A Case for Accelerating Software RTL Simulation

Scott Beamer
University of California, Santa Cruz

***Abstract*—RTL simulation is a critical tool for hardware design but its current slow speed often bottlenecks the whole design process. Simulation speed becomes even more crucial for agile and open-source hardware design methodologies, because the designers not only want to iterate on designs quicker, but they may also have less resources with which to simulate them. In this work, we execute multiple simulators and analyze them with hardware performance counters. We find some open-source simulators not only outperform a leading commercial simulator, they also achieve comparable or higher instruction throughput on the host processor. Although advanced optimizations may increase the complexity of the simulator, they do not significantly hinder instruction throughput. Our findings make the case that there is significant room to accelerate software simulation and open-source simulators are a great starting point for researchers.**

Introduction

Simulation is an invaluable tool for hardware design due to the high financial and temporal costs of fabricating a chip. Simulation is used in a variety of settings, whether it be development, design space exploration, debugging, verification, or validation. To improve simulation speed, common techniques include using reduced fidelity models (e.g. transaction-accurate simulation) or hardware acceleration (e.g. FPGA emulation). As such, cycle-accurate RTL simulation performed by software is still the most commonly used tool and remains a persistent bottleneck for hardware design.

Fast simulation is even more important in an agile or open-source hardware design context. The designers not only want to iterate on designs quicker, but they may also have less resources with which to simulate them. It may be much more economical for them to reuse conventional compute resources they already have instead of pursuing a hardware-accelerated simulation platform. Additionally, for simulations of modest duration or shorter, the software simulator may return results quicker in practice than the hardware-accelerated simulator because of a much shorter compile time. At a minimum, software simulation is complementary to hardware-accelerated simulation, if not the only means of simulation.

Software simulation was formerly a vibrant research topic, but has received substantially less attention recently [7], [11]. A number of factors may have reduced the appeal of cycle-accurate simulation research, but we identify two prominent concerns. First, since simulation is a mature research topic, one might doubt how much additional speedup remains to be gained. Second, since simulators are typically frontend-bound on the host processor, the benefit of an optimization might be negated by exacerbated frontend bottlenecks. In this work, we counter these concerns to argue for a revival of software simulation research.

With the emergence of agile and open-source hardware design methodologies, the need for fast simulation is even greater. The rise of fast open-

 Published by the IEEE Computer Society

source simulators provides an interesting solution to the problem. Their open-source nature not only reduces adoption costs for practitioners, but it also provides a great platform for researchers to quickly evaluate their ideas. Software simulation research will be essential for agile hardware development, and we demonstrate its potential with the success of recent open-source simulators. This work makes the following contributions:

- A quantitative evaluation highlighting the performance advantages of two open-source simulators over a state-of-the-art commercial simulator which reveals there is substantial room to improve simulation performance.
- A thorough workload characterization using hardware performance counters which confirms the frontend bottleneck but also proves that aggressive optimizations can deliver net speedups despite increasing control-flow complexity. The practical utility of these novel optimizations demonstrates the benefit of researching simulation performance.
- A road map of potential research directions to improve simulation performance.

The combination of the availability of high-performance open-source simulators and the needs of agile design methodologies strongly motivate further research into accelerating software RTL simulation.

Simulation Background

Early RTL simulators computed the values of hardware designs by propagating signal updates as events, which also matched the semantics of the languages they modeled (e.g. Verilog). With these *event-driven* simulators, each time a signal is evaluated, it creates events to evaluate its children. Tracking which signals to activate and then dynamically scheduling them adds considerable overhead. *Full-cycle* simulators eliminate the scheduling overhead by performing the scheduling once at compile time and reusing it each cycle (static schedule) [7].

A full-cycle simulator is best thought of as a simulator *generator*. Rather than dynamically interpreting the design like an event-driven simulator, a full-cycle simulator generator consumes a design and produces a simulator specialized for that design. Typically, the simulator is produced by code generation and a conventional optimizing compiler. Creating a full-cycle simulator effectively "inlines" the entire design.

The generated simulators are typically bottlenecked by the frontend of the host processor. The simulator is essentially the entire design turned into mostly straight-line code. Each cycle, it simulates the entirety of the design, so instructions are not reused until the next simulated cycle. As the design gets larger, the resulting simulator binary also grows, further exacerbating the frontend bottleneck. With frequent instruction cache misses, conventional wisdom for simulator generators is to avoid unpredictable branches that could inhibit instruction flow. More generally, this fear results in a hesitation to consider aggressive optimizations. Full-cycle simulators are typically the fastest because the overhead reduction from static scheduling outweighs a loss in instruction throughput from the frontend bottleneck.

Emerging Open-Source Simulators

In this work, we analyze two open-source simulators to motivate additional simulator research.

Verilator

Verilator[1] is the fastest open-source Verilog simulator, and it is widely used not only due to its free nature, but also its absolute speed. It is a full-cycle simulator generator, and it emits C++ to be compiled by the host platform's compiler. Internally, it contains optimizations and heuristics that have been developed with years of feedback from real designs.

ESSENT

ESSENT is a new open-source simulator generator that employs a number of aggressive optimizations while still maintaining cycle accuracy [4]. At its core, ESSENT is a full-cycle simulator that uses conditional execution to skip over unnecessary simulation work. In this subsection, we briefly describe some of ESSENT's novel optimizations.

ESSENT's primary optimization exploits the fact that most signals in a hardware design rarely change (typical activity factor is 3%). The generated simulator recognizes which portions of

[1]https://www.veripool.org/wiki/verilator

the design are unchanged, and reuses those outputs. Detecting signal changes and thus determining which signals can be reused can introduce tremendous overheads if done at the fine granularity of individual signals. To amortize those overheads, ESSENT coarsens the design with its novel acyclic graph partitioner. The optimization is successful enough for ESSENT to enjoy a net speedup. The optimization results in an activity-proportional simulation in which a partition of the design is only evaluated if at least one of its inputs changes.

In addition to skipping unchanged signals, ESSENT also reduces simulation effort by not evaluating signals who will not have a persistent impact on the simulation. For example, for multiplexers, ESSENT evaluates the way select signal first, and then evaluates only the selected way. This skips the effort associated with the unselected way, and ESSENT analyzes the design to maximize the portion of the design that is safe to conditionally evaluate. Although this optimization adds more conditional branch instructions to the code, it can be quite beneficial in practice. Dovetailing on the mulitplexer optimization, ESSENT skips work that is bound for registers with writes disabled.

Like full-cycle simulators, the simulators generated by ESSENT strain the host processor with their large instruction working sets. Fortunately, ESSENT's extensive use of conditional execution provides an opportunity to shrink the effective instruction working set through better code layout. ESSENT emits branch hints the compiler uses to separate cold (infrequently used) code from hot code. ESSENT automatically instructs the compiler that the following activities are unlikely: multiplexer ways associated with reset, print statements, and triggered assertion handling.

ESSENT accepts hardware designs in FIRRTL, an intermediate representation language for hardware [9]. Compared to classic netlist formats, FIRRTL retains substantially more semantic information about the design which can better guide optimizing transformations. ESSENT generates C++ code that can be handed off to an optimizing compiler to produce a fast simulator. ESSENT effectively serves as yet another backend for FIRRTL, and it could thus use any language that produces FIRRTL, such as Chisel, PyRTL [6], and Spatial [8]. Additionally, using Yosys, one can translate most synthesizable Verilog to FIRRTL [12].

ESSENT itself is implemented in about three thousand lines of Scala, and it makes extensive use of FIRRTL's supporting library. Verilator is implemented with about an order of magnitude more code (C++) than ESSENT and FIRRTL combined. New open-source infrastructure like FIRRTL that leverage high-level languages like Scala make it easy for researchers to pioneer new concepts. The high-level language not only provides productivity benefits, but it also eases achieving correctness.

Evaluation

Methodology

In our evaluation, we compare **ESSENT** and **Verilator** to **CommVer**, a state-of-the-art industrial Verilog simulator. CommVer has been anonymized due to a license agreement, and we appropriately finesse its options to maximize performance, including using 2-state simulation. To isolate the benefits of ESSENT's optimizations, we also include **Baseline** which disables most of ESSENT's optimizations, and it is effectively a vanilla full-cycle simulator.

We use open-source processor designs to evaluate the simulators. Rocket Chip is a RISC-V SoC generator written in Chisel [3] that is used in research and industry [1]. To create more designs, we use versions from both 2016 (**rocket16**) and 2018 (**rocket18**). We also use BOOM, an out-of-order processor generator (**boom**) [5]. Qualitatively, in terms of size the three designs can be thought of as small, medium, and large. To animate the processors, we use three software workloads: **dhrystone** (synthetic microbenchmark), **matmul** (matrix multiplication), and **pchase** (synthetic pointer-chasing microbenchmark). We use an Intel 8-core 3.6 GHz i7-7820X (Skylake) which has 11 MB of L3 cache and 64 GB of DRAM to perform our experiments.

Performance

We first compare the simulators on overall performance, and we report speedups relative to CommVer since all of the open-source simulators outperform it (Figure 1). The (unoptimized) Baseline is comparable in performance to Verilator,

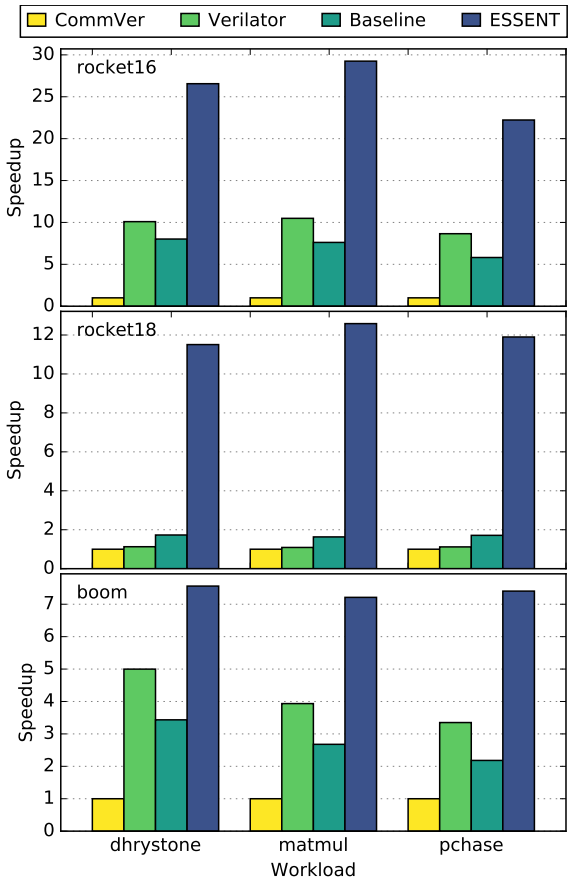

Figure 1. Speedups relative to CommVer.

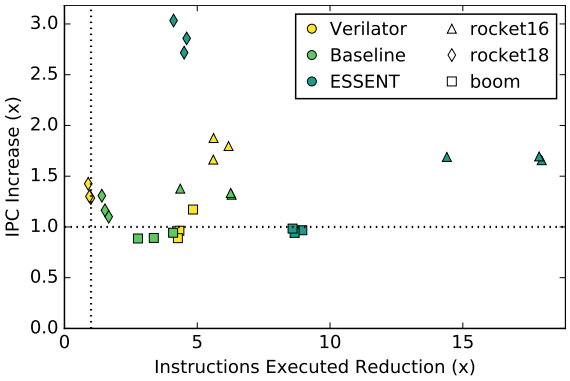

Figure 2. Increase in instructions per cycle (IPC) and reduction in instructions executed relative to CommVer (higher is better for both axes).

which is reasonable since they are both full-cycle simulators. Verilator significantly outperforms CommVer, and raw performance is one of Verilator's most compelling features. ESSENT significantly outperforms the other simulators, besting Verilator by 1.5–11.5× and CommVer by 7.2–29.3×. Despite being the newest and least mature simulator, ESSENT's performance advantage demonstrates the promise of novel optimizations.

Workload Analysis

We use hardware performance counters to analyze the interaction between the simulators and the host processor. In our analysis, we find the workloads' common limiting factors and we tease apart the benefit of ESSENT's optimizations.

As a starting point, we first consider the number of instructions executed by the host processor. We find the open-source simulators derive their performance advantage over CommVer primarily by executing far fewer instructions (Figure 2). The large reduction in instructions executed highlights the importance of the simulation method used. ESSENT's novel ability to detect and exploit inactivity within parts of the design allows it to avoid a substantial amount of simulation effort.

The open-source simulators also derive a speedup in part from executing instructions at a higher rate (Figure 2). For example, ESSENT's speedups on rocket18 are substantially greater due to a significant boost in instructions per cycle (IPC). To discern the cause of the IPC improvement, we first seek to understand the workload's bottlenecks.

Overall, the simulators we measure are strongly frontend-bound. They all use some form of full-cycle simulation, which causes the instruction working set to grow with the design size. Within a single simulated cycle, the large number of instructions fetched outmatches the host platform's 32 KB L1 instruction cache and often the combined 1 MB L2 cache (Figure 3). The reduction in instructions executed by the more efficient simulators directly reduces the number instruction cache misses because there is such disparate instruction reuse. The reduction in instruction cache misses is substantial, as it effectively makes the largest design (boom) for the efficient simulators (ESSENT & Verilator) comparable in size to the smallest design (rocket16) for CommVer.

In practice, we observe two qualitative performance regimes of normal performance (IPC $>$ 1.2) and substantially impaired performance (IPC $<$ 0.75). If the processor's frontend

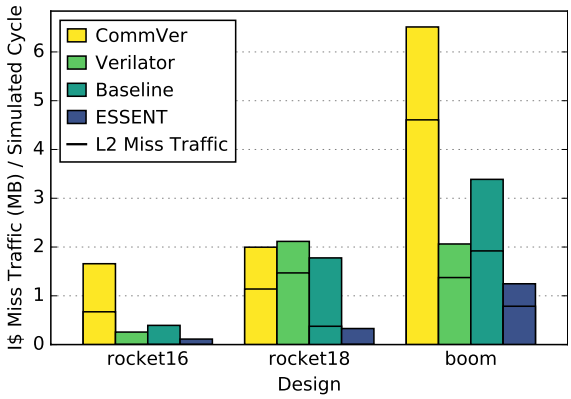

Figure 3. Total amount of data transferred per simulated cycle for instruction cache misses when executing dhrystone. The peak of each bar is for the L1 instruction cache, while the black line on each bar denotes the traffic that goes beyond the L2 cache.

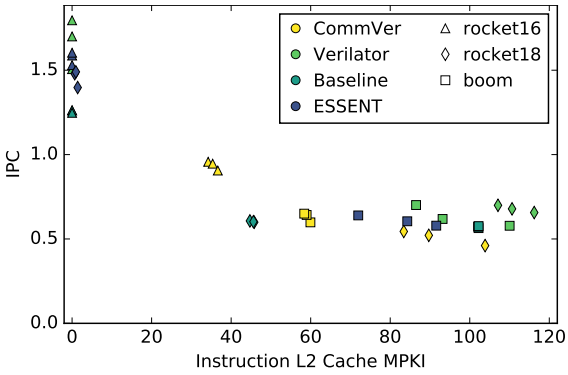

Figure 4. For instructions, frequent L2 misses per kilo instruction (MPKI) strongly limit IPC.

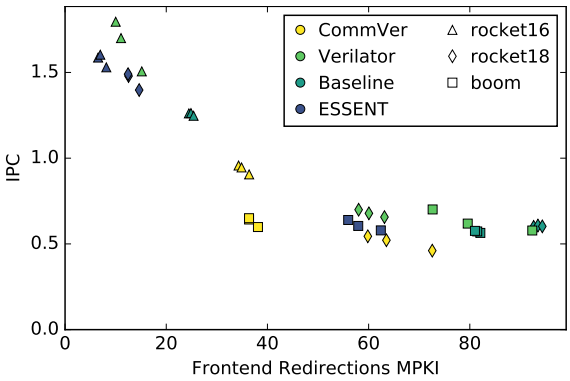

Figure 5. Frequent frontend redirections typically caused by BTB misses can severely hinder performance.

is able to handle the workload, normal performance is achieved, but if the workload grows too large and overwhelms the frontend, we see impaired performance. When the instruction footprint grows too large, it not only overflows the L1 instruction cache and the L2 cache, but it also overwhelms the frontend's ability to predict branches.

A given simulator's instruction mix only has a mild sensitivity to the processor design it is simulating, as the mix is primarily determined by the simulation approach. The software workload for the simulated processor has a near negligible impact on the dynamic instruction mix of the simulator. Despite the large fraction of memory operations, we find the L1 data cache is well suited to handle them (L1 data cache miss rate typically $<$10%). As data cache miss rates increase for larger designs, the frontend becomes more encumbered and continues to be a larger limiting factor.

We find that for our Skylake core, performance is good as long as instructions come primarily from the L1 instruction cache or even the combined L2 cache. Once there is a significant number of instruction misses that go beyond the L2 cache, the IPC drops substantially (Figure 4). All of the designs studied are suitably large to cause substantial L1 instruction cache misses, so whether the instruction working set can remain in the L2 is commonly the determining factor for performance (Figure 3). The smallest design (rocket16) typically fits in the L2, while the largest design (boom) does not. Depending on the simulation strategy for rocket18, the simulator will or will not fit within the L2.

Since instruction fetch is the main bottleneck, one might be concerned that the frontend is bandwidth bound when fetching cache lines for instructions. Due to the Skylake microarchitecture's increased cache bandwidths, we observe the bandwidth utilization is below 20% and typically around 10%. Thus, we conclude the frontend performance pathologies are more likely caused by latency penalties more so than bandwidth limitations for fetching instruction cache misses.

The large instruction footprint also stresses the core's branch prediction capabilities, as many unique branch instructions execute before a single branch repeats. Because of the long interval before branches repeat, branches are often evicted

from the branch target buffer (BTB) before they can be reused. Thus, the number of times the frontend is redirected due to a branch misprediction or a BTB miss is the most instructive metric. With a small design such as rocket16, the frontend is able to remember the branch targets and the IPC is high, however, larger designs overwhelm the BTB and other predictors, causing performance to drop substantially (Figure 5).

For rocket18, ESSENT's use of branch hints succeeds in shrinking the instruction footprint beyond a critical threshold to allow the program to remain within the L2 cache and the BTB. The optimization reduces the number of frontend redirections by 10× and the number of L2 cache misses by 7× on average. With the branch hints, the compiler is able to optimize the code layout to separate cold code from hot code. Conversely, the success of this optimization can be seen as an indication of the significant space cold code takes in the instruction working set [2]. The other two designs see only a modest benefit from the branch hints optimization, as rocket16 is already well handled by the frontend and boom continues to be too large for the frontend.

Interestingly, all of the simulators we evaluate execute a similar fraction of branch instructions. The existing simulators (CommVer & Verilator) execute nearly the same fraction of branches (13–16% & 14–16%). With help from ESSENT's multiword arithmetic code which is designed to be statically unrolled, Baseline executes a slightly smaller fraction of branch instructions (11–15%). All of ESSENT's aggressive optimizations only slightly increase the fraction of branch instructions to 11–19%. Despite the variance in techniques, the nearly constant fraction of branch instructions executed suggests aggressive optimizations in practice may not greatly aggravate the control-flow challenges for the host processor. In the worst case, performance may not degrade much further from frontend bottlenecks, as indicated by the flatlining to the right of Figure 4 & Figure 5. Finally, ESSENT's ability to deliver a net speedup demonstrates that novel aggressive optimizations for RTL simulation are both possible and worthwhile.

Road Map for Future Research

There are many promising ways to improve simulation, and in this section we highlight some potential avenues for future research, especially in directions enabled by advancements from other fields.

Considering the most commonly used RTL simulators are single-threaded, parallelization is a clear opportunity to improve performance. Although there is substantial prior work on parallelizing simulation, we believe a number of recent developments make it more practical now than before. First and foremost, the ubiquity of multicore systems eases development and deployment. Second, increasing design sizes ease parallelization (i.e. weak scaling) since they amortize synchronization overheads. Additionally, the increased flexibility of SIMD extensions makes them easier to utilize, and they may be a good match for replicated portions of the design. Finally, a multithreaded simulator may benefit from simultaneous multithreading (SMT), since the frontend's latency bottleneck could be overlapped across threads.

Taking advantage of advancements in programming languages and compilers could also help improve simulation. High-level hardware intermediate representations (IR) such as FIRRTL allow for more intelligent transformations within simulator generators. The availability of high-quality open-source compiler infrastructure such as LLVM could be used in a few ways. First, a simulator generator could produce the compiler's IR rather than C++, removing the need for C++ generation and parsing as an intermediate step in simulator creation. Secondly, the simulator generator could take advantage of sophisticated compiler optimizations such as polyhedral auto-vectorization to target SIMD extensions on the host processor. Finally, recent work has shown the promise of making the simulator generation process just-in-time (JIT) instead of ahead-of-time, which allows for simulation to start immediately but accelerate over time [10].

Research on memory layouts could also benefit simulation. At a minimum, improved code layout could ameliorate the frontend bottleneck with improved instruction locality. These insights could be deployed via profile-guided optimization

or even simple heuristics like ESSENT's fruitful use of branch hints. Improved data layout could help by reducing instruction count due to improved spatial locality (smaller load offsets) or collocating data to exploit bit-level parallelism with sub-word SIMD.

Improvements to open-source simulation technology will not only benefit the agile hardware design community, but it could also help accelerate development in related areas. The existence of a high-quality simulation backend will lower the barrier to creating new hardware description languages. Additionally, the simulation infrastructure's performance advantages could enable more ambitious verification methods.

Discussion

Our experiences from this work add further evidence of the feasibility of open-source RTL simulators. Not only can they significantly outperform industrial tools, their open nature enables others to contribute. This is especially helpful for researchers, who can quickly modify an existing working system to prototype an idea instead of starting completely from scratch.

This work focuses on software RTL simulation, which is complementary to other simulation techniques. The low equipment and startup time costs make software simulation especially attractive for agile design, but the other techniques could still prove useful. Transaction level modeling is best for early rapid design space exploration as well as mocking units for testing. Hardware-accelerated simulation can deliver the highest performance once the large startup overhead has been overcome.

Our analysis of ESSENT's optimizations with performance counters reveals that conditional execution can deliver net speedups. The optimizations substantially reduce the number of instructions executed, and the IPC is typically unchanged. Thus, future researchers should be bold when considering new simulation optimizations, even if they may initially appear to increase control complexity. At a minimum, software simulation will most likely continue to be bottlenecked by the host processor's frontend, so optimizations targeting the processor's instruction throughput will likely prove fruitful. Conversely, full-cycle simulators could serve as a great open-source workload for computer architects to optimize when considering the challenges of large application binaries [2].

ESSENT's use of Scala and FIRRTL's library provides an additional demonstration of the utility of high-level languages and open-source. More generally, open-source and high-level languages are enabling tools to develop the next generation of fast simulators necessitated by agile design methodologies. The emergence of agile hardware design methodologies and the impressive simulation speedups obtained by recent work show there is both a need and a means to accelerate simulation.

Acknowledgments

This material is based upon work supported by, or in part by, the Army Research Laboratory and the Army Research Office under contract/grant W911NF1910466.

REFERENCES

1. Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.
2. Grant Ayers, Parthasarathy Ranganathan, Nayana Prasad Nagendra, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, and Tipp Moseley. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers. In *International Symposium on Computer Architecture (ISCA)*, pages 462–473. ACM Press, 2019.
3. Jonathan Bachrach, Huy Vo, Brian Richards, et al. Chisel: constructing hardware in a scala embedded language. *DAC*, pages 1216–1225, 2012.
4. Scott Beamer and David Donofrio. Efficiently exploiting low activity factors to accelerate RTL simulation. *DAC*, 2020.
5. Christopher Celio, Krste Asanovic, and David Patterson. The berkeley out-of-order machine (BOOM): An industry- competitive, synthesizable, parameterized RISC-V processor. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167*, 2015.
6. John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. A pythonic approach for rapid hardware prototyping and instrumentation. In *International Conference*

on Field Programmable Logic and Applications (FPL), 2017.

7. Craig Hansen. Hardware logic simulation by compilation. *DAC*, 1988.
8. David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, et al. Spatial: A language and compiler for application accelerators. *Conference on Programming Language Design and Implementation (PLDI)*, 53(4):296–311, 2018.
9. Patrick S Li, Adam M Izraelevitz, and Jonathan Bachrach. Specification for the FIRRTL language. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-9*, 2016.
10. Eric Schkufza, Michael Wei, and Christopher J Rossbach. Just-in-time compilation for verilog: A new technique for improving the fpga programming experience. In *ASPLOS*, pages 271–286, 2019.
11. Laung-Terng Wang, Nathan E Hoover, Edwin H Porter, and John J Zasio. SSIM: A software levelized compiled-code simulator. *DAC*, 1987.
12. Clifford Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/, 2016.

Scott Beamer is an assistant professor at the University of California, Santa Cruz. His research interests include open-source hardware design, high-performance graph processing, and computer architecture. He has a PhD in Computer Science from the University of California, Berkeley. He is a member of the Computer Society, IEEE, ACM, and SIAM. Contact him at sbeamer@ucsc.edu.