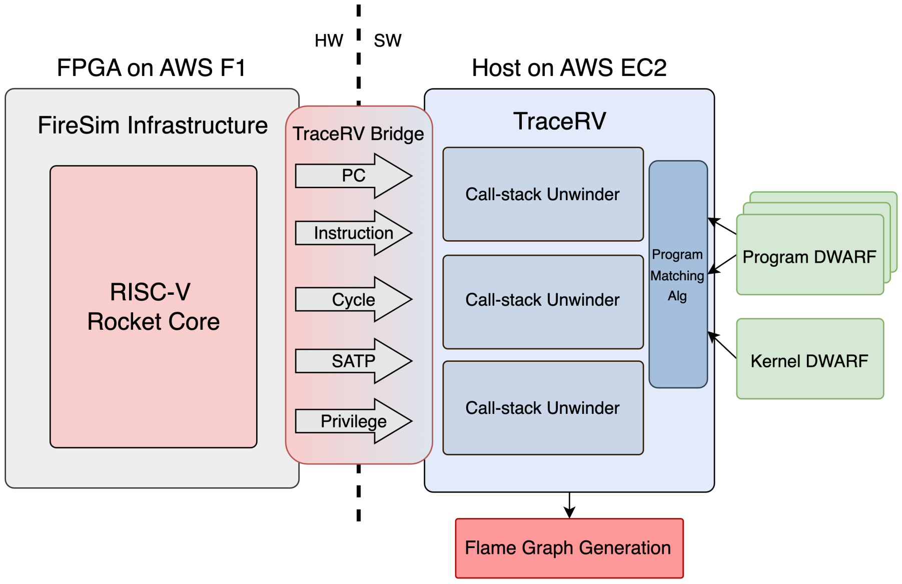

Fig. 2. System Architecture Overview. FireSim FPGA infrastructure is depicted on the left, and the TraceRV bridge directs each instruction token to the TraceRV C++ code for profiling.

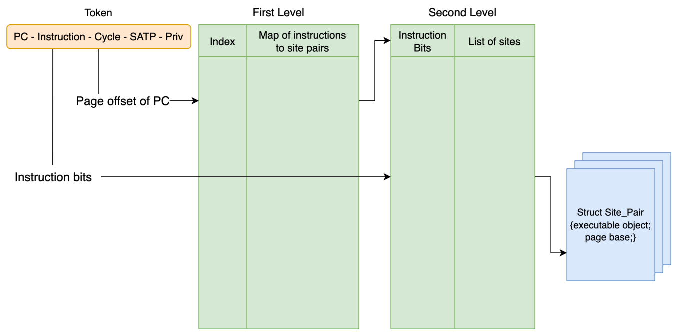
Token
PC - Instruction - Cycle - SATP - Priv
First Level
Index
Map of instructions to site pairs
Second Level
Instruction Bits
List of sites
Page offset of PC
Instruction bits
Struct Site_Pair {executable object; page base;}

Fig. 3. The Two-Level Map Data Structure. It stores possible sites based on page offset and instruction bits. Using the page offset of the PC, we access the first level to get the second-level map; using the instruction bits, we index into the second level to get a list of possible sites. A site pair contains both an executable and a page number.

Specification [4]. As it is highly unlikely for processes to share base page tables, we use SATP as a proxy for process ID (PID) to uniquely identify processes.

Privilege: The privilege mode register indicates the processor state in terms of execution privileges. In RISC-V, privilege modes are assigned in ascending order, i.e., 0: user, 1: supervisor (kernel), 2: hypervisor, 3: machine mode [4].

After our modifications, the trace port sends PC, instruction bits, cycle count, SATP, and privilege mode over the bridge as one 512-bit packet, called a token, per committed instruction from the core, as shown in Figure 2.

B. Source Binary Pre-Processing

For each target user program, we require 1) a DWARF file and 2) a hex dump of the .text section of the executable containing raw instructions. The DWARF file contains debugging information, created at compilation time, that is used to derive function names to construct human-readable flame graphs. The mechanisms to process and use DWARF files have been retained from the original FirePerf implementation.

The hex file is more interesting and is unique to this userspace extension. Before the simulation starts, TraceRV reads through the provided hex files and constructs a single map based on all hex files. This map is the foundation of our matching algorithm, thus, its structure and correctness are crucial.

By excluding position-independent code, we expect a contiguous, one-to-one, page-aligned mapping between virtual addresses and executable addresses of instructions. (Relaxations of this constraint are discussed in later sections. However, this has restricted us to statically-built executables currently.) Thus, we know the page offset bits of an instruction's PC and its executable address are the same, and we must infer the mapping of page numbers (upper PC bits).

We construct a two-level map, as shown in Figure 3, where the first level is indexed by the page offset of the instruction, and the second level is indexed by the instruction bits, eventually leading to a list of <executable, executable page number> pairs that we call possible sites.

When given a token from the device under test, we traverse this map as follows:

1) page offset bits of the PC serve as the first level index
2) instruction bits serve as the second level index

We now have a list of possible sites where each possible site uniquely identifies an instruction at a specific offset in a specific executable.

C. Runtime Token Processing

At runtime, committed tokens are streamed to TraceRV over the bridge, and we use a novel algorithm to match each token

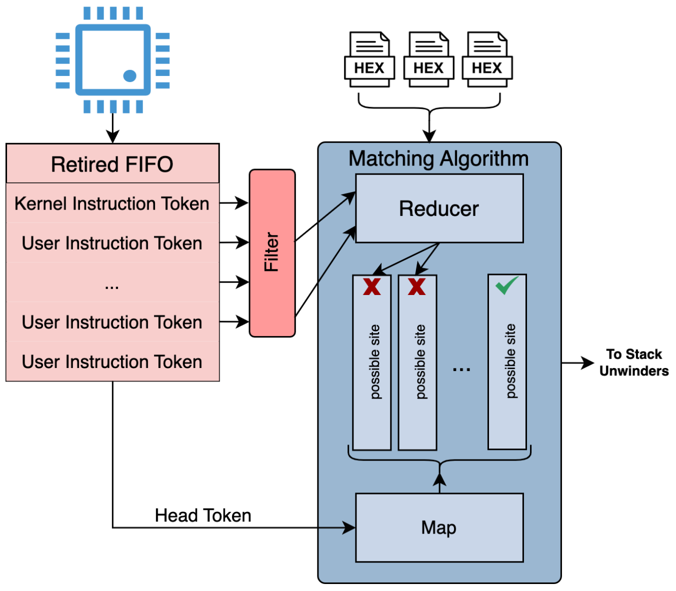

Fig. 4. The Matching Algorithm. A to-be-matched token first traverses the map to get a list of possible sites, which is then reduced to a single site to produce a match.

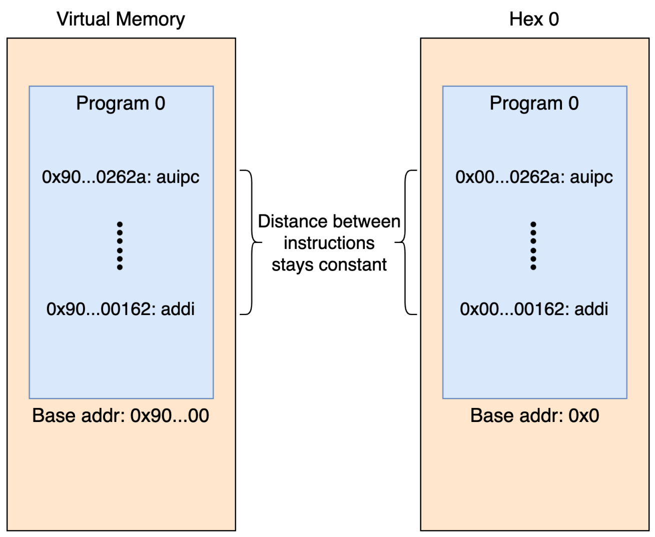

Fig. 5. Memory Layout Assumption. The matching algorithm is based on the assumption that the distance between instructions from the same executable stays constant throughout any memory remapping or manipulation done by the kernel.

with the executable that it came from.

Our matching algorithm uses a combination of information provided by the bridge, DWARF files, and hex files to match each instruction token to the right function label. Each token {PC, Instruction, Cycle, SATP, PRIV} received from the bridge first enters a FIFO buffer in commit order, as shown in Figure 4. On each tick of the clock, the token at the head of the buffer is popped for matching. We need multiple instructions to correctly identify the execution of any single user program. The retired buffer allows us to create a local history of instructions for this purpose.

Using the page offset of the token's PC (PC % PAGE_SIZE) as well as the token's instruction bits, we index into the map data structure depicted in Figure 3 to get a list of possible sites. At this stage, there may be zero or more possible sites.

- **Zero possible sites:** We are unable to find this instruction, i.e., it does not exist in any of our source executables.
- **One possible site:** There may exist programs that we do not have source executables for, so a unique site does not necessarily mean a correct site, and we must still try to match a longer chain of instructions to confirm this site.
- **Multiple possible sites:** Need to use instructions from the buffer to form a longer chain of instructions and thus reduce to a unique site.

Most assembly functions contain template-like prologue and epilogue sections, that share similar instructions and can lead to ambiguous matches. Therefore, regardless of whether we are confirming a single possible site or reducing multiple possible sites into one matched site, we must form a chain with additional tokens and try to match that chain to a sequence of instructions in the executable address space.

To form the aforementioned chain, we need to use tokens that come from the same userspace program. The length of this chain is decided by a variable matching depth. Thus, shorter matching depths reduce successful matches whereas longer matching depths increase compute overhead. We gather tokens from the buffer that do not belong to the kernel address region and have the same SATP value as the original token we started with.

Our need for a chain of instructions for identification influenced our choice of retired buffer size. We ran BusyBox commands on a target, collected traces, and processed them to identify the average gap between n instructions from the same process. This allowed us to settle on a buffer size of 2048 entries.

For each possible site found in the previous step for the original, we use the chain to match against that site. We know the distance between two instructions from the same program in virtual memory (on the target). Therefore, there must be mappings for these two instructions from the same executable at the same distance in our map data structure. We check if such a mapping at the required distance exists and reduce the number of possible sites,

In the case that all tokens in the chain match with the same site, we can conclude that the original token is indeed from that specific site. We then add this token to a list of matched sites. At the end of the algorithm, we inspect the contents of this list. There are a few cases depending on the number of matched sites.

- **Zero matched site:** All possible sites were eliminated in the reduction stage, so this instruction does not exist in any of our source executables. This is a matching failure.
- **One matched site:** We have confirmed a unique match. Matching successful!
- **Multiple matched sites:** We are not unable to uniquely map this token to a specific instruction in our search

space with the information available, so this is a matching failure.

With the executable address of a matched instruction, we can index into the array of DWARF information for that executable. This information is fed into the TraceTracker corresponding to the matched executable. Each TraceTracker is responsible for one program and unwinds that call stack instruction-by-instruction by noting when a function is entered into or returned from.

Note that our algorithm only succeeds on unique matches, and doesn't pass ambiguous matches to TraceTrackers.

Matching Algorithm Pseudo Code

```
struct pairs {executable, page_base};
Vector<int, map<int, pairs>> table;
Vector<struct token> buffer;
Token t <- TraceRV Bridge
possible sites :=
    table[t.pc.page_offset][t.inst];
if possible sites.size == 0 then
    no match; return false;
else then
 let matching vec := vector<token>
 for til matching vec.size < MATCH_DEPT {
  if (buffer(i).satp == t.satp) then
    matching vec.add(buffer(i))
  fi
 let Vector<pairs> matched sites;
 for s in all possible sites {
  for m in all matching vec {
    let p = m.pc.page_base -
            t.pc.page_base +
            s.page_base;
    Vector<pairs> v =
     table[m.pc.page_offset][m.inst];
    if (v not contain p and s.bin) then
      break;
    fi
    if (last m in matching vec) then
      matched sites.add(s)
    fi
  }
 }
 if (matched sites.size == 1) then
   match; return true;
 else then
   no match; return false;
 fi
fi
```

D. Optimizations

There are multiple ways to speed up the matching process. We present one we have already implemented and then discuss another to be pursued as a next step.

After each successful match, we propagate the matched executable to all tokens in the retired buffer with the same SATP as the token we just matched, effectively caching the matching result, as shown in Figure 6. However, there exists an edge case where a program may terminate, and a new program may start running, but the two programs may have the same SATP value and both have tokens in the buffer. It must be noted that this is extremely unlikely given the large space of possible SATP values. In this situation, tokens with the same SATP value may actually come from two different executables. Therefore, we cannot fully trust the cached data and must verify them. Fortunately, verifying a cached match is still much faster than running the entire matching algorithm again, so the propagation caching scheme is still valid.

A cached match is verified by traversing the map in Figure 3. To verify a token A, we use A's PC page offset and instruction bits to index into the map, retrieving a list of possible sites. Then we check if A's cached matching executable and page number are among those possible sites. If yes, A is verified and can go on to the stack unwinder; otherwise, A's cached data is invalidated and must go through the entire matching algorithm.

Table I shows the impact of this optimization on the effective FPGA frequency. This result shows that we improve our overall profiling performance by 6% with this optimization.

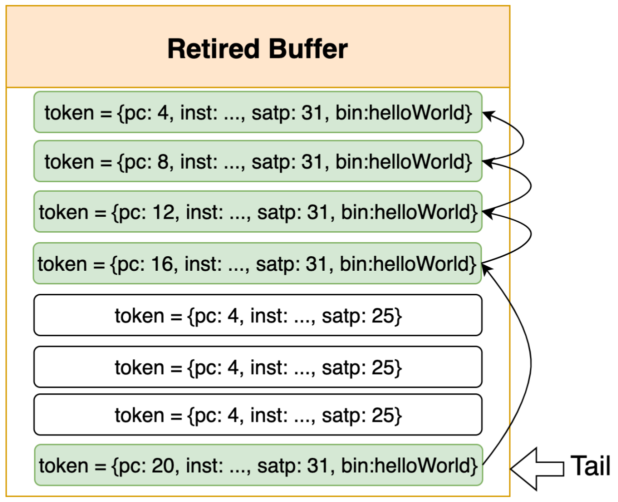
Retired Buffer
token = {pc: 4, inst: ..., satp: 31, bin:helloWorld}
token = {pc: 8, inst: ..., satp: 31, bin:helloWorld}
token = {pc: 12, inst: ..., satp: 31, bin:helloWorld}
token = {pc: 16, inst: ..., satp: 31, bin:helloWorld}
token = {pc: 4, inst: ..., satp: 25}
token = {pc: 4, inst: ..., satp: 25}
token = {pc: 4, inst: ..., satp: 25}
token = {pc: 20, inst: ..., satp: 31, bin:helloWorld}
Tail

Fig. 6. Optimization process starts after matching the tail token. Once the token packet is matched and labeled, other packets which have the same SATP value could be labeled with the same executable. When we see that a to-be-matched token has already been matched due to this optimization, we would check if the matched executable is correct.

TABLE I
FPGA BUILD COMPARISON WITH HOST FREQUENCY OF 90 MHZ, MATCHING DEPTH OF 3, AND RETIRED BUFFER SIZE OF 2048

Optimization level	Effective Frequency (MHz)	FMR
Baseline	16.949	5.310
Optimized	17.961	5.011

This optimization is limited to caching successful match results for instructions with the same SATP in the retired buffer at that time. An extension to this strategy would involve caching the executable - SATP mapping in a separate structure and utilizing the match until that SATP is reused, thus requiring only one round of matching per executable.

VI. Results

As a proof-of-concept, our first attempt to profile a user program was done using a simple synthetic C program: a hello_world program that performs 1 arithmetic operation in main, 2 arithmetic operations in foo, and prints a final string in main. We intended to control the environment so we could isolate the result of our matching algorithm. To do this, we manually ran hello_world in the init() function of the kernel, which causes the kernel to power off immediately after. This is so that we don't pollute the matching traces with extra userspace programs or system services. By doing this, we were able to profile the execution of hello_world. As shown in Figure 7, we can clearly see that hello_world is dominated by the print system call, as expected. This confirmed that our implementation worked in the simplest case. The flame graph generated demonstrated in figure 7 shows that the algorithm correctly labels instructions from the hello_world program and the stack unwinder has successfully created the correct call stack. (foo shows up correctly post stack-unwinding, but is omitted by the flame graph generator as it occupies <0.1 px and cannot be rendered.)

To further verify our implementation, we started a simulation with two userspace programs running normally alongside the kernel. We used the same hello_world program as well as BusyBox, a software suite that contains more than 300 common Linux utilities in a single executable file. As depicted in Figure 8, the flame graphs gives us the information needed to analyze the system performance of BusyBox.

While printf's domination of our hello_world isn't remarkable, the sheer magnitude of cycles consumed is. We added a for loop to foo that accumulated a sum, while still printing once. We observed that foo consumes $O(10^{n+1})$ cycles for 10^n loop iterations. Since printf prints to the system console through a kernel syscall, it is more variable but utilized between $O(10^4)$ and $O(10^5)$ cycles. Thus, through our experiments, we realized the potentially massive cost a single printf can have, and how much arithmetic one could compute in the same time.

To stress-test the design we ran a simulation with two nodes of FPGA instances, that contained network interface cards. We used iperf3 [2], a network profiling tool, and profiled the system running iperf3 client on one node and iperf3 server on the other. This workload also addresses the unobserved userspace behavior of the iperf3 case study in the original FirePerf paper. In Figure 9, the flame graph generated from iperf3 client shows calls to iperf_tcp_send, Nwrite, and libc_write. In Figure 10, the flame graph indicates calls to iperf_tcp_recv, Nread, and libc_read to monitor TCP packets.

The large gap between Nread and libc_read is quite interesting and warrants further investigation, especially into the alignment of buffers and the cost of kernel-user boundary crossings. Additionally, with 96% of iperf_run_server spent in Nread and 52% of iperf_run_server spent in libc_read, one can understand the motivation for kernel-bypass methods in networking.

VII. Evaluations

We evaluated the performance of our profiler by comparing against vanilla FireSim without any tracing and the original kernel-only implementation of FirePerf, by running the same workloads multiple times across different target build/simulation frequencies. As a prerequisite to performing these evaluations, we successfully integrated 'Full FirePerf' into FireSim.

FireSim forces the simulation to pause at points when the surrounding processing environment needs to process the already committed instructions. This is done by forcing the clock signal to be low for some duration. During the time that the clock is being held down, the core, and the logic around the core such as TraceRV port do not receive any additional packets or operations. We use the FPGA Model Ratio or FMR as a metric to indicate how many cycles of actual host FPGAs have been affected by the clock being held down. Actual frequency is determined after a run has finished by using the cycle count from the design. The lower the FMR, the closer the FPGA is running to its initially programmed target frequency. Higher FMR indicates that this simulation's actual frequency is lower than the target frequency.

As we see in Figure 11, the FMR derived for a target build frequency of 90 MHz indicates that:

- FireSim without tracing faces significant synchronization and target console logging overheard
- Kernel FirePerf runs 2.2x longer as it passes data over the trace port/bridge and performs kernel stack-unwinding
- Full FirePerf runs 1.6x longer than Kernel FirePerf and 3.6 longer than Vanilla FirePerf. These effects are unsurprising and are discussed further below.

Our approach for profiling out-of-band has no impact on the target being traced. However, the large amount of data being retrieved from the TraceRV port and its processing (particularly matching) on-the-fly causes the simulation itself to be run at a lower frequency. This is in part due to physical restrictions on the FPGA ports and the maximum frequency at which the bus can operate.

These effects are consistent with the 140 MHz target build frequency shown in Figure 12.

Effective simulation frequency has been shown for 90 MHz in Figure 13 and for 140 MHz in Figure 14 for the three types of builds. While the slowdown of Full FirePerf.

While the slowdown compared to other FireSim simulation variants is apparent, it must be noted that the performance of Full FirePerf is scalable at nearly constant FMR across target build frequencies. Given the much greater introspection made available by Full FirePerf, and its practical viability as a high-fidelity full-system profiler, we believe this performance tradeoff is entirely acceptable.

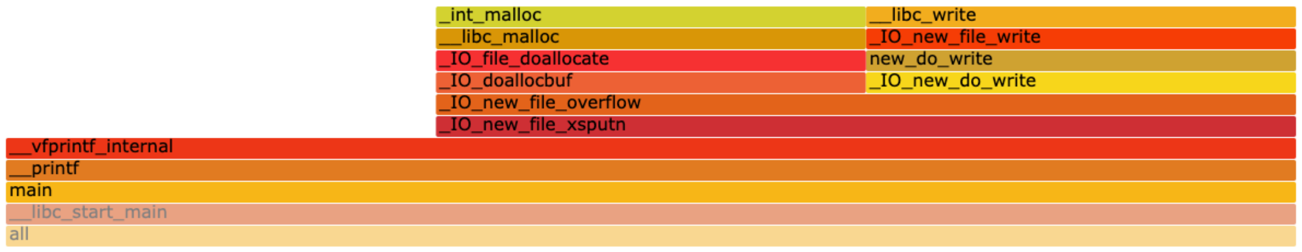

Fig. 7. Profile of a synthetic C program that prints a string to the terminal. The call stack is depicted from the main function to the printf function and system calls.

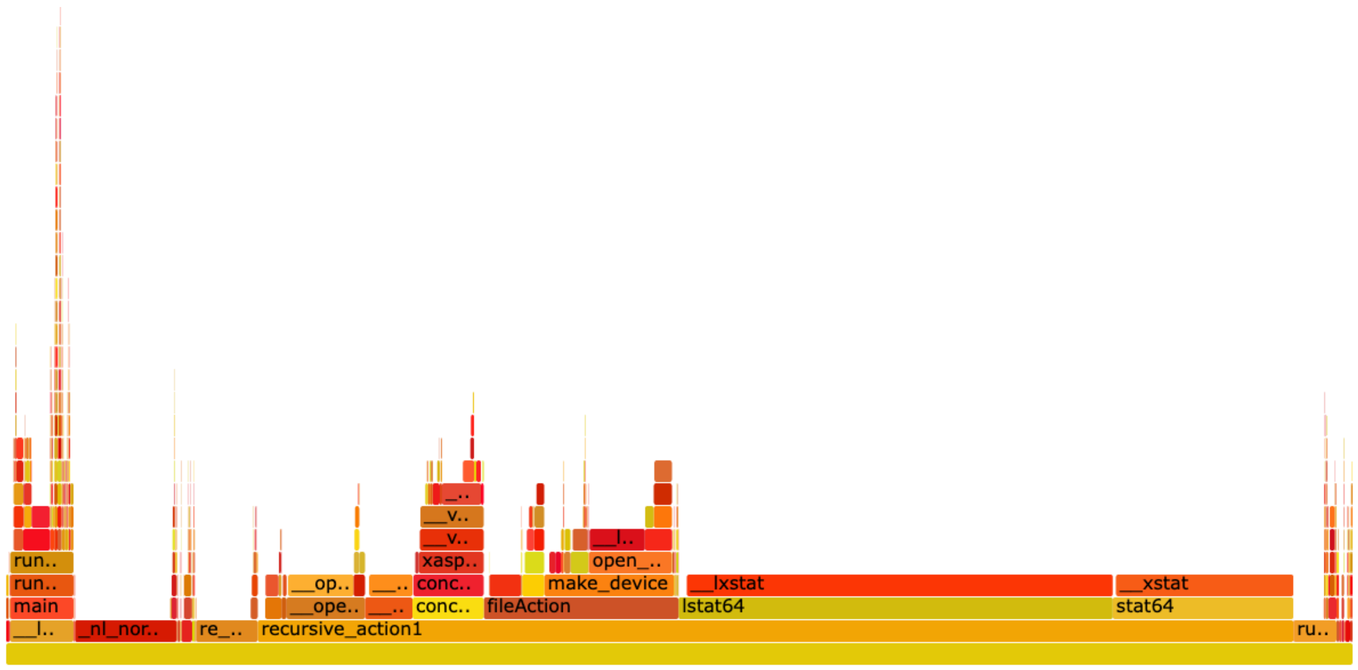

Fig. 8. Profile of BusyBox program which contains commonly-used tools Linux utilities.

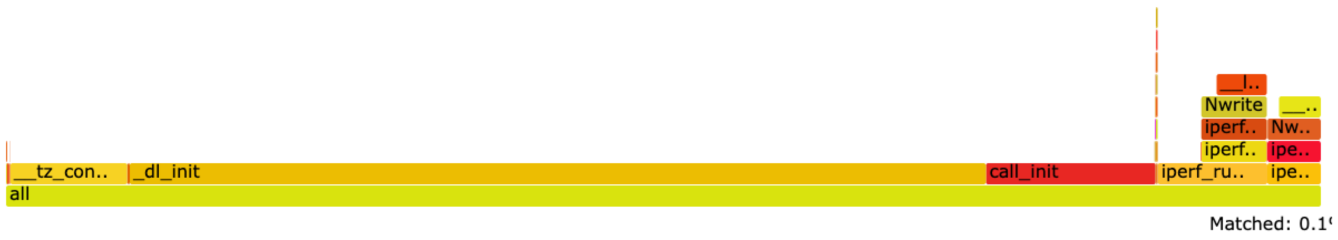

Fig. 9. iperf3 client-side profiling.

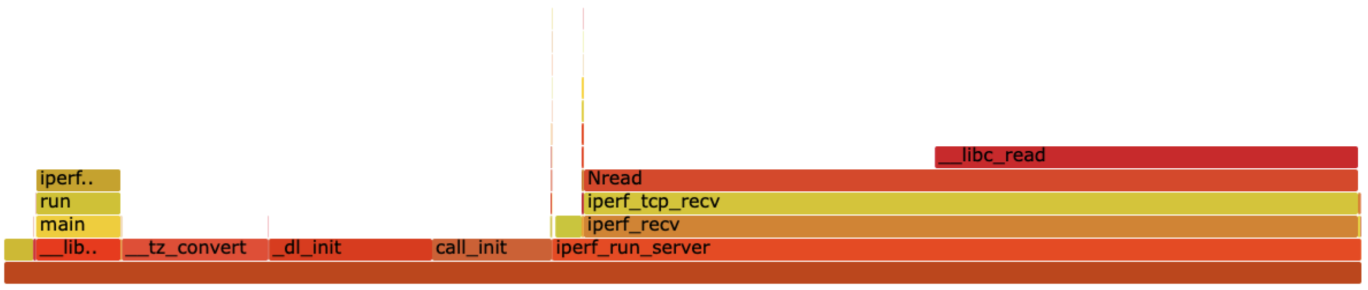

Fig. 10. iperf3 server-side profiling.

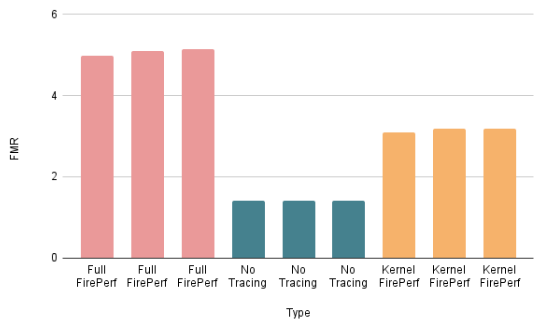

Fig. 11. FPGA Model Ratio VS Type of Simulation For 90 MHz build. Lower FMR is better.

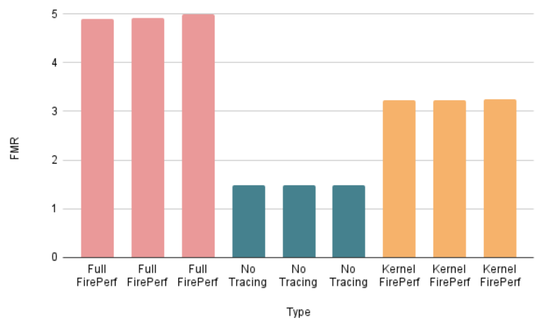

Fig. 12. FPGA Model Ratio VS Type of Simulation for 140 MHz build

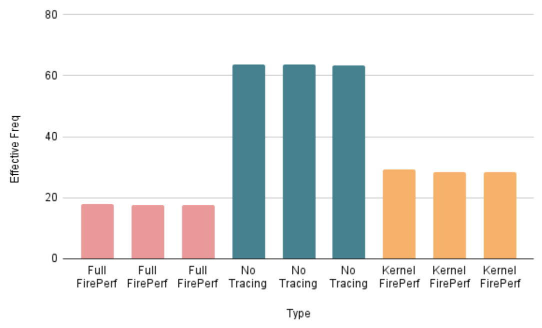

Fig. 13. Effective Simulation Frequency VS Type of Simulation Run for 90 MHz build

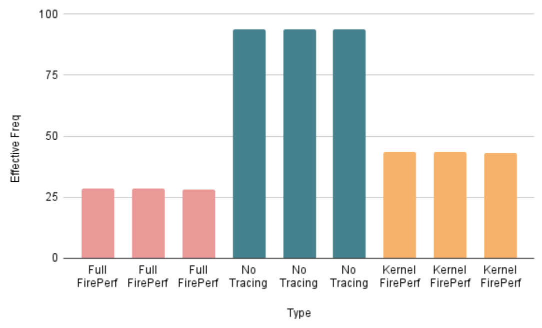

Fig. 14. Effective Simulation Frequency VS Type of Simulation Run for 140 MHz build

VIII. Limitations

Position-Independent Code / Dynamically Linked Libraries: The matching algorithm heavily relies on DWARF and hex files that are generated at compile-time. It expects a contiguous, one-to-one, page-aligned mapping between virtual addresses and executable addresses of instructions. Thus, it expects the distances between instructions to be constant across the virtual and executable address spaces. Various kinds of position independent code break this assumption, requiring us to run statically-linked executables currently. With PIC/DLLs, Shared libraries and position-independent executables are mapped to arbitrary virtual addresses. To account for these challanges, we need to process the .got (Global Offset Table) and .plt (Procedure Linkage Table) and ldd output to acquire a list of shared libraries and their relative offsets. Then, we must instrument the target to reveal actual base address of executables in virtual memory. However, this approach deviates from our philosophy of out-of-band profiling. While the instrumentation required would be much lower than traditional in-band software profilers, it would come at a cost. Additionally, our current page-based matching implementation lays down a strong extensible foundation to add such features in the future.

ASLR is particularly tricky as it re-places and re-orders sections of an executable. Note: The original kernel FirePerf implementation does not support KASLR as it relies on a known fixed location of the kernel in the virtual address space. However, user programs' security while running under a system profiler such as FirePerf is not a top priority as FirePerf exposes many of the secure processor states for the purpose of profiling.

dlopen presents a difficult challenge as it allows arbitrary loading of shared objects at runtime. However, its use is restricted to complex software stacks and can be supported with trigger NOP instructions or a specialized user-level trace support library.

An Extreme Case in Propagation Caching: As discussed in Section V.D, we must verify all cached matches due to a potential same-SATP-different-program scenario.

Yet, there is another edge case within this edge case. The verification procedure checks if the cached site contains the exact instruction bits at the exact offset, but it does not consider aliases: two identical instructions with the same offset may exist in two different pages and/or two different executables. The extreme case is that two programs with the same SATP value can have the same instruction at the same page offset. If this were to happen, our verification procedure will not detect that this new token is from a new program, and thus it will fail in correctness.

Nevertheless, we justify that the same-SATP-different-program scenario is already rare, and the probability of them having identical instructions at the same page offset is even lower. We could handle this extreme case by matching a pattern rather than a single token during verification, but this defeats the entire purpose of propagation caching. The extreme case is so unlikely to happen that it's not worth giving up massive speedup for.

IX. Next Steps

Moving forward, our immediate next step is to support hypervisor profiling and perform a case study on hypervisors running atop the RISC-V Hypervisor Extension. Our current codebase and logical structure already account for the hypervisor; it's just not explicitly supported yet. To perform profiling for the hypervisor mode, we could rely on privilege mode and match the instructions based on the same userspace program DWARF, and hex files. Supporting hypervisor mode would give us information on hypervisor scheduling, system call interface, and host kernel behavior while running the hypervisor. There are many areas that can be optimized in both hardware and software systems to improve the performance of hypervisors. With a full-fidelity and out-of-band profiling infrastructure, software, hardware, and system engineers can effectively address such bottlenecks.

To make profiling using FirePerf more approachable and interactive, we are working towards implementing dynamic real-time flame graphs in FirePerf. FirePerf is able to boot up a Linux kernel and the user is able to run different programs which are all being profiled on-the-fly. We can periodically update flame graphs as the target executes. To do this, we need to create a new flame graph every time due to the underlying generation flow of flame graphs. A polling mechanism can be adopted to check for updates to existing stack-unwinders or the creation of new stack-unwinders. Then, new output files can be generated and the flame graph generation flow can be run on them. We are planning to implement this interactive system and perform a working demo.

Currently, FirePerf generates individual flame graphs for each program. With this setup, it is missing crucial user-kernel boundary crossing information that would otherwise be conveyed by a unified flame graph. In order to achieve this, we can identify context switches into the kernel syscall handler with PC and privilege information. Then, we pause accumulation of cycles on the top of the user stack. When returning from syscalls, we mark tops of stacks for each process and continue accumulating on those stacks for future syscalls from the same process.

X. Future Work

There exist many interesting features to augment such a powerful profiling infrastructure. We details some of them as follows:

Multi-Threaded Program Support: Following the trend in multi-core processors, more and more programs nowadays are multi-threaded. These programs exploit thread-level parallelism because they are highly performance-sensitive, therefore behavior profiling can be extremely useful in such applications. Multi-threaded programs are often complex and hard to reason about during runtime, but a high-fidelity profiler like FirePerf can directly reveal hidden bottlenecks at the lowest level, like unexpected lock contentions. Moreover, multi-threaded programs often try to utilize all available machine resources, so a conventional in-band profiler would not be a good fit as it may impose unwanted pressure on the program.

However, profiling multi-threaded programs is a huge endeavor. By design, the processor does not have any information about which thread is running: multi-thread information is exclusively stored in the kernel as an abstraction, and the hardware simply executes instructions sent by the kernel. Therefore, our current design, which only inspects processor architectural states, isn't sufficient. An ideal design must instrument the OS kernel to send out specific thread information so that the profiler can know which instruction belongs to which thread. Yet, such instrumentation breaks the out-of-band property, as now the kernel must do additional work that has nothing to do with the program being profiled.

Multi-Core Processor Support: Currently, FirePerf treats each core as a separate machine and generates individual flame graphs, which is quite useful if one wants to analyze core-specific behavior, but not so much to analyze program-specific behavior. To achieve the latter, we must identify and track the thread-to-core mapping in order to generate a unified flame graph that correctly reflects the behavior of the program. Therefore, multi-core processor support boils down to the same requirements as multi-threaded program support, which is discussed earlier.

Out-of-Order Core Support: As briefly mentioned in Section V, our architecture theoretically already supports BOOM [7], but in practice, there are limitations that prevent us from running FirePerf on out-of-order processors. The FPGAs on AWS F1 instances have an output bandwidth limit of 512 bits, meaning that they can send out at most 512 bits of data per cycle. Our current implementation uses about half of this bandwidth, but this is only for one instruction; BOOM, or any out-of-order core, can commit many instructions per cycle, which will quickly overflow this 512-bit limit. Therefore, running FirePerf on out-of-order cores requires actual hardware changes on the cloud end.

Acknowledgements

We would like to acknowledge Sagar Karandikar who supported and mentored us during this project. Albert Ou, Jerry Zhao, Abraham Gonzalez, and the rest of the FireSim and Chipyard development team. Professor John Kubiatowicz and Zhuohan Li for providing us with feedback throughout the duration of CS262A. Professor Borivoje Nikolic, Professor Krste Asanović, and Professor Sophia Shao for their guidance in our research efforts at UC Berkeley's SLICE Lab. SLICE Lab technical support and administration for providing us with AWS credits to perform the simulations for this research project.

References

[1] "Dwarf debugging information format committee. 2017. dwarf debugging information format version 5. standard." [Online]. Available: http://www.dwarfstd.org/doc/DWARF5.pdf

[2] "Iperf3: A tcp, udp, and sctp network bandwidth measurement tool." [Online]. Available: https://github.com/esnet/iperf

[3] "strace: strace is a diagnostic, debugging and instructional userspace utility for linux." 2019. [Online]. Available: https://github.com/strace/strace

[4] K. A. Andrew Waterman and J. Hauser, "The risc-v instruction set manual, volume ii: Privileged architecture, document version 20211203," Tech. Rep., December 2021.

[5] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[6] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1216–1225. [Online]. Available: https://doi.org/10.1145/2228360.2228584

[7] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanović, "Boom v2: an open-source out-of-order risc-v core," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2017-157, Sep 2017. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html

[8] B. Gregg, "Flame graph," 2019. [Online]. Available: http://www.brendangregg.com/flamegraphs.html

[9] B. Gregg, "Flamegraph: Stack trace visualizer." 2019. [Online]. Available: https://github.com/brendangregg/FlameGraph

[10] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanović, "FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 29–42. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00014

[11] S. Karandikar, A. Ou, A. Amid, H. Mao, R. Katz, B. Nikolić, and K. Asanović, "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*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 715–731. [Online]. Available: https://doi.org/10.1145/3373376.3378455

[12] A. C. D. Melo, "The new linux perf tools," Tech. Rep., 2010, in Slides from Linux Kongress, Vol. 18.

[13] R. Sites, *Understanding Software Dynamics*, ser. Addison-Wesley professional computing series. Addison Wesley Professional, 2021. [Online]. Available: https://books.google.com/books?id=TklozgEACAAJ