FPGA-Accelerated Evaluation and Verification of RTL Designs

Donggyu Kim

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-57
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-57.html

May 17, 2019

FPGA-Accelerated Evaluation and Verification of RTL Designs

by

Donggyu Kim

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Krste Asanović, Chair
Adjunct Assistant Professor Jonathan Bachrach
Professor Rhonda Righter

Spring 2019

FPGA-Accelerated Evaluation and Verification of RTL Designs

by

Donggyu Kim

Abstract

FPGA-Accelerated Evaluation and Verification of RTL Designs

by

Donggyu Kim
Doctor of Philosophy in Computer Science

University of California, Berkeley
Professor Krste Asanović, Chair

This thesis presents fast and accurate RTL simulation methodologies for performance, power, and energy evaluation as well as verification and debugging using FPGAs in the hardware/software co-design flow.

Cycle-level microarchitectural software simulation is the bottleneck of the hardware/software co-design cycle due to its slow speed and the difficulty of simulator validation. While simulation sampling can ameliorate some of these challenges, we show that it is often insufficient for rigorous design evaluations. To circumvent the limitations of software-based simulation and sampling, this thesis presents MIDAS v1.0, which automatically generates the FPGA-accelerated RTL simulator as an instance of FAME1 from any RTL. These simulators are not only up to three orders-of-magnitude faster than existing microarchitectural software simulators, but also truly cycle-accurate, as the same RTL is used to build the silicon implementation.

The increasing complexity of modern hardware design makes verification challenging, and verification often dominates design costs. While formal verification and unit-level tests can improve the confidence in some blocks or some aspects of a design, dynamic verification using simulators or emulators with real-world applications is usually the only plausible strategy for system-level RTL verification. Therefore, this thesis presents DESSERT, an effective simulation-based RTL verification methodology using FPGAs. The target RTL design is automatically transformed and instrumented to allow deterministic simulation on the FPGA with initialization and state snapshot capture. Assert statements, which are present in RTL for error checking in software simulation, are automatically synthesized for quick error checking on the FPGA, while print statements in the RTL design are also automatically transformed to generate logs from the FPGA for more exhaustive error checking. To rapidly provide waveforms for debugging, two parallel simulations are run spaced apart in simulation time to support capture and replay of state snapshots immediately before an error.

Energy efficiency is the primary metric for all computing systems, requiring designers to evaluate energy efficiency quickly and accurately throughout the design process. Prior abstract energy models are only accurate for designs closely matching

the template for which the model was constructed and validated. Any energy model must be calibrated to a ground truth, usually a real physical system or a gate-level energy simulation. Validation of energy models is difficult because only a few design points will ever be fabricated as real systems and real systems typically lack adequate energy instrumentation, and gate-level simulation of proposed designs is extremely slow.

For fast and accurate power and energy evaluation of RTL, this thesis first presents Strober, a sample-based energy simulation methodology. Strober uses an FPGA to simultaneously simulate the performance of an RTL design and to collect samples containing exact RTL state snapshots. Each snapshot is then replayed in RTL/gate-level simulation, resulting in a workload-specific average power estimate with its confidence interval. For arbitrary RTL and workloads, Strober guarantees orders-of-magnitude speedup over commercial CAD tools and gives average energy estimates guaranteed to be within very small errors with high confidence.

Runtime power modeling is also necessary for dynamic power/thermal optimizations. This thesis finally presents Simmani, an activity-based runtime power modeling methodology which automatically identifies key signals for the runtime power dissipation of any RTL design. The toggle pattern matrix, in which each signal is represented as a high-dimensional point, is constructed from the VCD dumps of a small training set. By clustering signals showing similar toggle patterns, an optimal number of signals are automatically selected, and then the design-specific but workload-independent activity-based power model is trained with regression against power traces obtained from industry-standard CAD tools. Simmani also automatically instruments the target design with activity counters to collect activity statistics from FPGA-based simulation, enabling runtime power analysis of real-world workloads at speed.

In loving memory of my mother, Jeongja Seo (1963 - 2014).

Contents

List of Figures

List of Tables

Acknowledgments

I am very grateful I have been standing on the shoulders of giants at UC Berkeley.

First of all, I would like to thank my advisors, Krste Asanović and Jonathan Bachrach. I was very fortunate they gave me a chance to be part of a great research group at UC Berkeley. Even though I struggled a lot in my early days, they never gave up on me. With their continuous encouragement and patience, they have shown me how to build computer systems and how to do computer architecture research. I also like to thank Bora Nikolić and Rhonda Righter for gladly being on my quals and thesis committee.

It is a great pleasure to have many talented colleagues and friends surrounding me. Special thanks to Chris Celio, who developed BOOM for his Ph.D, which in turn enriched my research. Without his heroic effort, what I have done throughout my thesis would have been much less valuable. He also cheered for my research and inspired new research ideas like DESSERT. I also appreciate that BOOM is now supported and maintained by Abraham Gonzalez, Ben Korpan, and Jerry Zhao.

Big thanks to the MIDAS/FireSim team including David Biancolin, Sagar Karandikar, and Howard Mao. I really enjoyed working with them. It is amazing to see my early work on MIDAS now becomes FireSim, one of the most successful research project at UCB BAR. Nobody can imagine how much time and energy have been invested in this project, and I will remember that you guys have come to the lab most of weekends.

Thanks to everyone who makes Chisel and FIRRTL great, including Jonathan Bachrach, Adam Izraelevitz, Jack Koenig, Richard Lin, Albert Magyar, Jim Lawson, Chick Markley, Andrew Waterman, Stephen Twigg, Wenyu Tang, Angie Wang, Paul Rigge, and Schuyler Eldridge. Chisel and FIRRTL are the backbone of my thesis, without which my thesis would have been literally impossible. Beyond my creepy hacks on Chisel2 and its tester, I am glad to see Chisel and FIRRTL are widely used and contributed by the outside world.

I thank the Hwacha team including Yunsup Lee, Albert Ou, and Colin Schmidt. Hwacha played an important role in evaluation for Simmani, and it is amazing to see this sophisticated design working very well. I also thank Jerry Zhao for the SqueezeNet port for Hwacha, without which evaluation for Hwacha would have been much less insightful.

Thanks to Edward Wang for his contribution to HAMMER. Very interestingly, I first met him as a lab assistant when I was a CS 61C TA. What a small world! Without HAMMER, working with CAD tools for Simmani would have been much more painful.

Thanks to Kevin Laeufer and Jack Koenig for the RFUZZ project. It was very exciting we made it work on the FPGA in crunch time. Big thanks to the energy modeling team back in the day including Adam Izraelevitz, Brian Zimmer, Hokeun Kim, and Yunsup Lee.

Thanks to Andrew Waterman, an inventor of RISC-V, from which all greatness starts. He also gladly answered my silly questions and never hesitated to look at waveforms when I asked while working on DESSERT. Thanks to Martin Mass for his porting effort to Jikes JVM and the DaCapo benchmarks used in the case study of MIDAS. Also thanks to Scott Beamer, Henry Cook, Rimas Avizienis, Brian Zimmer, Eric Love, Palmer Dabbelt, Ben Keller, John Wright, Alon Amid, Nathan Pemberton, Jenny Huang, and Vighnesh Iyer for their contributions to various projects and tapeouts at UCB BAR.

I would like to thank my family. My father, Taegeun Kim, has always been supportive in Korea. My younger brother, Donggyun Kim, who by chance has almost the same first name as mine except the ending n[1], has been also a great roommate at Berkeley. I am happy he will soon receive his bachelor's degree from UC Berkeley. Finally, this thesis is devoted to my mother, Jeongja Seo, a true supporter for me during her entire life.

Funding Support. My research in this thesis has been supported by the following sponsors:

- **ASPIRE Lab:** DARPA PERFECT program, Award HR0011-12-2-0016.DARPA POEM program Award HR0011-11-C-0100. The Center for Future Architectures Research (C-FAR), a STARnet center funded by the Semiconductor Research Corporation. Additional support from ASPIRE industrial sponsor, Intel, and ASPIRE affiliates, Google, Huawei, Nokia, NVIDIA, Oracle, and Samsung.

- **ADEPT Lab:** 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 partially funded by ADEPT Lab industrial sponsor Intel, and ADEPT Lab affiliates Google, Siemens, and SK Hynix.

I was partly supported by the Kwanjeong Educational Foundation. The views and opinions of the author expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

[1] This is a big difference in Korean

Chapter 1

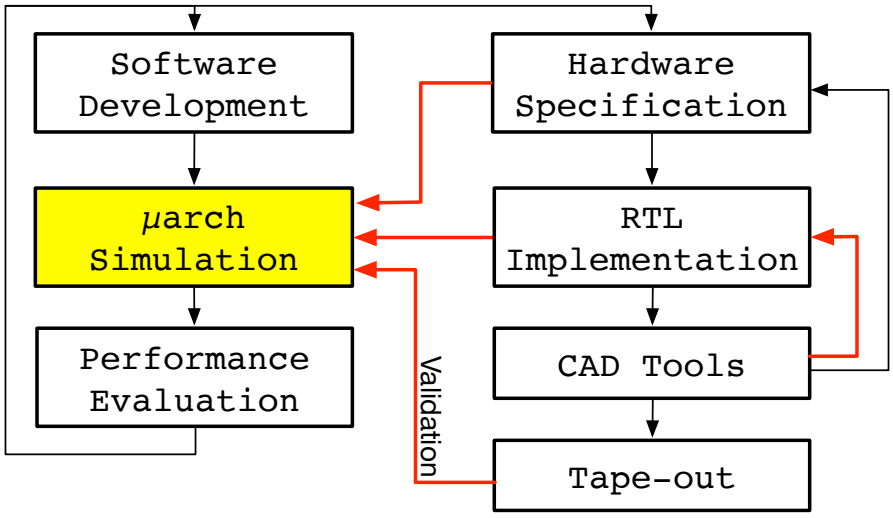
Introduction

This thesis proposes a new evaluation and verification methodology in the hardware/software co-design flow. Section 1.1 motivates why performance, power, and energy evaluation using RTL designs instead of microarchitectural software simulators is necessary for the recent hardware/software co-design trends. Section 1.2 outlines the remaining chapters of this thesis.

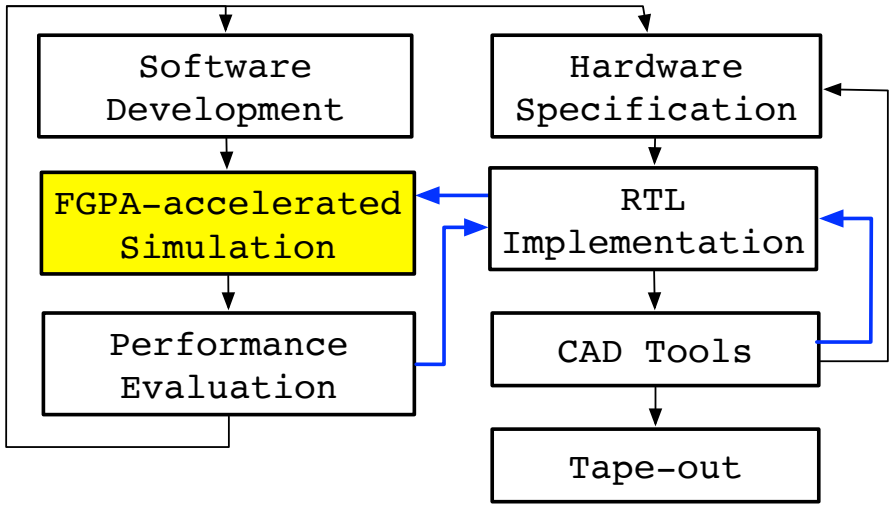
1.1 Why RTL-Based Computer Architecture Research?

Figure 1.1 shows two alternatives for performance evaluation in the computer system design process. Historically, cycle-level microarchitectural software simulation (e.g. [19, 109, 143]) has been widely-used by computer architects for performance evaluation (Figure 1.1a). Whereas RTL implementation has been tedious, labor-intensive and difficult to modify and verify, these microarchitectural software simulators are flexible and easy to use. For this reason, high-level software-based simulation remains an important tool to guide system designers in the early stages of system design, before RTL implementation has commenced. Ease-of-use concerns notwithstanding, evaluating RTL designs remains extremely slow when using commercial CAD tools, and thus, high-level simulation was necessary to evaluate the target systems with real-world applications.

However, *microarchitectural software simulation becomes the bottleneck of the recent hardware/software co-design cycles nowadays* for two main reasons. First of all, microarchitectural software simulators should be carefully validated against RTL designs and real systems. This is only feasible when new designs do not deviate tremendously from existing hardware or similar designs from previous design cycles that have already been validated. As shown in Figure 1.1a, there are frequent feedback loops from the CAD tools to the RTL designs in the hardware design cycle in



(a) Performance evaluation using microarchitectural simulation



(b) Performance evaluation using RTL designs

Figure 1.1: Comparison of performance evaluation methodologies in the hardware/-software co-design flow

order to improve the quality of the silicon implementation. Whenever any changes are made to the RTL designs, microarchitectural software simulators should also be carefully tuned. Moreover, microarchitectural software simulators should be exhaustively validated against the silicon implementations running real-world software. Otherwise, various kinds of modeling errors can be introduced by the abstraction of the target systems [53]. Recent hardware design trends, moving toward heterogeneous SoCs with a plethora of custom hardware accelerators, have made simulator validation more difficult as it has become harder to find an existing system against which these software-based simulators are validated.

To make matters worse, microarchitectural software simulation is too slow to run the full execution of today's realistic workloads with very complicated hardware designs. In general, any non-trivial modern application executes billions to trillions of dynamic instructions, making it practically impossible to simulate them without sampling even with the fastest microarchitectural software simulator. For example, it takes roughly a month to simulate one trillion instructions with a 400 KIPS simulator [109]. Therefore, the slow speed of microarchitectural software simulation prevents an agile hardware development approach, which is required for rapid hardware/software co-optimizations.

This thesis proposes *FPGA-accelerated RTL simulation* to resolve the difficulty of performance, power, and energy evaluation for the hardware/software co-design flow as shown in Figure 1.1b. First of all, RTL implementation is becoming more productive for computer architects thanks to various hardware construction languages (e.g. [10], [105], [118], [94], [135]) that greatly improve the expressivity over existing hardware description languages like Verilog and VHDL. In addition, there are an increasing number of open-source RTL designs (e.g. [8, 34]) with which to bootstrap a new project, dramatically reducing RTL development time.

Next, this methodology is truly cycle-accurate as the FPGA-accelerated simulator is generated using automatic transformations on any RTL design that will be the same RTL consumed by VLSI CAD tools to produce the silicon implementations. These transformations preserve the RTL behavior of the design, and thus do not introduce modeling errors. Therefore, system designers do not need to re-validate their simulator after changes on their RTL design, greatly improving the productivity of hardware/software co-optimization, where the RTL design may be frequently modified.

Finally, using FPGAs for performance evaluation increases the simulation speed by multiple orders of magnitude over existing cycle-level microarchitectural software simulation, which is well demonstrated by the previous work [132]. Since FPGA-accelerated simulators can execute tens of MIPS, it becomes possible to evaluate complete benchmark suites, like SPEC2006int, on a cycle-accurate model of the design.

1.2 Thesis Outline

This thesis describes fast and accurate RTL simulation methodologies using FPGAs for performance, power, energy evaluation as well as verification and debugging.

Chapter 2 describes the background of this thesis. Section 2.1 reviews various existing performance modeling methodologies including analytic modeling, software-based simulation, simulation sampling, and FPGA-accelerated simulation. Section 2.2 introduces Chisel and FIRRTL, which are crucial tools for this thesis. This section also describes the *compiler-in-a-pass* technique used for various custom transforms implemented in this thesis. Section 2.3 overviews example RTL designs including the Rocket in-order processor, the BOOM out-of-order processor, and the Hwacha vector accelerator evaluated in this thesis. Section 2.4 discusses various challenges arising with RTL implementations.

Chapter 3 presents MIDAS v1.0, a framework that automatically generates FPGA-accelerated performance simulators from any RTL designs. Section 3.1 discusses the motivation of MIDAS v1.0. Section 3.2 describes MIDAS v1.0 and its implementation with FIRRTL compiler passes. Section 3.3 shows the evaluation results of Rocket and BOOM running the SPECint2006 benchmark suite and the DaCapo benchmarks using MIDAS. Section 3.4 concludes this chapter.

Chapter 4 presents DESSERT, an effective RTL debugging methodology using FPGAs. Section 4.1 shows why RTL debugging can be very challenging and painstaking. Section 4.2 covers existing simulation-based RTL verification and debugging methodologies. Section 4.3 presents the DESSERT framework and its implementation for RTL debugging using FPGAs. Section 4.4 shows the debugging results for BOOM-v2 using DESSERT. Section 4.5 summarizes this chapter.

Chapter 5 presents Strober, a sample-based energy modeling methodology for average power and energy estimation. Section 5.1 motivates why RTL-based power and energy evaluation is necessary for computer architecture research. Section 5.2 describes existing design-time power modeling methodologies. Section 5.3 overviews our sample-based energy evaluation methodology. Section 5.4 presents the Strober framework as an implementation of sample-based energy evaluation. Section 5.5 shows the evaluation results of Rocket and BOOM with Strober. Section 5.6 concludes this chapter.

Chapter 6 presents Simmani, an activity-based runtime power modeling that automatically identifies key signals for the runtime power dissipation of any RTL design. Section 6.1 motivates why runtime power modeling is necessary for computer systems. Section 6.2 covers existing runtime power modeling methodologies. Section 6.3 describes how Simmani selects key signals for runtime power dissipation with signal clustering and then trains module-level runtime power models with regression against power traces from CAD tools. Section 6.4 explains how Simmani automatically instruments activity counters for the selected signals, enabling runtime power analysis with FPGA-based simulation. Section 6.5 and shows the evaluation results

of Rocket and BOOM with Simmani. Section 6.6 shows the evaluation results of Hwacha with Simmani. Section 6.7 summarizes this chapter.

Chapter 7 concludes this thesis by presenting its contributions as well as potential future work based on the progress of this thesis.

Chapter 2

Background

This chapter presents the background of this thesis. Section 2.1 overviews existing evaluation methodologies for computer architecture research. Section 2.2 describes Chisel and FIRRTL that play an important role throughout this thesis. Section 2.3 introduces example RTL designs evaluated in this thesis. Section 2.4 discusses various challenges arising from RTL implementation, motivating the methodologies developed in this thesis.

2.1 History of Evaluation Methodologies for Computer Architecture Research

2.1.1 Analytic Modeling

There is a long history on analytic performance modeling for microprocessors. First of all, there are a lot of efforts to determine the optimal pipeline length using analytic models. Emma and Davidson [45] present an analytic model to predict the optimal pipeline length for in-order processors using statistics on branches and data dependences collected from instruction traces. Hartstein and Puzak [54] predict the optimal pipeline length of superscalar processors with an analytic model in terms of pipeline stalls. Sprangle and Carmean [124] find that the optimal pipeline depth is primarily determined by branch misprediction penalties, but is independent of cache sizes.

There are also a huge amount of studies to explain system performance with respect to instruction-level parallelism (ILP). Jouppi [69] studies the non-uniform distribution of ILP across various benchmarks. Debey et al. [43] model the throughput of a processor using the ILP distribution and the cost of branches. Noonburg and Shen [106] model the ILP distribution in the matrix form, considering available parallelism in each pipeline stage. Michaud et al. [99] find the $\sqrt{N}$ relationship between the

ILP and the instruction window size, the number of instructions waiting for execution in the pipeline, and compute the threshold fetch rate using this relationship.

Karkhanis and Smith [71] present performance modeling for out-of-order processors in terms of the rate of instructions issued per cycle (IPC) over time. They observe the IPC rate becomes zero with pipeline miss events such as i) branch misprediction, ii) instruction cache and TLB misses, and iii) long-latency backend events such as L2 cache and TLB misses. By extending this performance modeling, Eyerman et al. [47] suggest a performance counter architecture, which is used to construct CPI stacks that identify performance bottlenecks across various workloads. Taha and Wills [129] present a similar performance model in which the IPC is modeled in terms of macro blocks, a group of instructions divided by branch mispredicts.

The Roofline model [145] is a simple 2-D graph model for high performance computing in terms of operational intensity (x-axis), floating-point performance (y-axis), peak floating-point performance (a strait line), and peak memory bandwidth (a line of unit slope). Operational intensity is application-dependent while floating-point performance and memory bandwidth are only machine-dependent. By measuring the operational intensity of a given application, we can readily visualize whether it is computation-bounded or memory-bandwidth-bounded as well as whether or not its theoretical maximum performance is achieved. This performance model is also used to evaluate Google's Tensor Processing Unit (TPU) [68].

There are also various studies on analytic performance modeling for memory systems. Hill and Smith [55] present an efficient algorithm for cache performance estimates with various cache parameters from a single run of simulation. There is a large amount of work on cache performance modeling based on reuse distance [4, 16, 117, 13]. Sorin et al. [123] describe analytic modeling for shared memory systems. Willick and Eager [146] present analytic modeling for interconnection networks.

Accurate analytic performance modeling has been always our dream because analytic models are simple, intuitive and quick. However, analytic modeling is inaccurate in many cases by its nature. In hardware designs, there are lots of overlapping operations interacting each other. We may improve modeling accuracy by introducing more variables to explain more about these operations, which may end up being as complex as detailed simulation. Moreover, analytic models have been effective only for well-known hardware designs as they should be rigorously validated against real implementations, which are unlikely available for novel hardware designs such as domain-specific custom accelerators.

2.1.2 Software-Based Simulators

Microarchitectural software simulators [19, 143, 109] are the most popular tools for computer architecture research. These simulators are arguably cycle-accurate as they keep track of cycle-by-cycle microarchitectural state. However, these simulators are truly cycle-accurate only if they are rigorously validated against RTL or silicon

```
541  590        if(cacheLines_->get_port(queueEntry->request)) {
     591  +             /***** by vteori *****/
     592  +             // for perfect caches
     593  +             int robid = queueEntry->request->get_robid();
     594  +             bool icache_walk = queueEntry->request->is_instruction();
     595  +             bool itlb_walk = memoryHierarchy_->is_itlb_miss();
     596  +             bool dtlb_walk = memoryHierarchy_->is_dtlb_miss(robid);
     597  +             bool perfect_l2_icache = config.perfect_l2_icache && type_ == L2_CACHE && icache_walk && !itlb_walk;
     598  +             bool perfect_l2_dcache = config.perfect_l2_dcache && type_ == L2_CACHE && !icache_walk && !dtlb_walk;
     599  +
     600  +             // for debug by vteori
     601  +             /*if (!icache_walk) ptl_logfile
     602  +                     << (type_ == L2_CACHE ? "L2 $ " : "L1 $ ")
     603  +                     << "access => rob : " << robid << " addr : "
     604  +                     << (void *) queueEntry->request->get_physical_address() << endl;*/
     605  +
542  606             CacheLine *line = cacheLines_->probe(queueEntry->request);
543  607             bool hit = (line == NULL) ? false : line->state;
     608  +             hit |= perfect_l2_icache;
     609  +             hit |= perfect_l2_dcache;
```

Figure 2.1: Single-cycle perfect caches in MARSSx86

implementations because *microarchitectural state is a subset of RTL state manually defined by hardware designers*. Gutierrez et al. [53] describe possible source of errors in these simulators due to manual abstraction and modeling.

Indeed, we can easily introduce software hacks for unrealistic hardware configurations in software-based simulators. Figure 2.1 shows 10 lines of C++ code for single-cycle perfect caches in MARSSx86[1], which may be useful for hypothetical studies. However, this also shows that one can be easily unaware of constraints in the real implementation, leading to unrealistically-optimistic conclusions (for publications), which can happen when new design ideas are evaluated only with software-based simulators.

Another issue is microarchitectural software simulators are too slow to execute real-world applications to completion. These cycle-level simulators run only at tens or hundreds of kilo instruction per second (KIPS), orders-of-magnitude slower than real machines. To overcome the slowness of cycle-level microarchitectural simulators, a variety of software simulators that trade off accuracy for speed are proposed [102, 28, 113]. Another popular technique is simulation sampling as explained in Section 2.1.3. However, these workarounds are sometimes unacceptable as they can further increase modeling errors.

[1]This was done when I was an undergraduate student.

2.1.3 Simulation Sampling

Simulation sampling is a popular technique to overcome the sluggishness of cycle-level microarchitectural software simulators. There are two major approaches: phase-based sampling [121] and statistical sampling [148].

Phase-based sampling [121] provides simulation points through phase analysis on dynamic basic-block traces. Here, the underlying assumption is the dynamic execution of software consists of short periods of phases and each phase will exhibit similar microarchitectural behavior, and thus instructions per cycle (IPC), whenever repeated. The simulation points produced by this methodology are those centered about the basic blocks that are most frequently visited. This approach provides no guaranteed error bounds.

However, this is not necessarily true because the periods of phases can be lengthy and the performance characteristics of each phase depend on the dynamic state of the systems. Figure 2.2 shows the IPC trace over the entire execution of `401.bzip2` in the SPEC2006int benchmark suite with its reference input, running on BOOM-2w (Table 3.1). A sample was collected every 100 million cycles: each point represents the average IPC over that 100 million cycle interval. Samples were collected from the FPGA as the simulation executed. Even though there are some distinct execution phases, each phase has a non-trivial length period and shows different performance characteristics when repeated. Thus, *running a couple of million instructions in dozens of representative points from phase analysis is not enough to characterize the machine's performance for real-world applications.*

On the other hand, statistical simulation sampling [148] provides performance estimates with statistically-bounded errors. The main idea is the intervals between detailed simulation points are fast-forwarded using fast functional simulation. However, for this approach to be effective, the microarchitectural state of the machine must be reconstituted in a **warming** phase. This requires considerable execution time in the detailed simulator even before evaluating the sample point.

Various methods (e.g. [143]) have been suggested to address this state warming problem, most of which propose keeping track of some microarchitectural state in functional simulation. Of course, this slows down the fast functional simulation, making it difficult to recollect simulation samples. Thus, architects must choose between either the increased design iteration time to perform sample recollection, or potential simulation inaccuracy that may result from using stale samples. Finally, it is not clear what subset of state should be warmed for non-traditional hardware designs such as custom hardware accelerators.

2.1.4 FPGA-Accelerated Simulators

Motivated by the dawn of the multicore era, the multi-university RAMP project [142] was initiated to improve the simulation speed of multi-core processors using FPGAs.

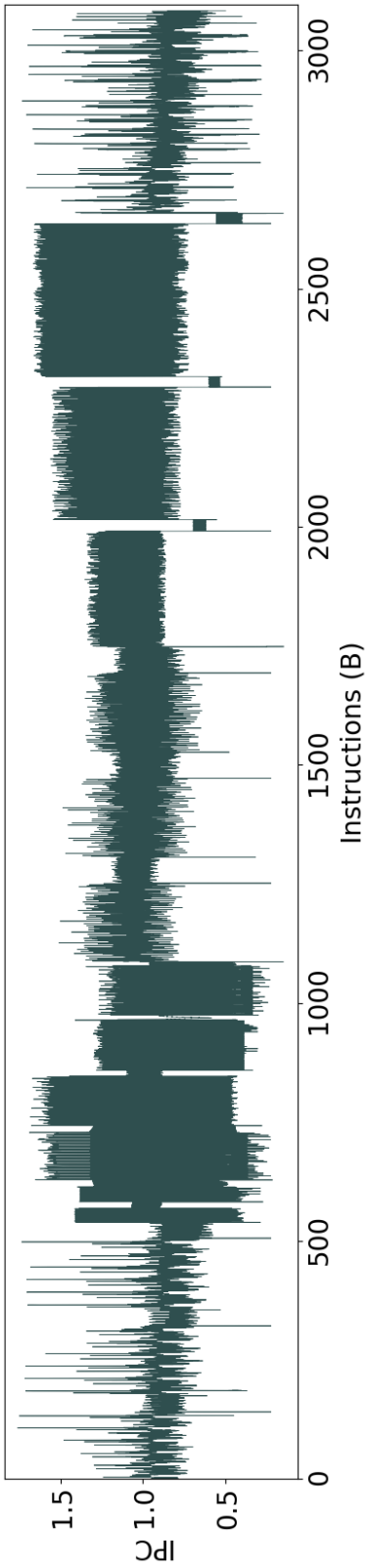


Figure 2.2: IPC trace for `401.bzip2` on BOOM-2w

Under the umbrella of the RAMP project, there have been significant efforts to develop techniques for FPGA-based performance simulators [38, 36, 132, 131, 111, 130]. ProtoFlex [38] implements a multi-core functional simulator on the FPGA. FAST [36] is a hybrid approach simulating a function model in software and a timing model on the FPGA. Tan et al. [132] describe different FAME levels. FAME0 simulators directly emulate the RTL design on the FPGA. FAME1 simulators are decoupled from the host memory simulation to match the target DRAM timing models. FAME7 simulators implement abstract models and simulation multi-threading on top of FAME1. RAMP Gold [131] and HASim [111] are examples of FAME7 simulators. To model a datacenter-scale system, DIABLO [130] leveraged RAMP Gold's multithreading to simulate 3072 servers on 24 FPGAs with abstract network interface card (NIC) and switch models.

The simulators above are orders of magnitude faster than software simulators, but they require simulator engineers to manually describe abstract models in RTL, which may be more difficult than writing RTL for the target design. In contrast, this thesis presents a methodology to automatically generates FAME1 simulators directly from target RTL designs to accurately model the target design's timing behavior.

2.2 Chisel & FIRRTL: Improving Productivity with Hardware Generators and Compiler Transforms

In this section, we introduce Chisel and FIRRTL, which have been the heart of research tools at Berkeley Architecture Research and widely adopted by both academia and industry nowadays. Figure 2.3 highlights the hardware development stack using Chisel and FIRRTL.

Chisel [10] is a hardware construction language embedded in Scala [107] that helps hardware designers generate RTL with various parameters by providing access to advanced parameterization systems [2]. Note that Chisel is not a high-level synthesis tool; like Perl or Python scripts that modify or generate Verilog, a designer uses Chisel's host language Scala to create and connect structural RTL components.

A core idea of design methodologies using Chisel is writing *reusable libraries* with *hardware generators* instead of single instances of hardware designs, which can be accomplished by high-level language features and advanced parameterization. As a result, RTL implementation is much more productive since designers can reuse well-verified existing libraries for their own designs without pain. Otherwise, they need to rewrite similar instances of existing hardware blocks or end up with error-prone scripts to generate them.

The hardware design productivity can be further enhanced by custom compiler transforms. For example, memory blocks are technology-dependent and need to

[2] Available at https://github.com/freechipsproject/chisel3.git

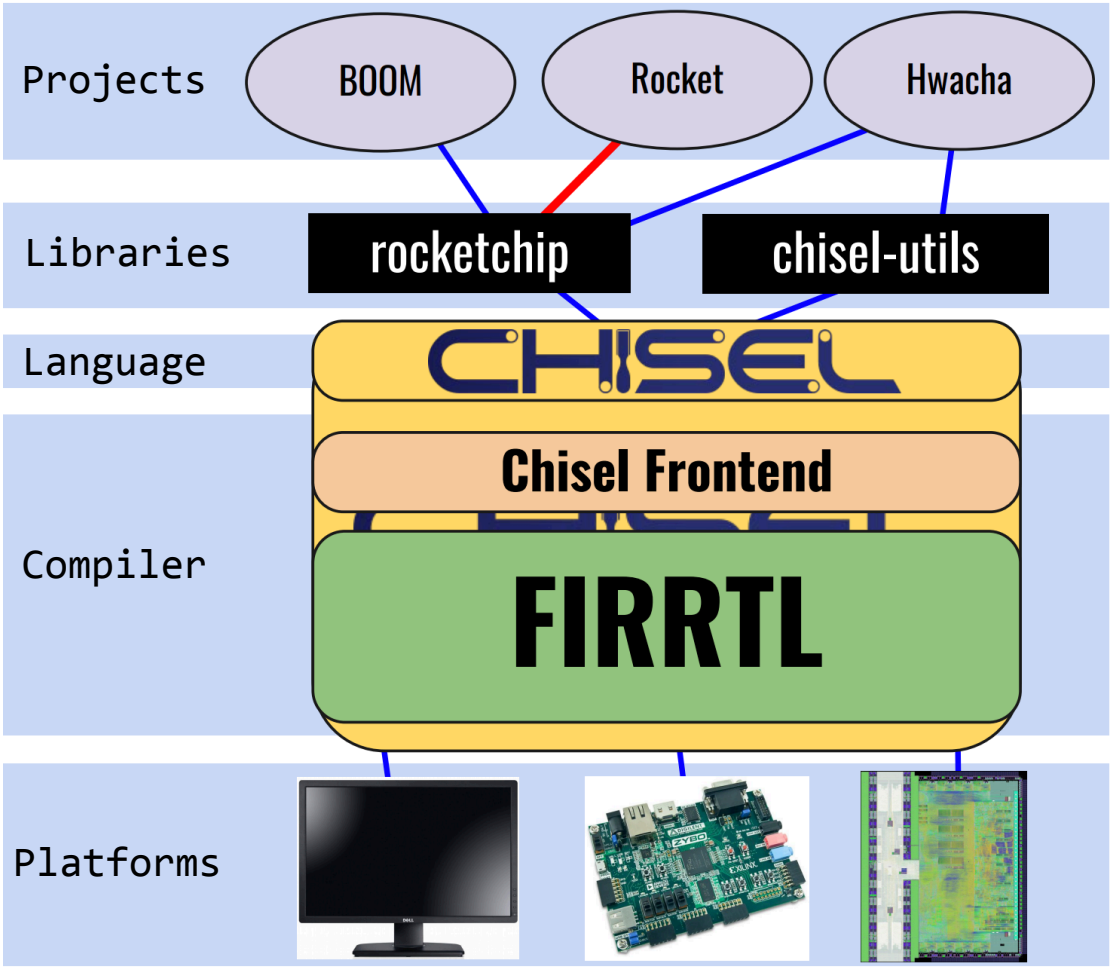


Figure 2.3: Hardware development stack with Chisel and FIRRTL
(Courtesy of Adam Izraelevitz)

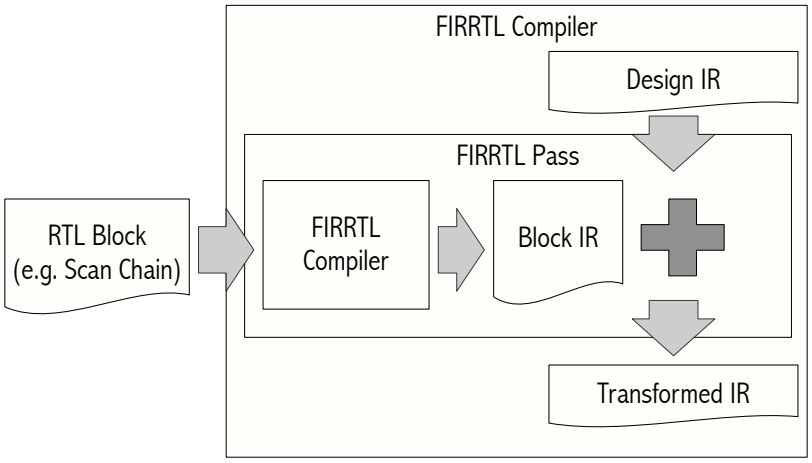


Figure 2.4: Compilers-in-a-pass for custom transforms

change their instantiations in RTL whenever new technologies are employed. Instead of manually modifying RTL, we may want to write a compiler pass to automatically map technology-independent memory blocks into technology-dependent macros.

The FIRRTL compiler [64] is a language-agnostic framework to support compiler passes operating on its well-defined intermediate representation, FIRRTL (Flexible Intermediate Representation for RTL) [91] [3]. A FIRRTL design in memory is represented as an abstract syntax tree (AST) structure consisting of IR nodes, each of which is one of the following IR abstract classes: `circuit`, `module`, `port`, `statement`, `expression`, `type`. Therefore, compiler passes in the FIRRTL compiler in general transform or analyze a given circuit by recursively visiting IR nodes based on their class types.

Custom transforms in the FIRRTL compiler is the crux of methodologies in this thesis. Indeed, most of techniques in this thesis would have been unavailable if it were not for the FIRRTL compiler. We will introduce various custom transforms and instrumentation techniques using the FIRRTL compiler throughout this thesis.

2.2.1 Compilers-in-a-Pass

Notably, for a wide range of instrumentations, it may be very tedious and cumbersome to express logic blocks only using the FIRRTL IR nodes that are appended to the target design. A core technique to reduce this burden is *compilers-in-a-pass* as shown in Figure 2.4. Instead of explicitly writing different IR code for different configurations of the same hardware module, we can automatically generate the corresponding IR to be instrumented from parameterized Chisel RTL using this technique, greatly reducing manual effort. This technique is used for various custom transforms

[3] Available at https://github.com/freechipsproject/firrtl.git

including scan chain insertion (Section 4.3.3) throughout this thesis.

2.3 Example RTL Designs

In this section, we introduce various RTL designs evaluated in this thesis. Since our goal is general evaluation and verification methodologies for any RTL designs, we need a various range of target designs from simple to complex.

To test and validate our ideas on evaluation methodologies, we should always start from extremely simple examples such as the greatest common divisor (GCD). If some idea does not work for these small examples, there is no way it will work for more complicated designs. Before moving on with real-world designs, our methodologies are tested with RISC-V mini (Section 2.3.1) as an intermediate example. Finally, to see the effectiveness of our methodologies, we should apply these methodologies to more realistic hardware designs such as RocketChip (Section 2.3.2), BOOM (Section 2.3.3), and Hwacha (Section 2.3.4).

Even though all RTL examples in this section are written in Chisel, for methodologies introduced in this thesis, there is no fundamental limitation on the frontend language in which the target design is written. In fact, all custom transforms implemented in this thesis operate on the language-agnostic IR in the FIRRTL compiler, and thus, designs in any frontend language such as Verilog should work with our frameworks once this frontend language is supported by FIRRTL.

2.3.1 RISC-V Mini

RISC-V mini is a simple RISC-V 3-stage pipeline (Figure 2.5) written in Chisel. It has served as a crucial example in various project developments, including Chisel3, FIRRTL, and simulation and verification methodologies. It implements RV32I of the user-level ISA version 2.0 [138] and the machine-level ISA of the privileged architecture version 1.7 [140]. Unlike other simple pipelines, it also contains simple direct-mapped instruction and data caches.

RISC-V mini is now open-source and publicly available [4]. In addition, we can run custom C programs on RISC-V mini as a script is provided to install the necessary RISC-V tools.

From my experience, RISC-V mini could catch many idea and implementation bugs of our methodologies not found by simpler examples. On the other hand, when tests with RISC-V mini passed, our methodologies worked pretty well with much more complex designs like RocketChip and BOOM, too. This is why RISC-V mini plays a central role in developing methodologies in this thesis although there is no evaluation with it.

[4] https://github.com/ucb-bar/riscv-mini.git

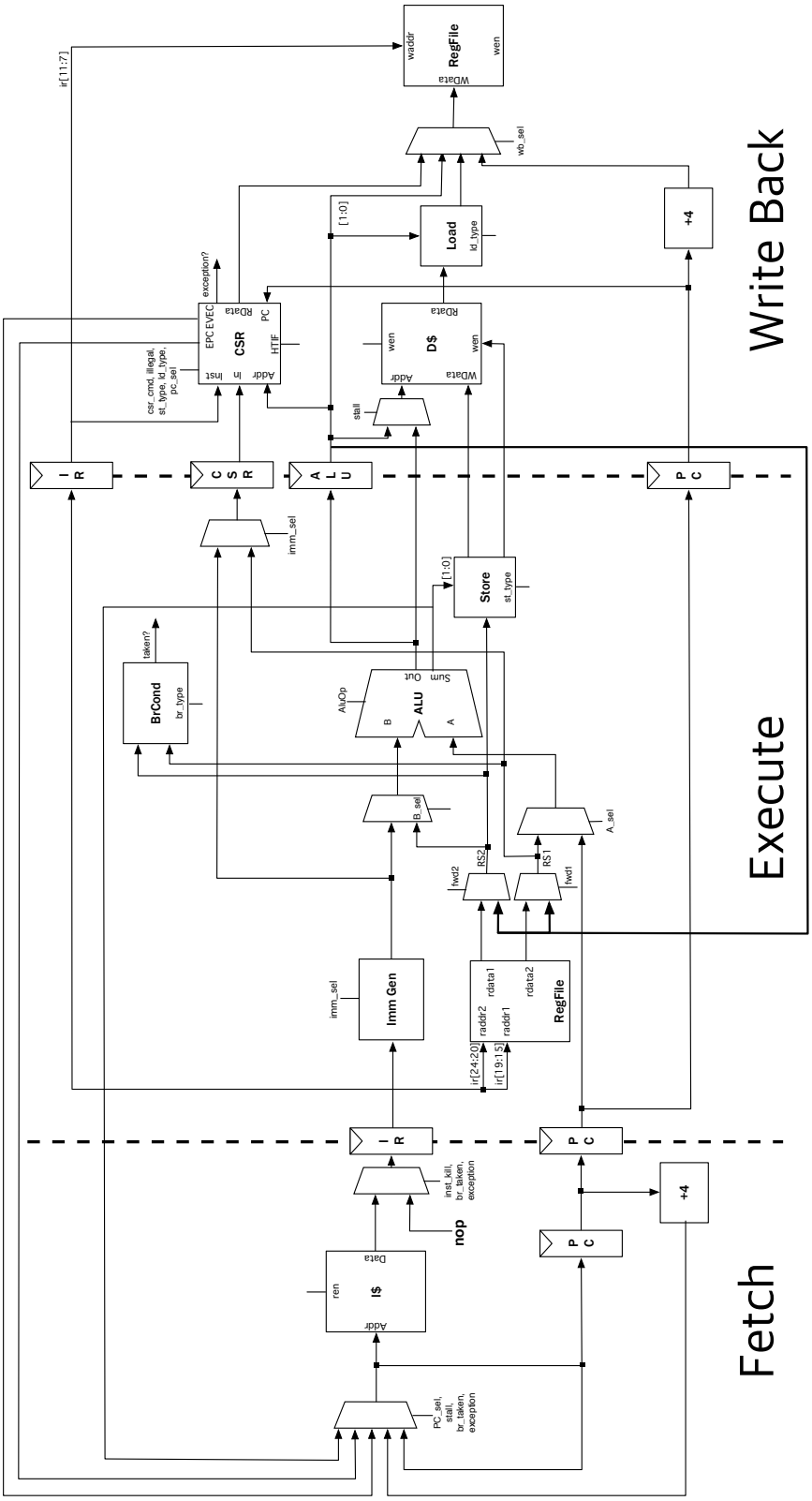


Figure 2.5: RISC-V mini pipeline

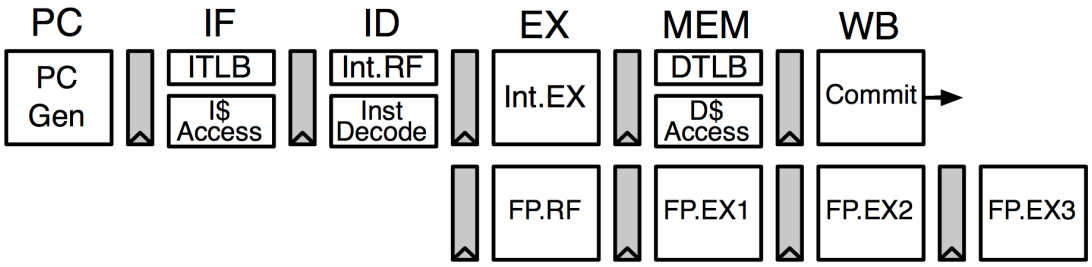


Figure 2.6: Rocket core pipeline
(Source: [8])

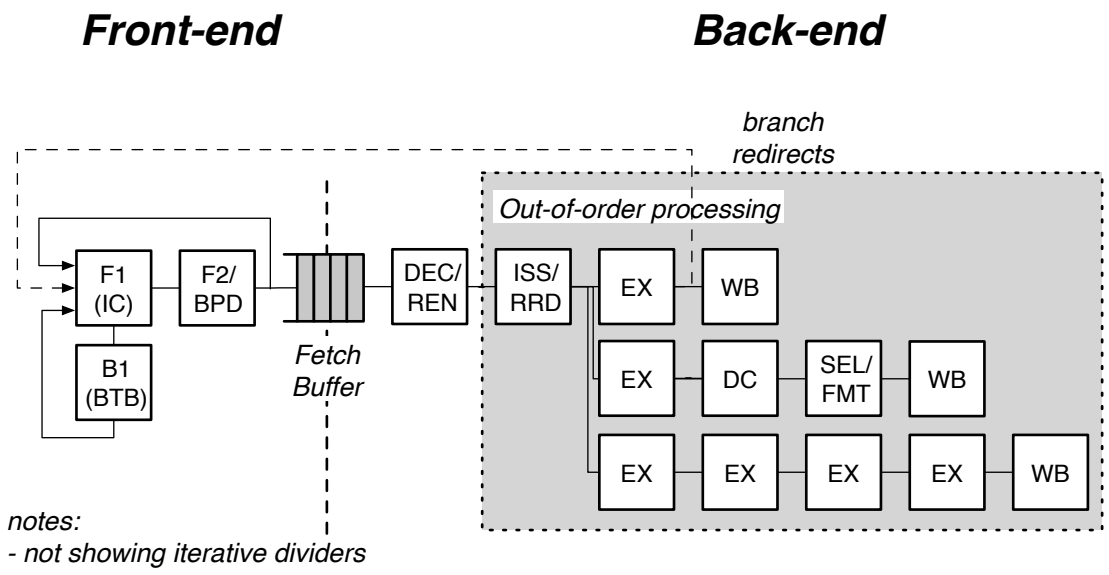


Figure 2.7: BOOM core pipeline
(Courtesy of Christopher Celio)

2.3.2 RocketChip Generator

RocketChip [8] is an open-source SoC generator suitable for research and industrial purposes. Rather than being a single instance of an SoC design, RocketChip is a hardware design generator, capable of producing many design instances from a single piece of Chisel source code. Multiple industry products as well as silicon prototypes are manufactured using RocketChip. A RocketChip instance generally consists of three major components: processors, a cache hierarchy, and an uncore.

RocketChip instantiates an in-order processor, Rocket, by default, but also supports various core implementations including an out-of-order processor, BOOM. Rocket is a 5-stage in-order processor (Figure 2.6) that implements the RISC-V ISA [141, 139]. It has an MMU that supports page-based virtual memory, a non-blocking data cache, and a frontend with branch prediction. Branch prediction is configurable and provided by a branch target buffer (BTB) with its associative branch history table (BHT), and a return address stack (RAS).

A RocketChip cache hierarchy can include L1 instruction caches, L1 blocking or non-blocking data caches, and TLBs with configurable sizes, associativities, and replacement policies. A RocketChip uncore consists of networks of cache coherent agents and the associated cache controllers for multi-core systems. These components are shared across both Rocket and BOOM instances.

2.3.3 BOOM

BOOM [34, 32] is a superscalar out-of-order processor with a unified physical register file (Figure 2.7) with configurable fetch widths, issue widths, and instruction window sizes. BOOM supports full branch speculation using a BTB, RAS, and a parameterizable backing branch predictor. BOOM is written in only 14K lines of Chisel code as it reuses many of RocketChip's components.

BOOM v2 [33] is a major microarchitecture update on BOOM v1, motivated by the tapeout process with the TSMC 28 nm technology. Major changes are i) the issue window and physical register file have been distributed, and ii) an additional cycle has been added to the fetch and rename stages. However, since BOOM v2 went through dramatic design changes in a short period of time, its RTL debugging with long-running applications became a big challenge, which is a strong motivation of Chapter 4.

2.3.4 Hwacha

Hwacha [87, 85, 86, 84] is a decoupled-vector accelerator within the RocketChip SoC. Hwacha is connected to Rocket as a co-processor through the Rocket Custom Coprocessor (RoCC) interface (Figure 2.8). Therefore, Hwacha's instructions are

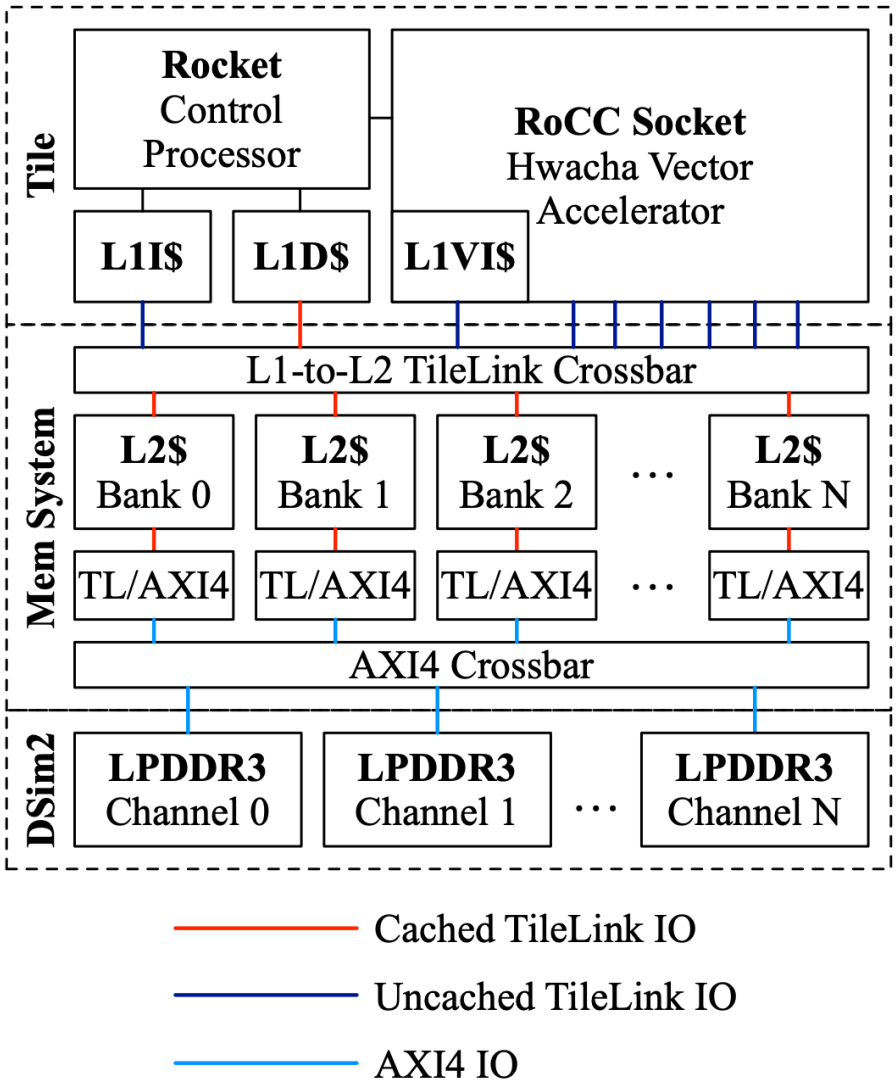


Figure 2.8: Hwacha as a co-processor in the RocketChip SoC
(Source: [85])

encoded as a custom extension of RISC-V. The Hwacha ISA manual [87] explains custom instructions and the programming model for Hwacha.

Hwacha achieves high performance by i) executing multiple data operations with a single instruction, and ii) having address calculations run ahead of data operations. In this thesis, Hwacha serves as an example of non-conventional hardware designs for the evaluation of runtime power modeling in Chapter 6.

2.4 More Challenges in RTL Implementations

Even though Chisel and FIRRTL greatly improve hardware design productivity, there are still remaining challenges for RTL implementations, which stems from the fact that *contemporary hardware designs are fairly complicated and their real-world applications are very long for detailed simulation.*

2.4.1 Performance Evaluation

An RTL design needs to be simulated for its performance evaluation before expensive tape-out. There are various commercial or open-source software RTL simulators available. However, these simulators are far slower than real machines, with which we can only run microbenchmarks for target designs.

On the other hand, FPGA emulation is fast enough to execute real-world applications to completion on complex hardware designs. However, performance evaluation with FPGA emulation can be misleading unless the timing of the DRAM and the I/O peripherals are properly modeled. Moreover, FPGAs are lack of visibility and controllability, which makes debugging extremely difficult.

In Chapter 3, we present FPGA-accelerated RTL simulation for accurate performance modeling on the FPGA, which is also as fast as FPGA emulation.

2.4.2 Verification and Debugging

RTL verification and debugging may be the biggest challenge for hardware designs because:

- **Formal verification** is not scalable, and therefore, cannot be used for system-level verification.
- **Software simulation** is too slow to execute real-world applications. Also, it is very hard to generate small input sets that exercise conner cases during software simulation.
- **FPGA emulation** lacks controllability and visibility. When the execution crashes, it is extremely hard to figure out the root of the error.

In Chapter 4, we present an effective RTL debugging methodology using FPGAs.

2.4.3 Power and Energy Efficiency

Power and energy efficiency are primary constraints for hardware designs. First of all, power is highly correlated with heat dissipation. We do not want our systems melt down from high heat dissipation. Moreover, high power dissipation may affect the timing and the functionality of the silicon implementation.

On the other hand, energy efficiency is important for embedded systems because energy is a precious resource for these systems. For example, we want batteries in our smart phones to last as long as possible. Also, energy efficiency is crucial to reduce the cost of operation for large-scale systems such as datacenters. The more energy these systems burn per workload, the more money we should pay for computing as well as cooling.

Unfortunately, measuring power and energy efficiency is also non-trivial because dynamic power is a function of signal activities, which are another function of applications running on the system. Therefore, we should collect signal activities from the simulation of each application for power and energy estimation.

Existing power and energy modeling methodologies fall into the following categories:

- **Commercial CAD tools** such as Synopsys PrimeTime PX provide the most accurate power and energy estimation with RTL/gate-level simulation. However, these tools are extremely slow for complex hardware designs executing real-world workloads.

- **Analytic power modeling** such as McPAT [92] is a de-facto methodology for computer architecture research. However, this model needs to be carefully validated against RTL or the silicon implementation, which is very challenging for novel hardware designs.

- **Performance counters** can be used for runtime power estimation, which is successful for traditional microprocessors in real machines. However, this methodology requires designers' careful intuition about which signals or events are highly correlated with dynamic power dissipation. For novel hardware designs, it can be very difficult to manually find key signals for power dissipation.

In Chapter 5, we present sample-based energy modeling for fast and accurate average power and energy estimation. In Chapter 6, we present an activity-based runtime power modeling, which automatically selects key signals for dynamic power. This runtime power modeling is useful for dynamic power/thermal optimizations such as dynamic voltage and frequency scaling (DVFS).

Chapter 3

FPGA-Accelerated RTL Simulation

In this chapter, we present a methodology to simulate RTL designs hosted on FPGAs along with their abstract memory and I/O device models. Section 3.1 describes the motivation of our methodology over existing frameworks for RTL evaluation and verification. Section 3.2 describes MIDAS v1.0, our initial framework for FPGA-accelerated RTL simulation in detail. Section 3.3 shows performance evaluation results for Rocket [8] and BOOM [34] using MIDAS. Section 3.4 summarizes this chapter.

3.1 Motivation: Efficient and Effective Framework for RTL Evaluation and Verification

For efficient and effective RTL evaluation and verification, an RTL simulation framework is expected to have the following properties:

- **Fast:** RTL simulation should be fast enough to run realistic workloads to completion.
- **Accurate:** RTL simulation should provide accurate performance, power, and energy evaluations.
- **General and easy-to-use:** An RTL simulator should be automatically generated from any RTL design.
- **Deterministic:** The results of RTL simulation should be always the same whenever repeated.

- **Controllable:** RTL simulation should be paused and resumed whenever necessary. Moreover, its internal state and signal values should be loaded and changed when required.
- **Visible:** The internal state and signals of RTL simulation should be visible to the outside world.
- **Portable:** RTL simulation should be hosted on various platforms.
- **Affordable:** The simulation framework should not be expensive for availability.

As the complexity of modern hardware designs increases, existing RTL simulation frameworks have failed to meet all these requirements. For example, software RTL simulation is too slow to run realistic workloads, while FPGA emulation lacks controllability and visibility. On the other hand, commercial emulation machines are extremely expensive, and thus, unavailable for most people.

For this reason, MIDAS is developed to meet all these requirements. First of all, we use FPGAs to enable RTL simulation for trillions of cycles at speed. Next, we use the FIRRTL compiler [64] to automatically generate the FPGA-accelerated RTL simulator from any RTL design, which makes MIDAS a general and easy-to-use framework. This FPGA-accelerated RTL simulator is an instance of synchronous data flow (SDF) [81] that ensures accurate timing modeling as well as deterministic and controllable simulation on the FPGA. The controllability, visibility, and the deterministic execution of the FPGA-based simulator can be further enhanced by additional compiler passes as discussed in Chapter 4. Finally, MIDAS is portable and affordable as the FPGA-accelerated simulator can be hosted either on cheap FPGA boards or on the FPGA cloud such as Amazon EC2 F1 instances without large initial capital expense.

3.2 MIDAS v1.0: Open-Source FPGA-Accelerated RTL Simulation Framework

In this section, we present the implementation details of MIDAS v1.0, an open-source FPGA-accelerated RTL simulation framework[1]. Section 3.2.1 describes its overall tool flow to generate the FPGA-accelerated simulator automatically from any RTL design using the FIRRTL compiler. Section 3.2.2 describes compiler transforms for accurate timing modeling on the FPGA. Section 3.2.3 describes how various simulation models can be hosted together on the heterogeneous FPGA platform for high-speed RTL simulation.

[1] https://github.com/ucb-bar/midas-release

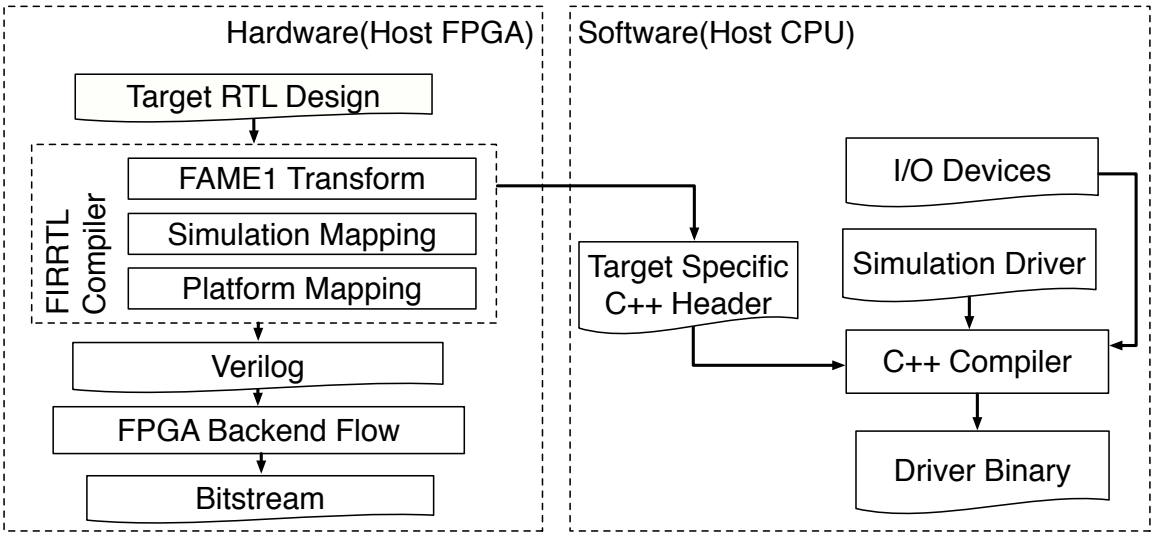


Figure 3.1: Tool flow to generate FPGA-accelerated RTL simulators

3.2.1 Tool Flow with FIRRTL Compiler Passes

For MIDAS to be general and easy-to-use, the target RTL design needs to be automatically transformed and instrumented without designers' manual effort. For this purpose, we use the FIRRTL compiler passes to transform RTL into an FPGA-hosted model. Figure 3.1 describes the complete tool flow for MIDAS.

For FPGA-hosted RTL models, the *FAME1 Transform* is performed to decouple the target clock from the host clock, while *Simulation Mapping* is applied to augment the logic for timing token communications between simulation models, which is explained in more detail in Section 3.2.2. *Platform Mapping* glues RTL models as well as abstract FPGA models together by adding platform-specific logic as explained in more detail in Section 3.2.3. Note that *Simulation Mapping* and *Platform Mapping* use the compilers-in-a-pass technique (Section 2.2.1) to generate their necessary logic.

To control the FPGA-accelerated RTL simulator, the simulation driver that runs on the host CPU is built by compiling a generated, target-specific header, which contains a memory map of software-addressable state on the FPGA-host; the simulation driver source code, which controls the advance of simulation; and software models of I/O devices not hosted on the FPGA. Once the host FPGA has been programmed, the simulation driver initiates the simulator state and then commences simulation and advances the I/O device models as soon as it is possible to do so.

Note that this framework is language-agnostic as all custom transforms in this thesis are implemented as compiler passes in the FIRRTL Compiler. Once the target design is translated into FIRRTL from its language frontend, we can apply the compiler passes presented in this thesis regardless of the design's host HDL.

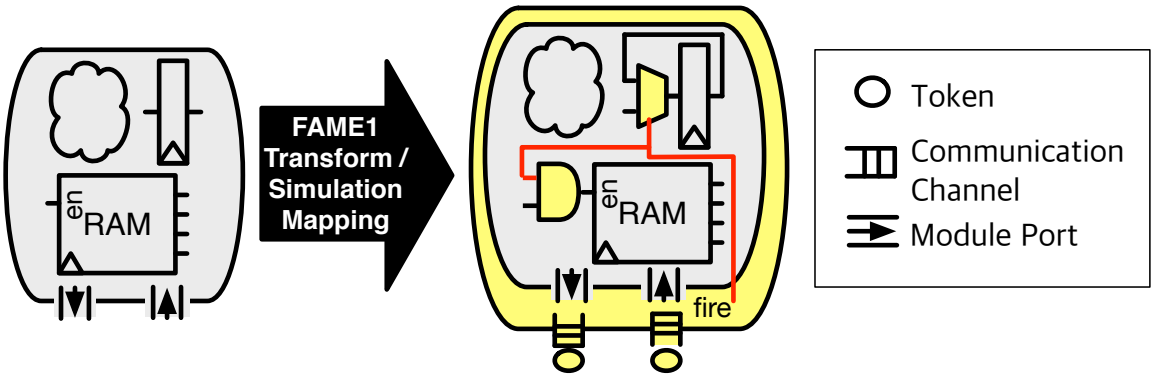


Figure 3.2: FAME1 Transform and Simulation Mapping

3.2.2 FAME1 Transform and Simulation Mapping

The *FAME1 Transform* and *Simulating Mapping* automatically generate a FAME1 model from any RTL design, similar to token-based timing simulators manually implemented in previous work [132, 130, 38, 36, 111]. Note that these simulators are also instances of synchronous dataflow [81].

Figure 3.2 depicts an automatic transformation on an arbitrary RTL design to generate a FAME1 model. The *FAME1 Transform* adds a mux, globally enabled by the `fire` signal, in front of each register allowing it to capture its own output. For memory blocks, this transform masks their enable signals with the `fire` signal. As a result, this compiler pass enables the entire design to stall when `fire` is not set by external events.

In addition, *Simulation Mapping* attaches communication channels, which buffer timing tokens from other simulation models, to the I/O ports of the FAME1 model. The FAME1 model asserts `fire` when timing tokens for all input communication channels are available, and simulates one cycle by consuming input timing tokens and generating output timing tokens. On the other hand, the FAME1 model stalls when any input communication channel is empty or any output communication channel is full.

These transformations allow heterogeneous simulation models to run decoupled, which is an important optimization when all components including software models cannot be hosted on a single FPGA as shown in Section 3.2.3.

3.2.3 Platform Mapping

Another challenge is how to map heterogeneous simulation models, including FAME1 models, software models, and abstract timing models hosted on the FPGA, to the FPGA host platform for fast simulation. Figure 3.3 shows how the target designs are mapped to the FPGA host platform.

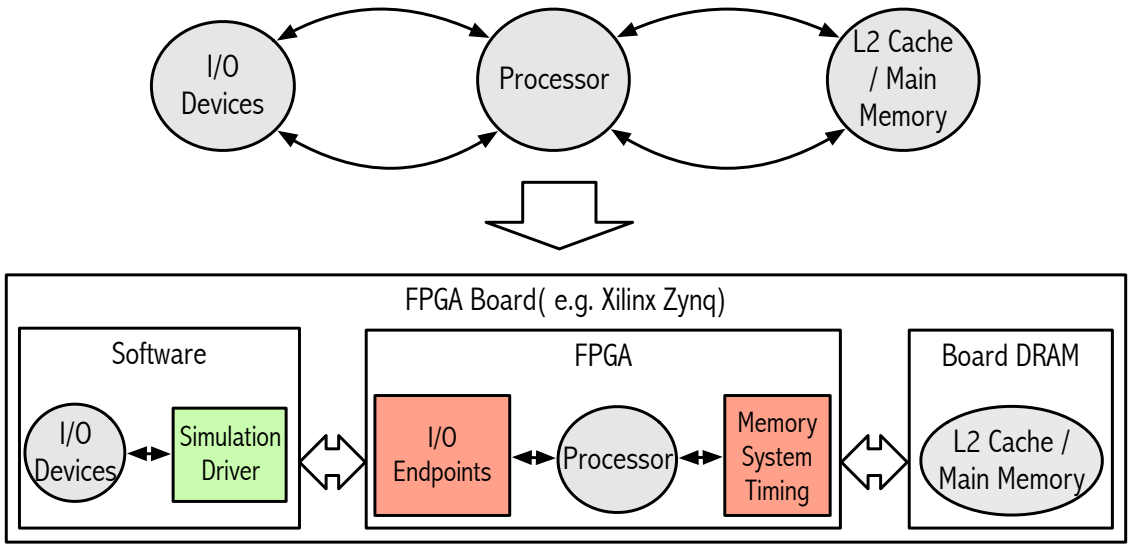


Figure 3.3: Target design mapping to the FPGA host platform

The processor models are generated from RTL designs and mapped to the FPGA as described in Section 3.2.2. The I/O devices are modeled in software and run alongside the simulation driver as their transactions are infrequent. However, without careful optimizations, the simulation rate is not fast enough to run real-world workloads to completion due to frequent timing token exchanges between the CPU-hosted software models and the FPGA-hosted RTL models.

A novel technique to optimize the communications between the CPU and the FPGA is *I/O endpoints*, special hardware widgets that translate low-level timing tokens to high-level transactions and vice versa. In other words, instead of exchanging low-level timing tokens, the I/O devices efficiently communicate the processor with high-level transactions through I/O endpoints only when necessary. As a result, this technique greatly improves the simulation rate very close to the FPGA operating frequency.

We implement an abstract timing model hosted on the FPGA for last-level caches (LLCs) and DRAM, while their actual data are hosted on the board main memory. Using an abstract model for the LLC was compelling for three reasons. Firstly, LLCs are missing in the open-source version of RocketChip, it is easier at first to write an abstract LLC model than to implement a complete LLC. Second, abstract models can provide runtime configurable parameters with low overhead. In our case, the size, associativity, and the latency of the LLC can be reconfigured without needing to recompile an FPGA bitstream. Finally, since we model the timing of the memory systems by only keeping the tags of LLCs on the FPGA, we can model an LLC that would be too large to fit on our FPGA. However, the downside is the simulation needs to stall even with LLC cache hits to obtain the actual data from the board

Parameter	Rocket	BOOM-2w
Fetch-width	1	2
Issue-width	1	3
Issue slots	-	20
ROB size	-	80
Ld/St entries	-	16/16
Physical registers	32(int)/32(fp)	110
Branch predictor	-	gshare: 16 KiB history
BTB entries	40	40
RAS entries	2	4
MSHR entries	2	2
L1 $ capacities	16 KiB or 32 KiB	
ITLB and DTLB reaches	128 KiB / 128 KiB	
L2 $ capacity and latency	1 MiB / 23 cycles	
DRAM latency	80 cycles	

Table 3.1: Target processors evaluated with MIDAS v1.0

main memory, slightly slowing down the simulation rate.

MIDAS has been hosted on the Xilinx Zynq boards, Amazon EC2 instances [1], and Microsoft Catapult [29]. However, thanks to the portability of MIDAS, other FPGA platforms can also be used in principle.

3.3 Evaluation

3.3.1 Target Designs and Host Platform

We evaluated two open-source RISC-V processors, Rocket, a productized scalar in-order processor, and BOOM, an industry-competitive, open-source out-of-order processor, using MIDAS. Table 3.1 shows the processor configurations used for this study.

The processor and its L1 caches represent the design-under-test (DUT) and are supplied as RTL, which are automatically transformed into a FAME1 model (Section 3.2.2). On the other hand, the supporting L2 cache and the DRAM are implemented as abstract timing models, which can be configured at runtime (Section 3.2.3).

We used the Xilinx Zynq ZC706 board for performance evaluation. The average simulation rate of BOOM-2w was 18 MIPS for the SPECint2006 benchmark suite.

3.3.2 Memory System Timing Model Validation

Since we introduce abstract timing models for the LLC and the DRAM, these models should be carefully validated. Specifically, we should validate 1) the size of the LLC, and 2) the latencies of the LLC and the DRAM.

Figure 3.4 shows the timing validation of the memory systems using `caches`, a pointer-chase benchmark from `ccbench` [30]. While `caches` runs, a pointer-chase through increasing sizes of arrays demonstrates the load-to-load latency of different levels of the memory hierarchy. In this validation, the target design observes the 16KiB L1 data cache (6 cycles), the 1MiB L2 data cache (6 + 23 cycles), and the main memory (6 + 23 + 80 cycles) as configured in Table 3.1.

3.3.3 Benchmarks

The SPECint2006 benchmark suite is widely used for computer architecture research as well as performance evaluation of real systems. However, only small fractions of the whole benchmark suite have been evaluated in computer architecture research with microarchitectural software simulators due to its non-trivial execution lengths as shown in Table 3.2. In our evaluations, all SPECint2006 benchmarks were compiled using Speckle [31] and simulated to completion. Unfortunately, `445.gobmk`, `456.hmmer`, and `462.libquantum` failed on the current version of BOOM-2w, and thus were excluded from our evaluations.

The DaCapo benchmarks [21] are widely used Java benchmarks that represent full Java applications. We run the DaCapo benchmarks on JikesRVM [5], a research Java Virtual Machine that is widely used in managed-language research. We used version 9.12 of the benchmark suite and excluded the benchmarks that did not run on recent versions of JikesRVM and the current version of the target processors. Table 3.3 also shows the dynamic instruction counts for the benchmarks with the default inputs running on BOOM-2w, accounting for class loading, Just-in-Time compilation, and garbage collection. We believe this provides a comprehensive and realistic view of Java applications.

We built an initramfs image including all necessary files for each benchmark that runs under RISC-V Linux kernel version 4.6.2. For complex workloads, such as our JVM, we built library dependencies using the Yocto (`riscv-poky`) Linux distribution generator.

3.3.4 Case Study: SPECint2006

Figure 3.5 shows the instructions per cycle (IPCs) of Rocket and BOOM-2w with the configurations in Table 3.1 across the SPECint2006 benchmark suite with its reference inputs. These IPCs are computed from the complete execution of each benchmark. Note that the parameters of BOOM-2w are chosen to approximate the

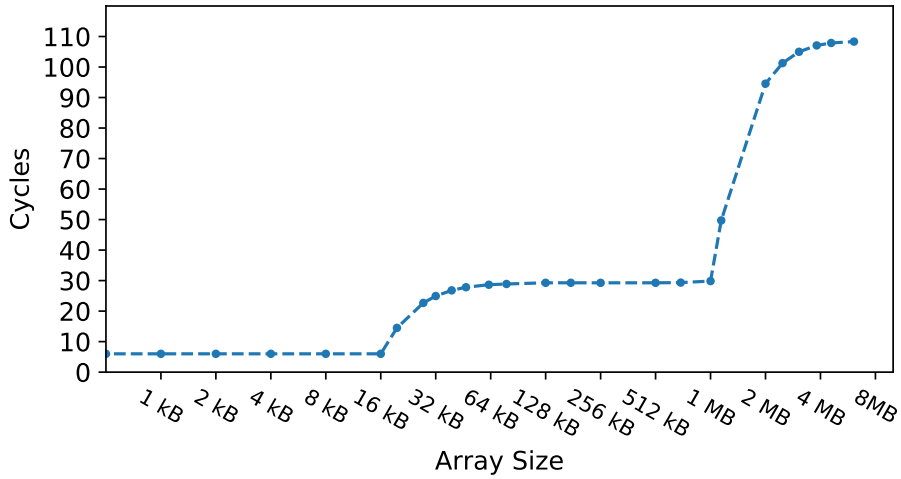


Figure 3.4: Memory system timing validation of BOOM-2w with the 16 KiB L1 data cache

Benchmark	Instructions (T)	Benchmark	Instructions (T)
`400.perlbench`	2.48	`458.sjeng`	2.85
`401.bzip2`	3.08	`462.libquantum`	2.09
`403.gcc`	1.37	`464.h264ref`	5.07
`429.mcf`	0.29	`471.omnetpp`	0.61
`445.gobmk`	2.04	`473.astar`	1.05
`456.hmmer`	2.95	`483.xalanbmk`	1.10

Table 3.2: Dynamic instruction counts for the SEPC2006int benchmark suite with the RISC-V ISA

Benchmark	Instructions (B)
`avrora`	137.0
`luindex`	48.0
`lusearch`	263.8
`pmd`	156.8
`xalan`	193.3

Table 3.3: Dynamic instruction counts for the DaCapo benchmark suite with the RISC-V ISA

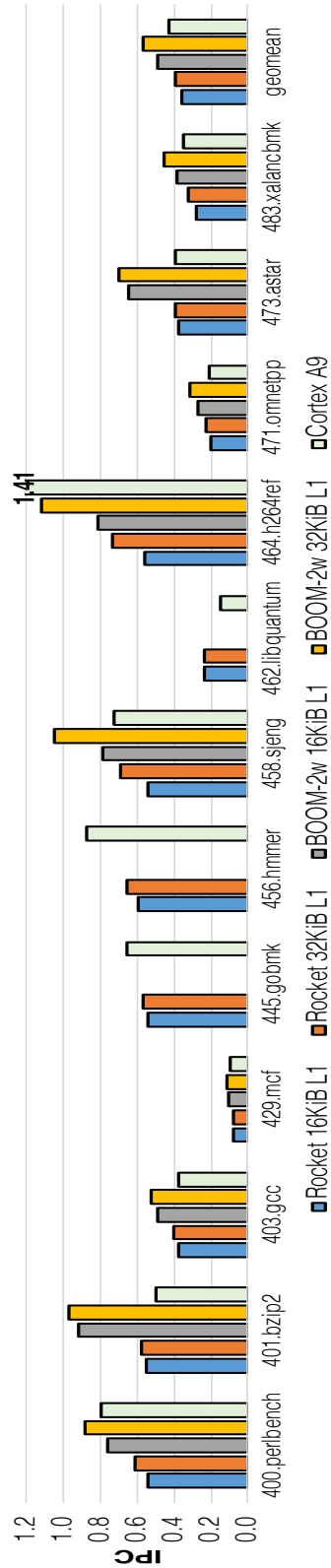


Figure 3.5: IPCs of Rocket, BOOM-2w, and Cortex A9 for the SPECint2006 benchmarks

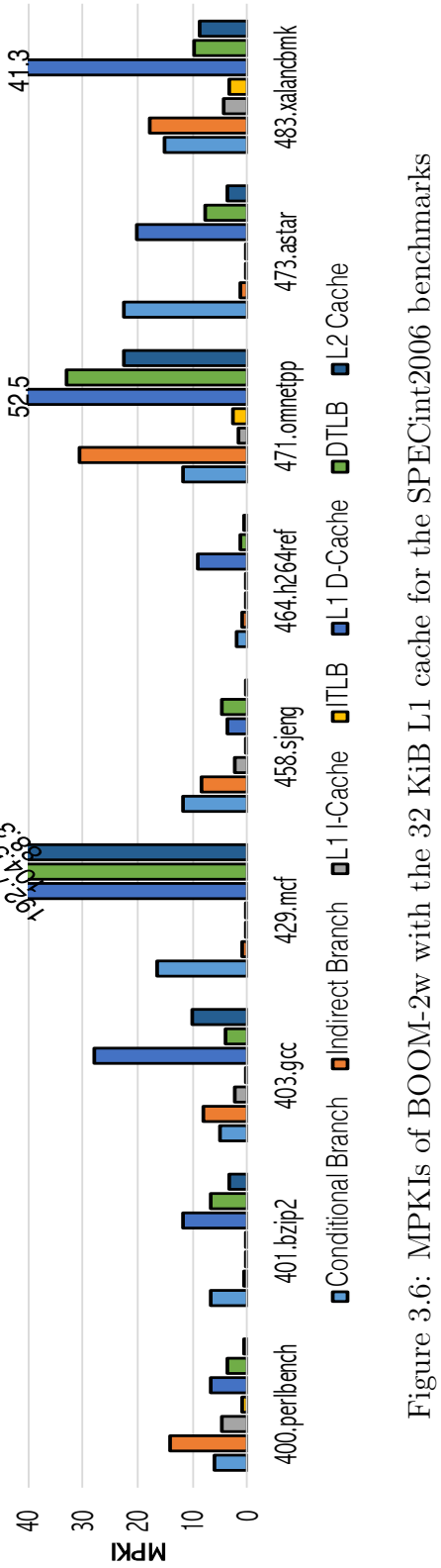


Figure 3.6: MPKIs of BOOM-2w with the 32 KiB L1 cache for the SPECint2006 benchmarks

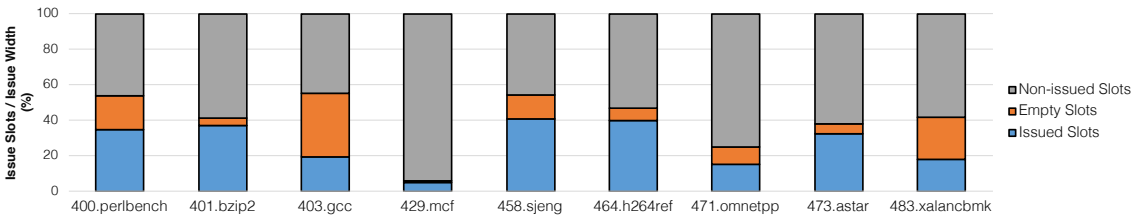


Figure 3.7: Issue queue utilizations of BOOM-2w for the SPECint2006 benchmarks

configuration of the ARM Cortex-A9 processor. For reference, the IPCs of ARM Cortex A9 for the SPECint2006 benchmark suite are also presented in Figure 3.5.

In this case study, we are mainly interested in the performance impact of the increase in the L1 cache sizes from 16 KiB to 32 KiB. As seen from Figure 3.5, there are big performance improvements for several benchmarks (e.g. `400.perlbench`, `458.sjeng`, `464.h264ref`) from this change. Therefore, it is desirable to have 32 KiB L1 caches unless it lengthens the critical path.

To understand the performance evaluations more deeply, we collect more performance statistics from the performance counters. Figure 3.6 shows the misses per kilo instructions (MPKIs) of BOOM-2w with the 32 KiB L1 caches. This gives us a hint for potential performance improvement. First of all, there is a big gap between the MPKIs of the L1 data cache and the L2 cache for `464.h264ref`, which explains why BOOM-2w underperforms ARM Cortex A9 for `464.h264ref`. Note that ARM Cortex A9 also has a 32 KiB L1 data cache. Therefore, having more aggressive prefetchers in the data cache may reduce the L1 data cache miss rate for `464.h264ref`. In addition, some benchmarks such as `471.omnetpp` and `483.xalancbmk` suffer from a large number of indirect branch mispredicts, which indicates the BTB size may need to be increased.

We can also figure out the pipeline utilization by examining the issue queue utilization. Intuitively, issued slots per cycle of the issue queue are equal to the IPC. As shown in Figure 3.7, more than 60 % of issue slots are wasted. The first case is issue slots are empty, which is caused by the frontend hazards including branch mispredicts and instruction cache misses, and/or lack of pipeline resources such as physical registers and re-order buffer (ROB) entries. For example, `403.gcc` exhibits lots of empty slots even though it has relatively small frontend MPKIs, which implies the benchmark wants more pipeline resources. Another case is issue slots are neither empty nor issued due to the backend hazards. For instance, `429.mcf` and `471.omnetpp` suffer from high data cache miss rates, therefore having a large fraction of non-empty non-issued slots.

We can also get comprehensive views on the time-based performance behaviors for each benchmark from its full execution with MIDAS. Figure 3.8, 3.9, 3.10, and 3.11 show performance traces for selected benchmarks. All instantaneous performance statistics are sampled every 100 million cycles from performance counters in BOOM-

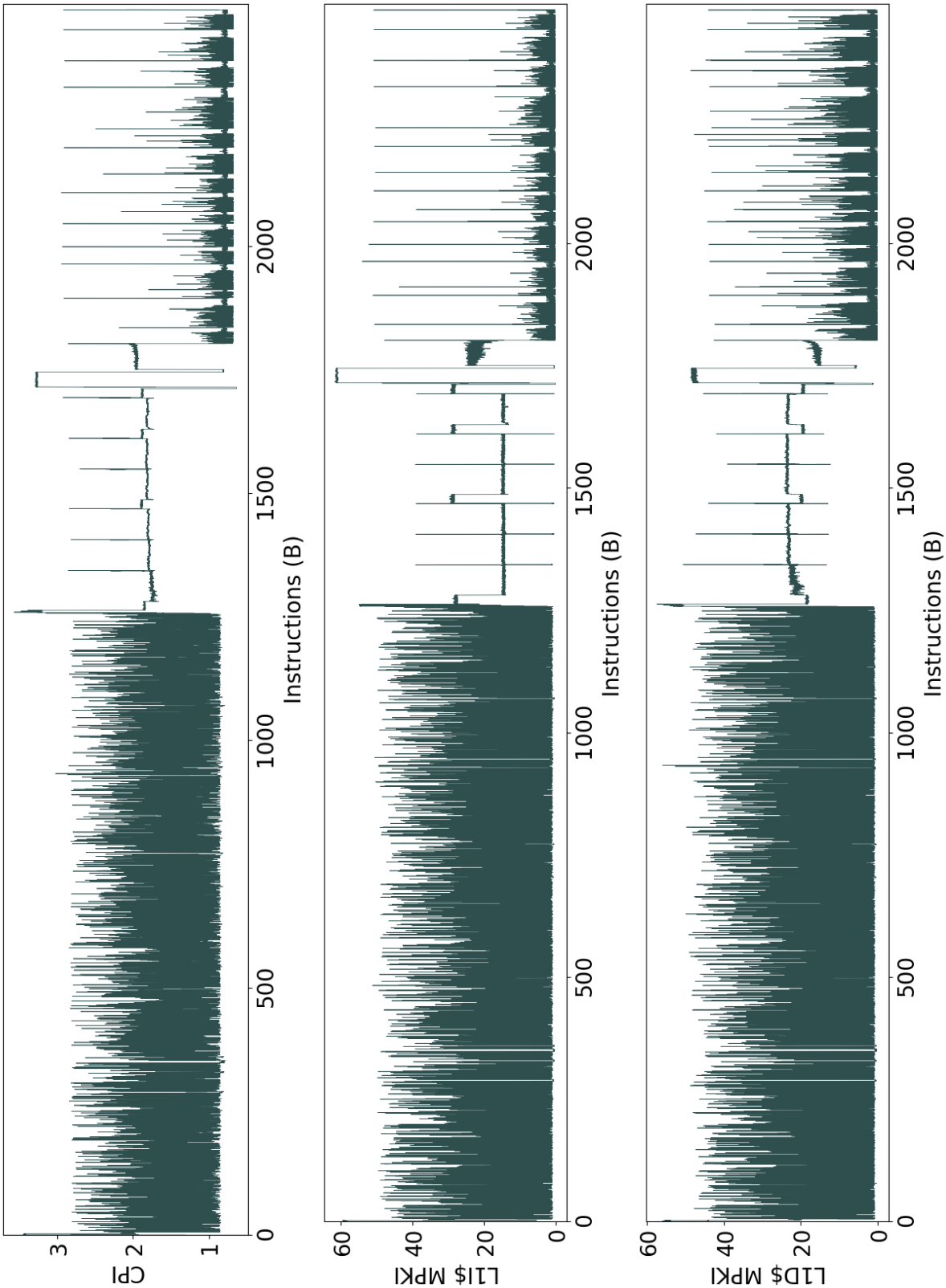


Figure 3.8: Performance traces for `400.perlbench` on BOOM-2w

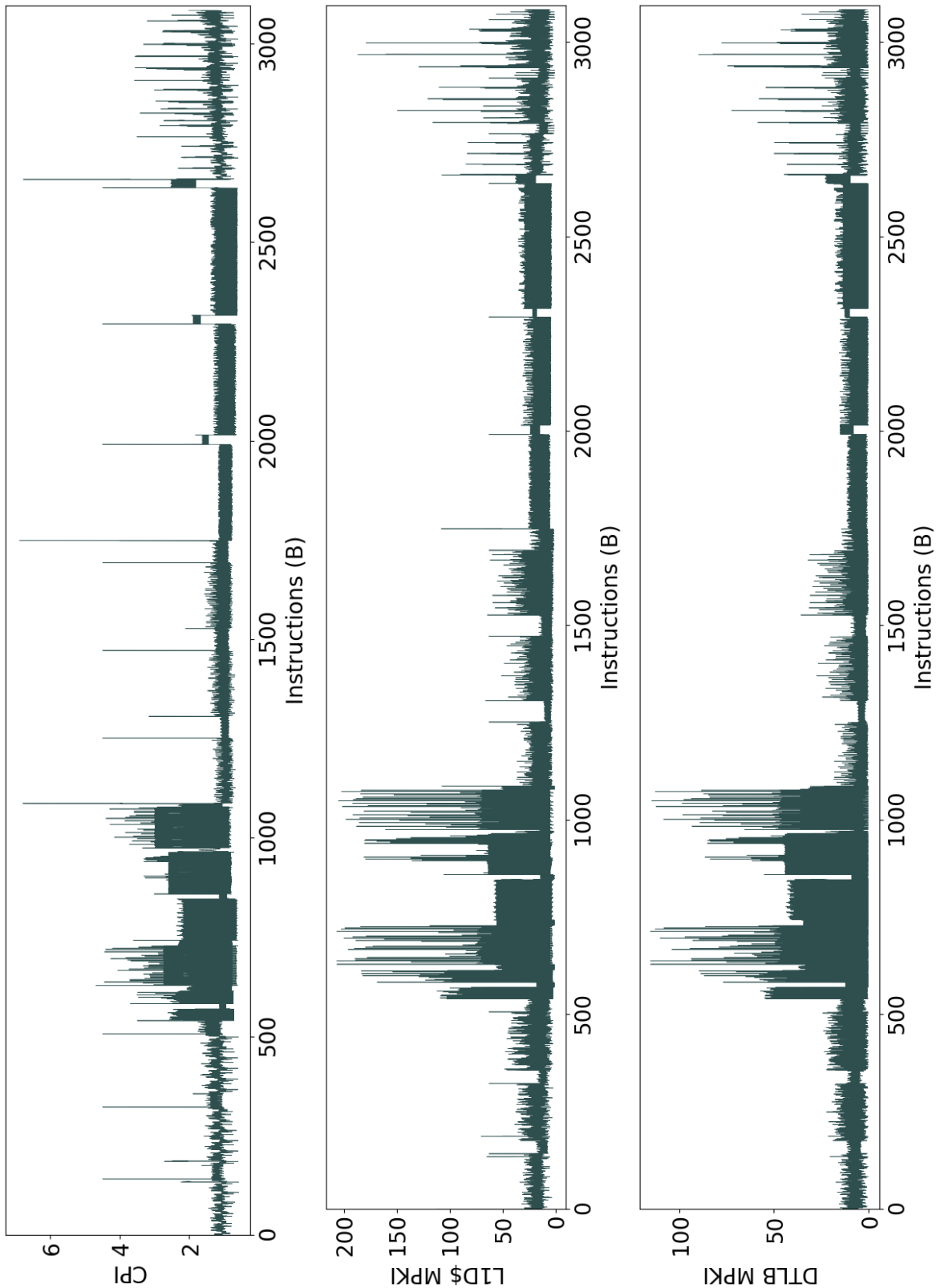


Figure 3.9: Performance traces for `401.bzip2` on BOOM-2w

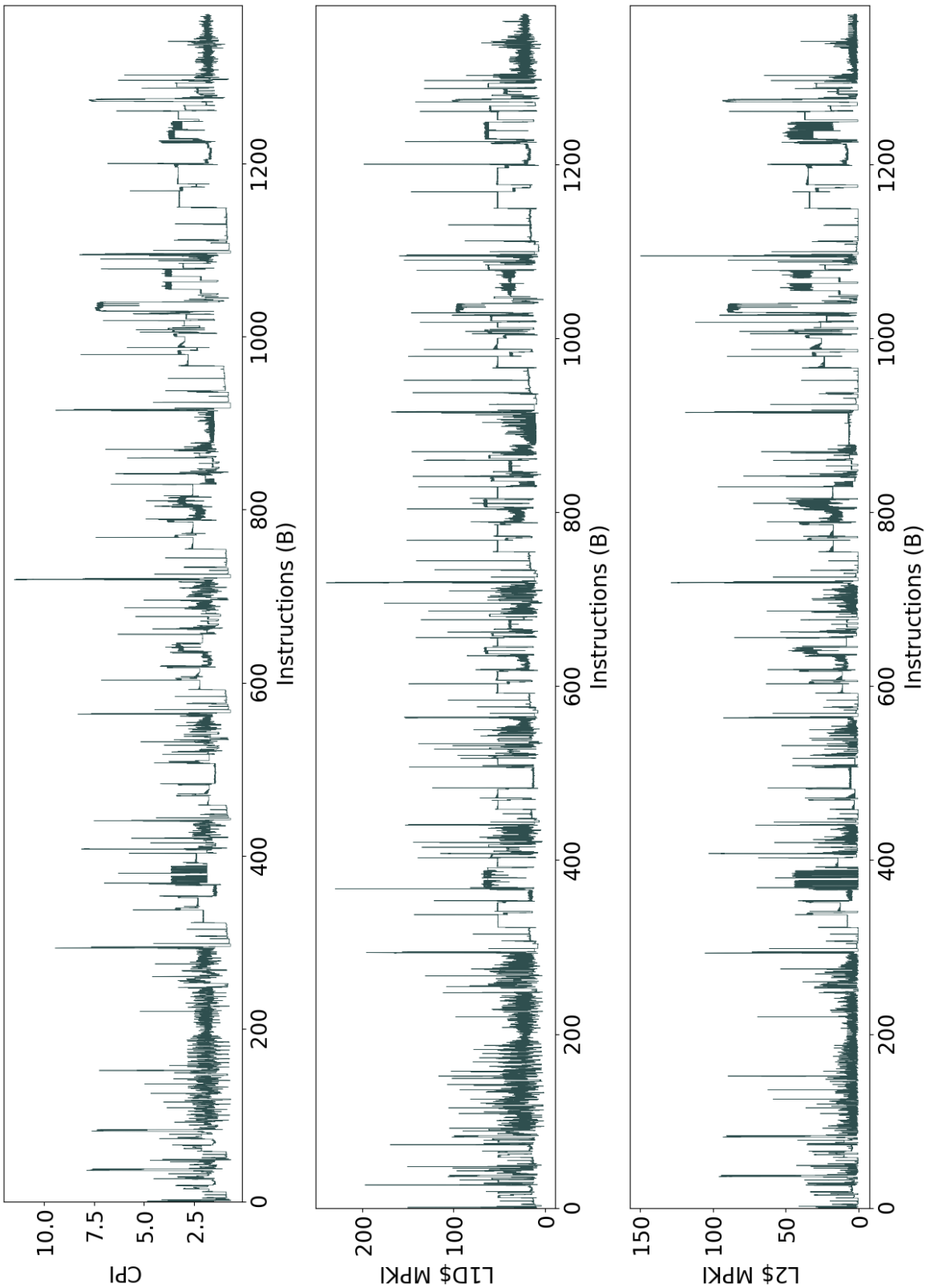


Figure 3.10: Performance traces for `403.gcc` on BOOM-2w

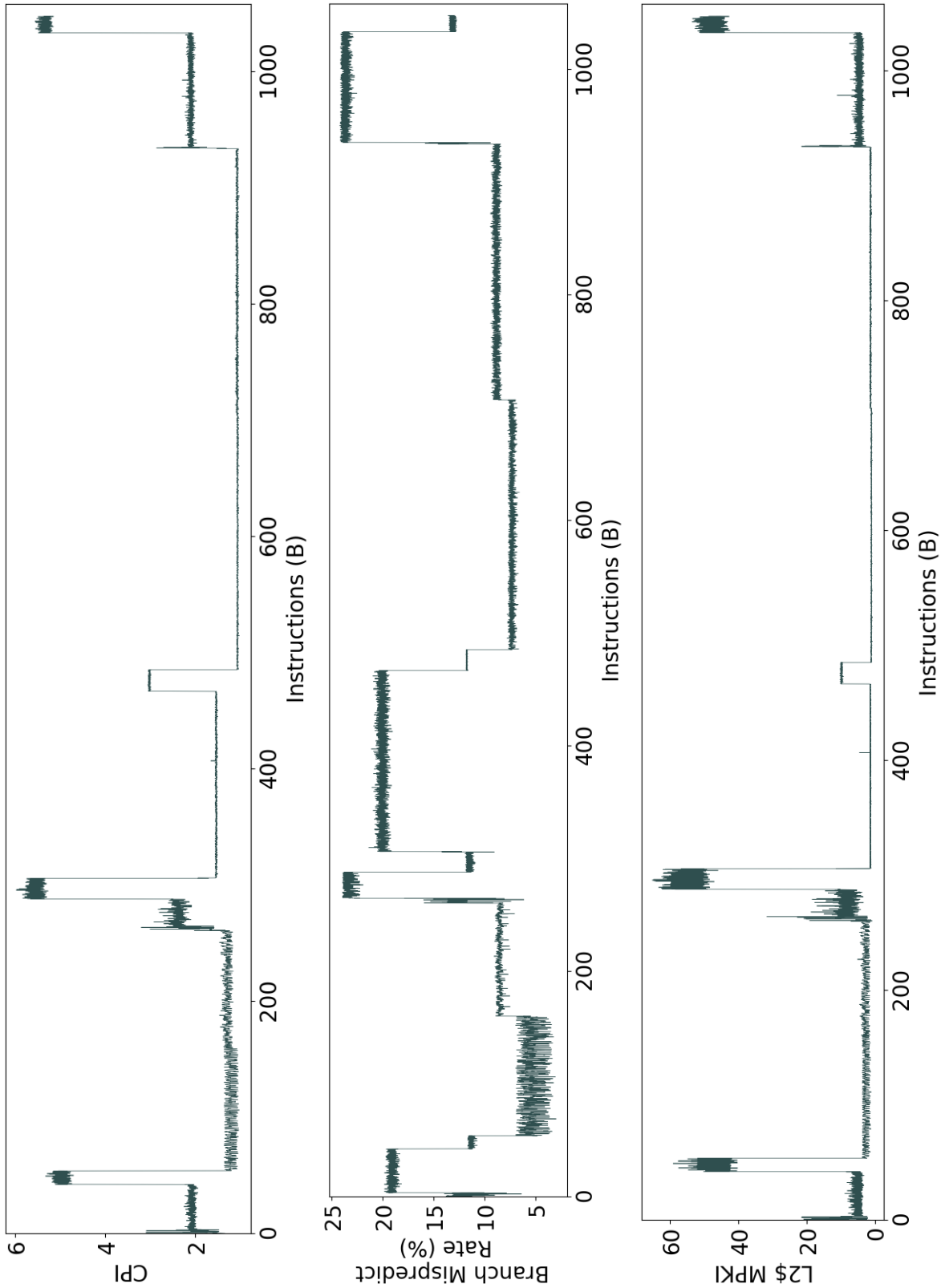


Figure 3.11: Performance traces for `473.astar` on BOOM-2w

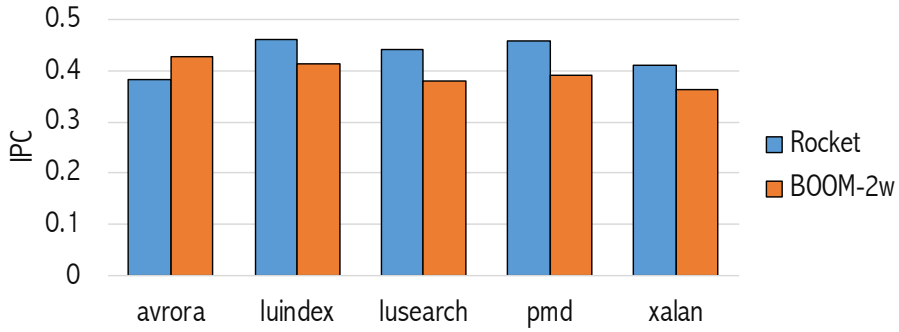


Figure 3.12: IPC with BTB-40 and 16KiB L1 for the DaCapo benchmarks

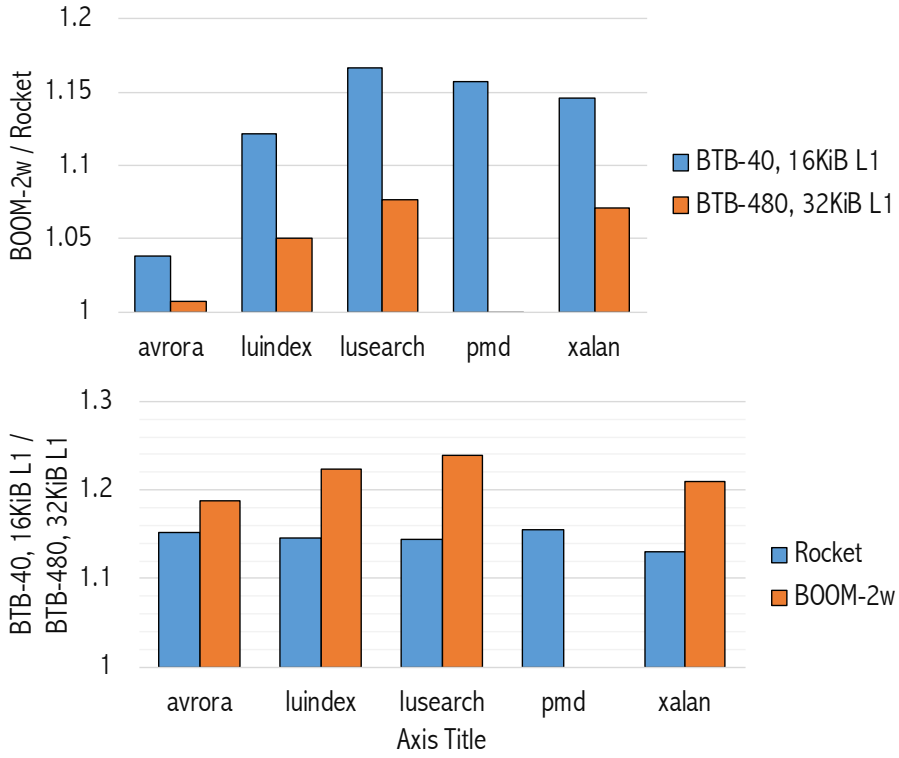


Figure 3.13: Total Cycle Ratios for the DaCapo benchmarks

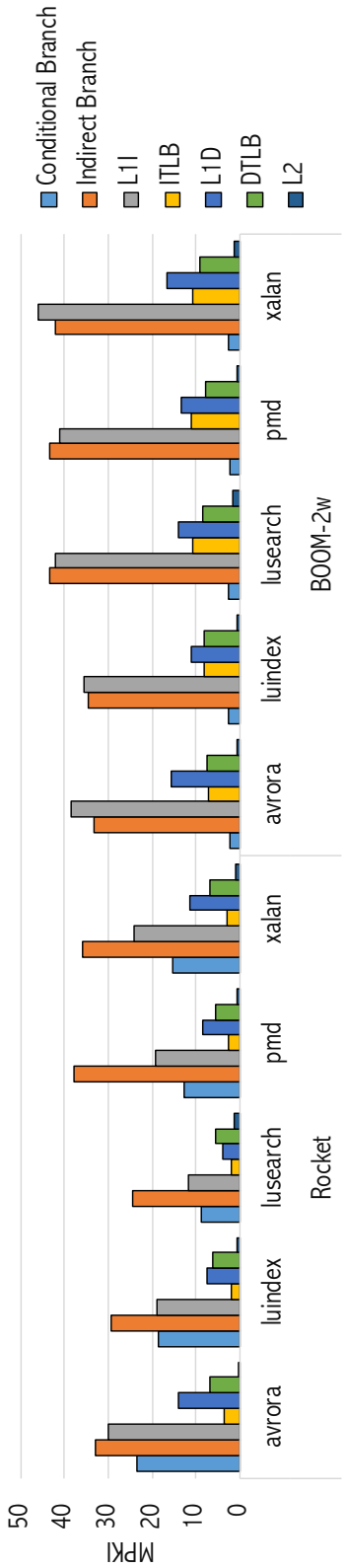


Figure 3.14: MPKIs with BTB-40 and 16 KiB L1 for the DaCapo benchmarks

2w. From this time-based analysis, we can get intuitions on which microarchitectural events are the most effective for each benchmark. For example, the instantaneous CPIs of `400.perlbench` are highly correlated with the instantaneous L1 instruction and data cache miss rates (Figure 3.8), while the instantaneous CPIs of `402.bzip2` change along with the instantaneous L1 data cache and DTLB miss rates (Figure 3.9). On the other hand, the instantaneous CPIs of `403.gcc` follow the instantaneous data cache miss rates (Figure 3.10), while the instantaneous CPIs of `473.astar` are highly affected by both the short-term branch misprediction rate and the spontaneous L2 cache miss rate (Figure 3.11). Also, time-based performance evaluations help figure out which parts of the code cause important microarchitectural events, enabling rapid hardware/software co-optimizations.

3.3.5 Case Study: DaCapo

The initial performance evaluations in Figure 3.12 and 3.13 show BOOM-2w underperforms Rocket for the DaCapo benchmarks. Interestingly, BOOM-2w spends more cycles than Rocket does for `avrora` even with the higher IPC. In general, BOOM-2w has lower IPCs than Rocket but similar instruction counts as Rocket for the DaCapo benchmarks.

To figure out the performance bottlenecks, we collect the microarchitectural event statistics as in Figure 3.14, which shows both Rocket and BOOM-2w suffer from high rates of L1 instruction cache misses and indirect branch mispredicts. This is even worse for BOOM-2w since the frontend hazards cause the underutilization of the pipeline backend. Moreover, misspeculations also incur unnecessary pipeline hazards. Therefore, the configurations in Table 3.1, which are good enough for the SPEC2006int benchmarks, are not good at all for these Java applications.

To reduce the instruction cache misses, we doubled the L1 cache sizes by increasing the associativity from four ways to eight ways. The number of sets cannot be increased by just changing the parameters because the L1 caches are virtually-indexed physically-tagged. Also, to reduce indirect branches mispredicts, we increased the entries of the fully-associative BTB from 40 to 480, which may be unrealistic for the silicon implementations. In reality, a small size fully-associative BTB can be backed by a large SRAM-based secondary BTB to increase the prediction accuracy for indirect branches. In this section, we want to see the potential performance improvement for bigger instruction caches and BTBs. Unfortunately, `pmd` does not run on BOOM-2w with this configuration.

Figure 3.13 also shows the speedups of Rocket and BOOM-2w with the new BTB and L1 cache parameters. Note that to compute the speedups for Java applications, we should use the total execution cycles instead of IPCs because dynamic instruction counts can change with different microarchitectures. BOOM-2w still underperforms Rocket, but the performance gains are larger with its new configurations. We believe BOOM will eventually outperform Rocket for Java applications if more aggressively

optimizations are introduced in the frontend. This case study also implies that we may need significant hardware/software co-optimizations for managed-language applications running on RocketChip and BOOM.

3.4 Summary

In this chapter, we presented MIDAS v1.0, an open-source FPGA-accelerated RTL simulation framework. MIDAS v1.0 is general and easy-to-use as the FPGA-accelerated RTL simulator is automatically generated from any RTL design. This simulator is not only fast, running at the FPGA speed and orders-of-order-magnitude faster than cycle-level microarchitectural software simulators, but also truly cycle-accurate, as it uses RTL identical to that used in the silicon implementation. As a result, MIDAS v1.0 enables the complete execution of the full-software stack for long-running applications with little loss of fidelity. We demonstrated how MIDAS v1.0 can be employed in practice for performance evaluation of Rocket and BOOM for the SPECint2006 benchmark suite and the DaCapo benchmarks.

Chapter 4

RTL Debugging with FPGAs

This chapter presents an effective RTL debugging methodology using FPGAs. Section 4.1 motivates our methodology by showing how difficult and painstaking RTL debugging can be. Section 4.2 covers existing simulation-based RTL debugging methodologies. Section 4.3 explains the implementation details of DESSERT, our RTL debugging methodology with FPGAs. Section 4.4 shows the debugging results of BOOM-v2 with DESSERT. Section 4.5 summarizes this chapter.

4.1 Motivation: How Challenging Is RTL Debugging?

The increasing complexity of modern hardware design makes verification challenging and verification often dominates design costs. While formal verification approaches are increasing in capability and can be successfully employed for some blocks or some aspects of a design, and while unit-level tests can improve confidence in individual hardware blocks, dynamic verification using simulators or emulators is usually the only feasible strategy for system-level verification. As well as verifying directed and random test stimuli, it is also important to validate the system specifications and design by running application software on the design. In addition to the large effort to create a system-level testbench, each bug found requires considerable effort to diagnose and repair.

Debugging errors found at the system-level while running realistic workloads is a notoriously difficult task. Since software RTL simulation is not fast enough, FPGA emulation is a popular way to boot Linux and run real-world applications on top to catch bugs before tape-out. However, because FPGAs are lack of visibility, it is extremely difficult to debug the target design when we encounter unexpected errors from long simulations.

Figure 4.1, 4.2, 4.3, and 4.3 show such cases when we run real-world applications

```
Duplicating 8479488 bytes
Duplicating 16958976 bytes
Duplicating 33917952 bytes
Duplicating 67835904 bytes
Duplicating 135671808 bytes
Duplicating 22257664 bytes
Input data 293601280 bytes in length
Compressing Input Data, level 5
[15460.780000] Oops - illegal instruction [#1]
[15460.780000] CPU: 0 PID: 45 Comm: bzip2 Not tainted 4.6.2-00044-gdf91b31 #243
[15460.780000] task: ffffffff8fe0d80 ti: ffffffff800b8000 task.ti: ffffffff800b8000
[15460.780000] sepc: ffffffff82343c78 ra : ffffffff820f3c0c sp : ffffffff800b9dc8
[15460.780000]  gp : ffffffff82354080 tp : ffffffff8fe0d80 t0 : 0000000000006000
[15460.780000]  t1 : 000000002e7ddb00 t2 : 0000000000000100 s0 : ffffffff8237b000
[15460.780000]  s1 : ffffffff8fe0d80 a0 : 0000000000000000 a1 : ffffffff823454a8
[15460.780000]  a2 : ffffffff823454e8 a3 : ffffffff82345528 a4 : 0000000000178321
[15460.780000]  a5 : ffffffff82343c78 a6 : 0000000000000000 a7 : 0000000000000000
[15460.780000]  s2 : ffffffffc4653000 s3 : 0000002003a33000 s4 : 8000000000000005
[15460.780000]  s5 : 000000000007a10d s6 : 0000002034daf140 s7 : 0000000000000000
[15460.780000]  s8 : 00000000000392e4 s9 : 00000000000392e5 s10: 0000002034c30ba4
[15460.780000]  s11: 0000002034c30ba0 t3 : 000000000003a400 t4 : 0000000000000070
[15460.780000]  t5 : 0000000000000006 t6 : 0000002034c30bf4
[15460.780000] sstatus: 0000000000000100 sbadaddr: 0000002003a33000 scause: 0000000000000002
[15460.850000] ---[ end trace 3232348a66aec084 ]---
[15460.850000] Kernel panic - not syncing: Fatal exception in interrupt
[15460.850000] ---[ end Kernel panic - not syncing: Fatal exception in interrupt
```

Figure 4.1: Kernel panic from `402.bzip2.ref` on BOOM-v2

```
 6 . . . . . . . O . . O . . 6
 5 . . X . . . . . X . . . . 5     WHITE has captured 0 stones
 4 . . . + . . + . . X . . . 4     BLACK has captured 0 stones
 3 . . . X . . . O . X O . . 3
 2 . . . . . . . . . O X . . 2
 1 . . . . . . . . . . . . . 1
   A B C D E F G H J K L M N
   A B C D E F G H J K L M N
13 . . . . . . . . . . . . . 13
12 . . . . . . . . . . . . . 12
11 . X O . . . . . . O . . . 11
10 . X . O . O + . . + . . . 10
 9 . . . . . . . O . O . . . 9
 8 . . X . X . . . . . . . . 8
 7 . . . + . . + X . + . . . 7
 6 . . . . . . . . . . O . . 6
 5 . . X . . . . . X . . . . 5     WHITE has captured 0 stones
 4 . . . + . . + . . X . . . 4     BLACK has captured 0 stones
 3 . . . X . . . O . X O . . 3
 2 . . . . . . . . . O X . . 2
 1 . . . . . . . . . . . . . 1
   A B C D E F G H J K L M N

gnugo 3.3.14 (seed 0): You stepped on a bug.
Please save this game as an sgf file and mail it to gnugo@gnu.org
If you can, please also include the debug output above this message.
```

Figure 4.2: Bug found from `445.gobmk.ref` on BOOM-v2

```
h264ref                          sss_encoder_main.cfg
./h264ref -d foreman_ref_encoder_baseline.cfg
./h264ref -d foreman_ref_encoder_main.cfg
./h264ref -d sss_encoder_main.cfg

Setting Default Parameters...
Parsing Configfile foreman_ref_encoder_baseline.cfg.....................................
.........................................................

------------------------------ JM 9.3 (FREXt) --------------------------------
 Input YUV file                   : foreman_qcif.yuv
 Output H.264 bitstream           : foreman_qcif.264
 YUV Format                       : YUV 4:2:0
 Frames to be encoded I-P/B       : 120/0
 PicInterlace / MbInterlace       : 0/0
 Transform8x8Mode                 : 0
-------------------------------------------------------------------------------
  Frame  Bit/pic WP QP   SnrY    SnrU    SnrV    Time(ms) MET(ms) Frm/Fld   I D
-------------------------------------------------------------------------------
0000(NVB)     176
0000(IDR)   21952 0 28  37.383  41.246  42.833        0       0     FRM    99
[   26.950000] h264ref[39]: unhandled signal 11 code 0x30001 at 0xfffffffffffffc20 in h264ref
[10000+11b000]
Segmentation fault
Setting Default Parameters...
@
```

Figure 4.3: Segmentation fault from `464.h264ref.ref` on BOOM-v2

```
-nan nan relu
-nan nan nan 0.000000 conv
-nan nan bias
-nan nan relu
-nan -nan -nan nan conv
-nan -nan 13.540989 nan conv
-nan -nan bias
-nan nan relu
-nan -nan bias
-nan 0.000000000 relu
-nan nan maxpool
-nan nan nan nan conv
-nan nan bias
-nan nan relu
-nan nan nan nan conv
-nan -nan -nan nan conv
-nan nan bias
-nan nan relu
-nan -nan bias
-nan nan relu
-nan -nan nan nan conv
-nan -nan bias
-nan nan relu
Detected hay
Cycles for /root/images/squeezenet_mousetrap.img = 12873460113
Instructions for /root/images/squeezenet_mousetrap.img = 6354376918
```

Figure 4.4: Floating-point errors from SqueezeNet inference on BOOM-v2

on BOOM-v2 [33]. When we encounter these errors on the FPGA, what we want to try first might be replaying them in software RTL simulation. Assume we use VCS for the two-way out-of-order processor BOOM. If we have waveform dumps enabled, the simulation rate is only 200 Hz. Typically, Linux-boot takes 700 million cycles, and thus, the time for BOOM-v2 to boot Linux in VCS will be 41 days. Unfortunately, these errors happen way beyond Linux-boot. Therefore, we can easily see that software RTL simulation is impractical for this kind of RTL debugging.

To cope with these difficulties, this chapter presents DESSERT, an FPGA-accelerated methodology for effective simulation-based RTL verification and debugging with the following contributions:

- We build upon earlier work in FPGA-accelerated RTL simulation to accelerate RTL verification and debugging. We extend MIDAS v1.0 in Chapter 3, which automatically generates the FPGA-accelerated RTL simulator as synchronous dataflow [81] from any RTL design to ensure *deterministic execution* on the FPGA given the same initial target state. For state initialization as well as state snapshotting, we also automatically insert scan chains with FIRTL compiler passes. The target memory space in the off-chip DRAM is also initialized through an automatically-instrumented loadmem unit.

- We implement FIRRTL custom compiler passes to *automatically synthesize assert and print statements* existing in RTL for *error checking from the FPGA*. Assertion synthesis provides quick error checking on the FPGA with a negligible simulation performance penalty. Print-statement synthesis, on the other hand, provides more exhaustive error checking by generating commit logs from the FPGA, which are compared on the fly against a functional golden-model software simulator.

- Since our FPGA-accelerated RTL simulators are deterministic, we run two identical FPGA simulation instances in parallel, spaced apart in simulation time, to allow errors detected by the lead instance to be replayed from an RTL snapshot captured by the trailing instance, significantly reducing state snapshotting overhead compared to periodic checkpoints. With this technique, DESSERT can provide fully-visible error traces of a buggy design without needing to rerun the simulator and without sacrificing simulation performance.

- We demonstrate a fast and easy-to-use methodology for system-level debugging. We demonstrate our methodology by simulating an open-source RISC-V in-order processor, Rocket [8], and an open-source RISC-V out-of-order processor, BOOM [33], to catch and fix bugs that occur hundreds of billions cycles into the SPECint2006 benchmark suite running under Linux. While in this chapter we study RISC-V processors and pipe a generated commit log to a reference ISA simulator, our approach can be generalized to other RTL designs for which a

golden model exists. In lieu of a golden model, inspecting synthesized assertions already present in the RTL is often a sufficient means to detect a simulation error.

4.2 Existing RTL Debugging Methodologies

Existing approaches for simulation-based system-level verification and debugging fall into a few categories as shown in Table 4.1.

Software RTL simulation with assertion detection can be an effective methodology for RTL verification and debugging, by producing waveform dumps that give full visibility into bugs. However, software RTL simulation is far too slow (up to tens of KHz) to run realistic workloads on complex hardware designs and becomes even slower when waveform dumps are enabled.

Hardware emulation engines, such as Cadence Palladium and Mentor Veloce, provide a software-like debug environment while being fast (around 1 MHz). But these custom emulation engines are extremely expensive, and can only be justified by the largest projects. Even in these projects, they remain a scarce resource that must be shared across multiple teams.

FPGA prototyping is a mainstay of pre-silicon full-system validation, as it is significantly cheaper than commercial hardware emulation engines and can be faster: single-FPGA prototypes can execute at tens to hundreds of MHz. (However, multi-FPGA prototypes are significantly slower or require expensive proprietary platforms like Synopsys HAPS). However, FPGA prototypes provide limited visibility for signal activities, making it extremely difficult to debug any errors encountered. Moreover, many bugs are sometimes difficult to reproduce, as they may depend on the non-deterministic initial state and latencies in the host-platform, such as DRAM or network I/O. While vendors provide FPGA signal monitoring tools, such as ChipScope [150] and SignalTap [58], these require manual selection of a few signals, leading to long debug loops as the design must be re-instrumented, re-synthesized, and re-executed to change the observed signals. There has been significant research towards improving controllability and visibility in FPGA prototypes by providing GDB-like interfaces [27, 63, 11] that allow emulations to be carefully advanced, halted, and resumed, selected internal signals to be *read* and *forced*, *breakpoints* to be set at runtime, and emulation to be *rewound*. Unfortunately, like the vendor-provided tools, effective debugging is predicated on selecting the right subset of signals to be instrumented for *reads*, *forces*, and *breakpoints*.

Checkpointed FPGA prototyping removes the need to intelligently select signals to instrument [95, 151, 37, 12, 75, 115] by allowing error waveforms to be reconstructed in software RTL simulation. While this provides full visibility of the design in a region of interest (ROI), checkpoint intervals must be carefully chosen as frequent checkpointing of large designs can easily become a simulation bottleneck,

RTL Verification Approach	Speed	Easy to Use	Deterministic	Controllability	Visibility	Cost
Software simulation	Very Slow	✓	✓	High	Full	Low
Hardware emulation engine	Fast	✓	✓	High	Full	Very High
FPGA prototype	Very Fast	✓	✗	Low	Limited	Low
Instrumented FPGA prototype	Fast	✗	✗	Moderate	Limited	High
Checkpointed FPGA prototype	Moderate	✗	✗	Low	Full	Moderate
DESSERT	Very Fast	✓	✓	High	Full in ROI	Low

Table 4.1: Comparison of contemporary simulation techniques for execution-driven RTL verification

while taking fewer snapshots lengthens the required I/O trace and the time it takes to replay the error in software simulation.

RTL verification against a golden model at a high-level description is another popular technique for functional verification. Brier et al. [25] verify DSP modules by comparing all the intermediate results between C/C++ models and software RTL simulation. Lee et al. [82] presents a technique for low-overhead state consistency checking with hash signatures. Iskander et al. [63] run FPGA RTL emulation for a SHA-1 core and compare the results between FPGA emulation and its high-level model.

There is also a commercial product that provides a verification solution for RISC-V processors running Spike, a *golden model* for RISC-V ISA, in tandem with software RTL simulation or FPGA emulation [22]. However, FPGA emulation does not guarantee deterministic execution, which makes functional validation much more difficult. In this chapter, commit logs are obtained from deterministic RTL simulation on the FPGA and compared against Spike.

Chatterjee et al. [35] also discusses how to prevent the divergence between the software model and the hardware emulation engine for instruction-by-instruction checking, but we discuss more cases that arise from real-world workloads instead of synthetic instruction streams. Also, we focus more on exact error checking than log compression as bulk data transfer is supported by host platforms through inter-FPGA-CPU DMA.

4.3 DESSERT: Debugging RTL Effectively with State Snapshotting for Error Replays across Trillions of Cycles

This section describes implementation details on the DESSERT framework. Figure 4.5 shows the tool flow of the DESSERT framework, which is extended from Figure 3.1 for effective RTL debugging on the FPGA.

Section 4.3.1 explains how we can achieve deterministic simulation on the FPGA. Section 4.3.2 explains how errors are detected on the FPGA. Section 4.3.3 describes how scan chains are automatically instrumented for state snapshotting and initialization. Section 4.3.4 describes compiler optimizations to reduce the FPGA resource overhead from instrumentation. Section 4.3.5 describes how to synchronize state between the golden model and the FPGA, preventing execution divergence. Section 4.3.6 shows how two identical simulation instances are run in parallel to efficiently detect and replay errors from the FPGAs.

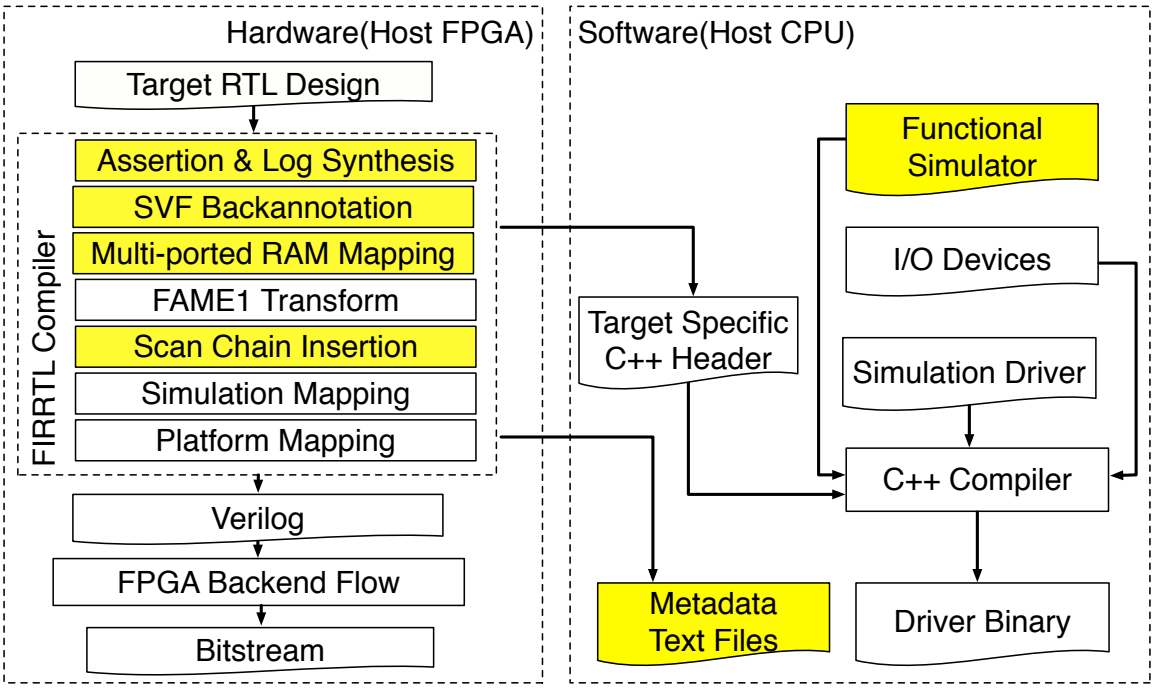


Figure 4.5: Tool flow to generate FPGA-accelerated RTL simulators

4.3.1 Deterministic RTL Simulation on the FPGA

Section 3.2.2 describes compiler transforms that enables accurate RTL performance modeling on the FPGA. The DESSERT framework takes advantage of these transforms to ensure *deterministic RTL simulation on the FPGA* for RTL debugging, which is crucial for pre-error state snapshotting using two identical simulation instances (Section 4.3.6). The *FAME1 Transform* and *Simulation Mapping* in Figure 4.5 generates a token-based simulator, an instance of synchronous dataflow (SDF) [81], which ensures *deterministic execution on the FPGA with the same initial state*.

4.3.2 Error Checking on the FPGA

4.3.2.1 Simulation APIs in Chisel

Target designs in this thesis such as RocketChip [8] and BOOM [34] are written in Chisel [10]. Chisel, like Verilog or VHDL, provides non-synthesizable print and assert constructs for software RTL simulation. Figure 4.6 demonstrates their use. The module contains a counter that increments until 10 when enabled (line 8). In this example, we expect the counter will never increment past 10: we check this with an `assert` on line 14. A `printf` in line 18-20 lets the engineer inspect the counter value without looking at the waveform. RocketChip and BOOM use assertions extensively

```
class Count extends Module {
  val io = IO(new Bundle {
    val en = Input(Bool())
    val done = Output(Bool())
    val cntr = Output(UInt(4.W))
  })
  // count until 10 when ‘io.en’ is high
  val (cntr, done) = Counter(io.en, 10)
  io.cntr := cntr
  io.done := done

  // assertion for software simulation
  // ‘cntr’ should be less than 10
  assert(cntr < 10.U)

  // printing for software simulation
  // show the counter value when ‘io.en‘ is high
  when(io.en) {
    printf("count: %d\n", cntr)
  }
}
```

Figure 4.6: Non-synthesizable simulation constructs in Chisel

to check their designs. In addition to `asserts`, traces of important activities, like commit logs, are generated with `printf`. `assert` and `printf` in Chisel are represented as `stop` and `print` in FIRRTL, respectively, to be synthesized by *Assertion and Log Synthesis* (Figure 4.5) as described in Section 4.3.2.2.

4.3.2.2 Assertion and Log Synthesis

DESSERT supports two ways to detect RTL bugs: quick hardware-based assertion checking and more exhaustive software-based checking that compares logs against a software golden-model functional simulator. Rather than manual instru-

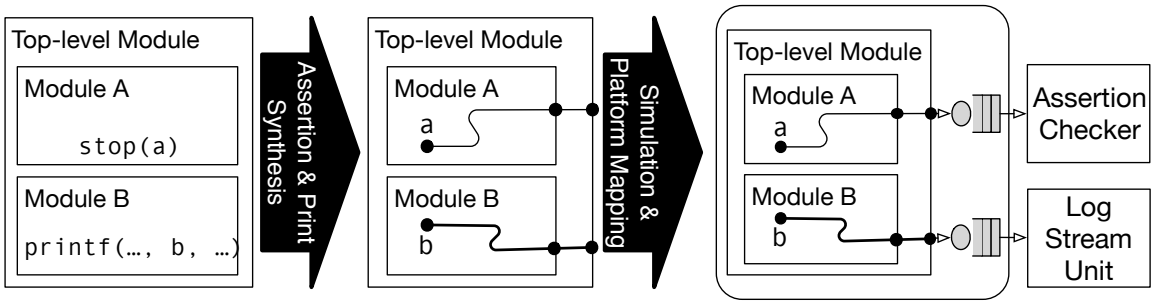


Figure 4.7: `stop` and `printf` synthesis for error checking on the FPGA

mentation, DESSERT automatically transforms assertions and logs that are already present in the source code for software RTL simulation (*Assertion and Log Synthesis* in Figure 4.5).

In FIRRTL there are two constructs to support assertions and logs: `stop` and `printf` [91]. `stop` is used to halt the simulation for a certain condition, while `printf` is used to print a formatted string when its condition is met. In general, assertions in HDL (e.g. `assert` in Chisel) are expressed as `stop` [1] with their error messages printed out by `printf`. Also, logs in HDL (e.g. `printf` in Chisel) are expressed as formatted messages in terms of RTL signal values with `printf`.

By default, `stop` and `printf` are emitted as non-synthesizable functions in System Verilog (e.g. `$fatal` and `$fwrite`). However, Figure 4.7 depicts how to automatically transform `stop` and `printf` into synthesizable logic for error checking on the FPGA. Note that their conditions and arguments are logic expressions of RTL signals. Thus, *Assertion and Log Synthesis* (Figure 4.5) inserts the combinational logic and the signals for the conditions and the arguments of `stop` and `printf`. This pass also creates output ports and connects the signals inserted for assertions and logs to these ports so that RTL errors are detected at the boundary of the top-level module. In addition, this compiler pass emits encodings of the assertions and logs that are synthesized (e.g. the error message for each `assert` and the print format for each `printf`) into text files that are used by the software simulation driver running on the host CPU.

4.3.2.3 Handling Assertions and Logs from FPGAs

After assertions and logs are synthesized, their top-level output ports are treated in the same way as the other top-level I/Os of the target design by *Simulation Mapping* in Figure 4.5. As a result, these output ports also generate their own timing tokens, which contain the cycle-by-cycle values of the output ports, every simulation cycle (Figure 4.7).

The timing tokens generated by assertions and logs are crucial for cycle-exact error checking from FPGAs, which will deterministically occur at the same target cycle both in software simulation and on the FPGA. Figure 4.7 also shows how these timing tokens are handled by instrumented hardware units in the FPGA, which are automatically inserted by *Platform Mapping* in Figure 4.5.

The *assertion checker* consumes timing tokens generated by assertions and inspects their values, which has no effect on simulation progress with no assertion failures. The assertion checker detects an error at cycle t if the value of the timing token at cycle t is non-zero, which means at least one assertion has fired. In this case,

[1] We assume the conditions of assertions are propositional. Temporal assertions found in SVA, which can be expressed as `stop` statements along with state machines, will be supported in the future.

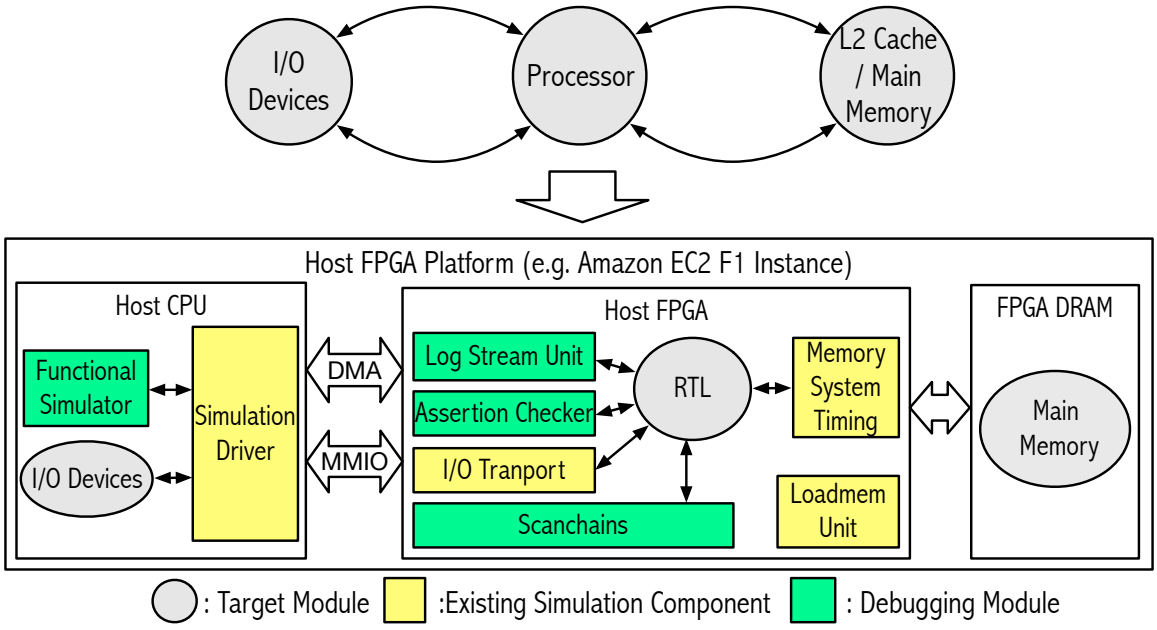


Figure 4.8: Mapping simulation to the host FPGA platform for RTL debugging

the checker records the target cycle t and the assertion id inferred from the timing token's value, and then stops accepting new tokens, which will halt simulation.

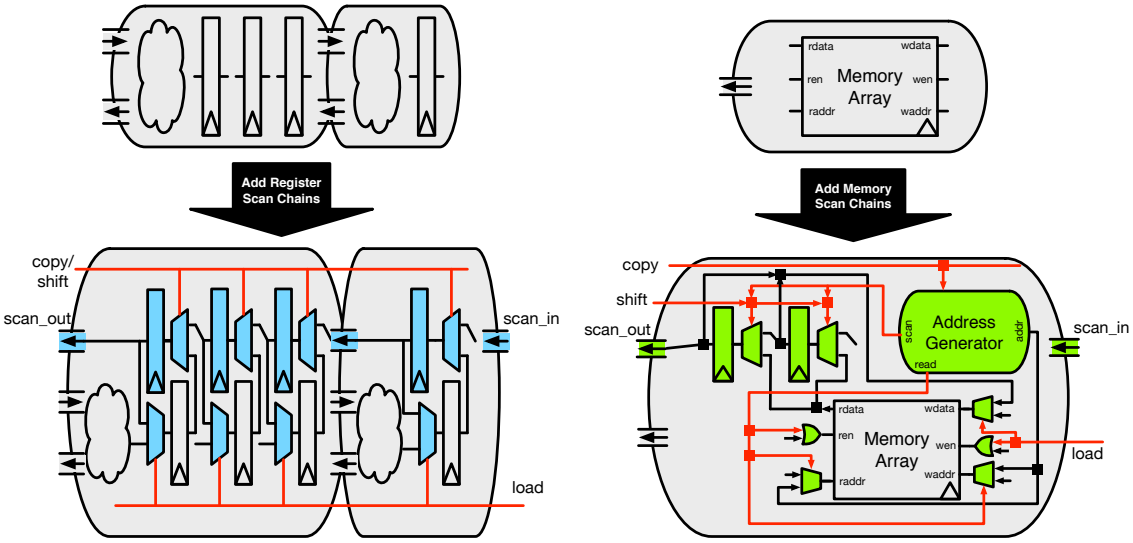
In parallel, the software simulation driver infrequently polls the assertion checker through memory-mapped I/O (Figure 4.8), and thus cycle-exact assertion detection can be achieved with negligible loss of simulation speed. When an assertion is detected from the FPGA, the simulation driver reads the target cycle and the assertion id from the checker and reports the assertion message along with its target cycle.

While the assertion checker simply drops timing tokens after inspecting them, in a log, these tokens along with their timestamps must be stored. Suppose a processor simulates at a clock rate of 50 MHz with an IPC of 0.5. If we print 64 bytes per committed instruction, this simulation would produce a commit log at 1.6 GiB/s. To manage this bandwidth, the *log stream unit* relies on inter-FPGA-CPU DMA to transfer the generated log en masse (Figure 4.8). Between DMA events, the log is buffered in a large BRAM FIFO[2]. When the buffer is full, the log stream unit stops consuming timing tokens to pause simulation until the buffer is drained, which prevents loss of log entries[3].

Once log entries are transferred from the FPGA to the buffers in the software simulation driver through DMA, they can be output on a console, piped to a file or consumed by a software golden model for exhaustive error checking.

[2]We plan to host this buffer in DRAM in the future.

[3] This may slow down simulation speed.



(a) Scan chains for registers (b) Scan chains for memory arrays

Figure 4.9: Automatic scan chain insertion

4.3.3 State Snapshotting and Initialization

4.3.3.1 Automatic Scan Chain Insertion

For state snapshotting and initialization, DESSERT implements automatic scan chain insertion (Figure 4.9) with a compiler pass in the FIRRTL compiler (*Scan Chain Insertion* in Figure 4.5). For registers, this compiler pass inserts platform-independent design-level shadow scan chains (Figure 4.9a), while for RAMs and large register files, it inserts specialized scan chains that automatically generate addresses to read out the whole data of large memory arrays without destroying their structures (Figure 4.9b). Imagine how tedious it would be if we implement these scan chains only with low-level IR nodes. To reduce this implementation overhead, we instead take advantage of the compilers-in-a-pass technique (Section 2.2.1) to generate scan chain IRs from parameterized scan chain modules written in Chisel.

Register Scan Chains. Figure 4.9a shows the instrumented logic for register scan chains. First, we insert multi-bit shift registers[4] that contain the values copied from or loaded to all registers in the target design. The scan chain is connected to the `scan_in` port and the `scan_out` port for its input and output, respectively. We also insert three control signals for these scan chains: `copy`, `shift`, and `load`.

For state snapshotting, we stall the simulation and assert the `copy` signal to copy the register values to the scan chain. Next, we pull out these values in the scan chain through the `scan_out` port one by one by asserting the `shift` signal.

[4]The width of shift registers is in general equal to the width of the data bus between the CPU and the FPGA.

For state initialization, we provide the values to be loaded through the `scan_in` port one by one by asserting the `shift` signal. When all necessary values are fed into the scan chain, we assert the `load` signal to load the values from the scan chain to the registers in the target design.

We insert these platform-independent design-level shadow scan chains to minimize the snapshotting latency and improve the portability[5]. However, the DESSERT framework can be extended to use more resource-efficient scan chains [75] or platform-dependent readback [144] to save FPGA resources that may significantly increase the snapshotting delay and sacrifice the portability.

Memory Scan Chains. We insert specialized scan chains to capture the state of RAMs and large register files due to their large volume and their limited numbers of read ports. Figure 4.9b shows the instrumented logic for memory scan chains. In addition to shift registers, ports, and control signals for each scan chain, we also add additional logic, the address generator, which not only generates addresses for the memory array but also asserts additional control signals (`read` and `scan`) for the memory array and the scan chain. All control signals are properly connected to the enable ports of the memory array as well as the muxes that are inserted to select the read/write addresses and the write data.

For state snapshotting, we stall the simulation and assert the `copy` signal to copy the data at address 0 to the scan chain. The address generator generates 0 for the read address and asserts the `read` signal simultaneously to read out the corresponding data. In the next cycle, the data at address 0 is available and the address generator asserts the `scan` signal to copy this data into the scan chain. We eventually read this data from the `scan_out` port by asserting the `shift` signal. To read the next data, we assert the `copy` signal again and the address generator generates the next address. We repeat this process until all data from the memory array is read out.

For state initialization, we first assert the `copy` signal to generate address 0 from the address generator. Next, the data at address 0 is fed into the `scan_in` port by asserting the `shift` signal. Finally, we assert the `load` signal to write the data from the scan chain to the memory array. We repeat this process until all necessary data is loaded into the memory array.

4.3.3.2 I/O Traces

We also need I/O traces for error replays in software simulation. Specifically, if an RTL snapshot is to be replayed for L cycles, the inputs and the outputs for L cycles must be recorded by communication channels (Figure 3.2), which is added by *Simulation Mapping* in Figure 4.5, after the RTL snapshot is taken. When the RTL snapshot is loaded in software simulation, the input traces are fed to the inputs of the target design to drive the replay, while the output traces are compared cycle by cycle against the outputs of the target design to check the correctness of the replay.

[5] The trade-offs for various checkpoint implementations are discussed by Koch et al. [75]

4.3.3.3 Off-chip Memory Initialization

As the target design's DRAM is mapped to the off-chip DRAM, it needs to be initialized for deterministic simulation. The loadmem unit (Figure 4.8), which is automatically added by *Platform Mapping* (Figure 4.5), not only loads the program to execute but also initializes the remaining target main memory space.

4.3.4 Optimizations to Reduce FPGA Resource Overhead

4.3.4.1 SVF Backannotation

The FIRRTL compiler applies word-level optimizations such as constant propagation, common-subexpression elimination, and dead-code elimination. However, these optimizations are not enough to eliminate redundant logic in the target design. In fact, during logic synthesis, CAD tools prune out more unnecessary logic with more aggressive bit-level optimizations. The problem is that this dead logic is alive if we insert scan chains without deleting it in the FIRRTL pass. This is because dead registers are connected to scan chains, and thus, they cannot be eliminated during FPGA synthesis, which is detrimental for the FPGA resource utilization.

Instead of implementing these bit-level optimizations in FIRRTL, we backannotate the dead registers from existing CAD tools to eliminate them. If we delete these registers from the backannotation, other related combinational logic is also eliminated by FIRRTL optimizations or CAD tools.

In this section, we use Synopsys Design Compiler for the backannotation. Design Compiler dumps its optimization information to the SVF file that is in turn consumed by Synopsys Formality for equivalence check. We use the text file that is generated when the SVF file is processed by Synopsys Formality. This text file contains the following information:

- **Renamed Signals:** Optimizations may change signal names. Specifically, each bit in a signal word may have its own name by bit-level optimizations.
- **Uniquified Instances:** A module can have multiple instances in the entire design. However, theses instances may be uniquified as they can diverge after optimizations.
- **Constant Registers:** Some registers become constant zero or one by optimizations.
- **Merged Registers:** Two identical registers are merged into a single register, which removes duplicate registers.

SVF Backannotation in Figure 4.5 uses the above information to conduct aggressive bit-level optimizations, which in turn greatly reduces the FPGA resource

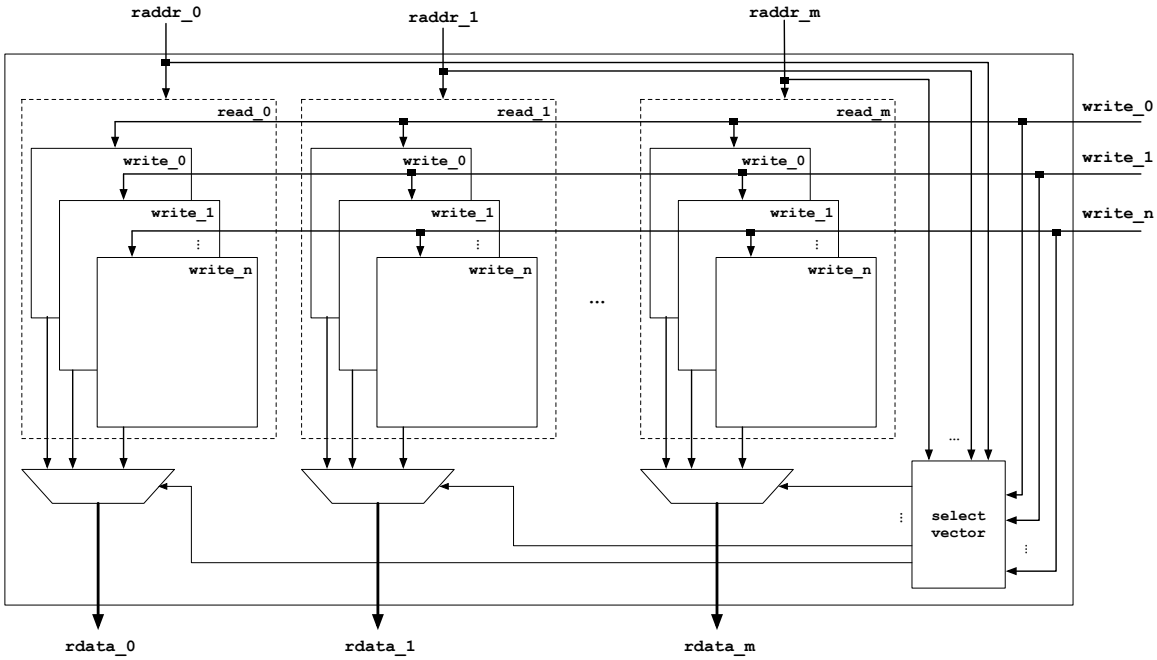


Figure 4.10: Resource efficient mapping of multi-ported RAMs on FPGAs

overhead. This pass first uniquifies instances and rename each bit of optimized registers. Next, it replaces constant registers with their constant values. Finally, this pass merges registers to eliminate duplicate registers.

4.3.4.2 Multi-ported RAM Mapping

Large multi-ported RAMs are building blocks for physical registers in high-performance out-of-order processors. These RAMs can be efficiently implemented with SRAMs or latches in the silicon, but are inefficiently mapped to flip-flops consuming lots of LUTs on the FPGA without cares. Dwiel et al. [44] present efficient mappings of multi-ported RAMs with replication and time multiplexing. Inspired by this idea, in this section, we automatically transform multi-ported RAMs into resource efficient register files, each of which is implemented with duplicate distributed RAMs and the select vector.

Figure 4.10 shows a resource efficient mapping of multi-port RAMs on FPGAs. For a multi-port RAM with m read ports, n write ports, and k elements, we need $m \times n$ copies of k-element memory arrays with the $k \times \lceil \log_2 n \rceil$ select vector to emulate this multi-ported RAM. When a value is written at address α through the ith write port, this value is written to the ith memory array of each read port (m copies in total). In addition, the αth element of the select vector is updated to i, the id of the memory array that contains the up-to-date value at address α. When the value at address β is read through read port j, all n memory arrays assigned to this read

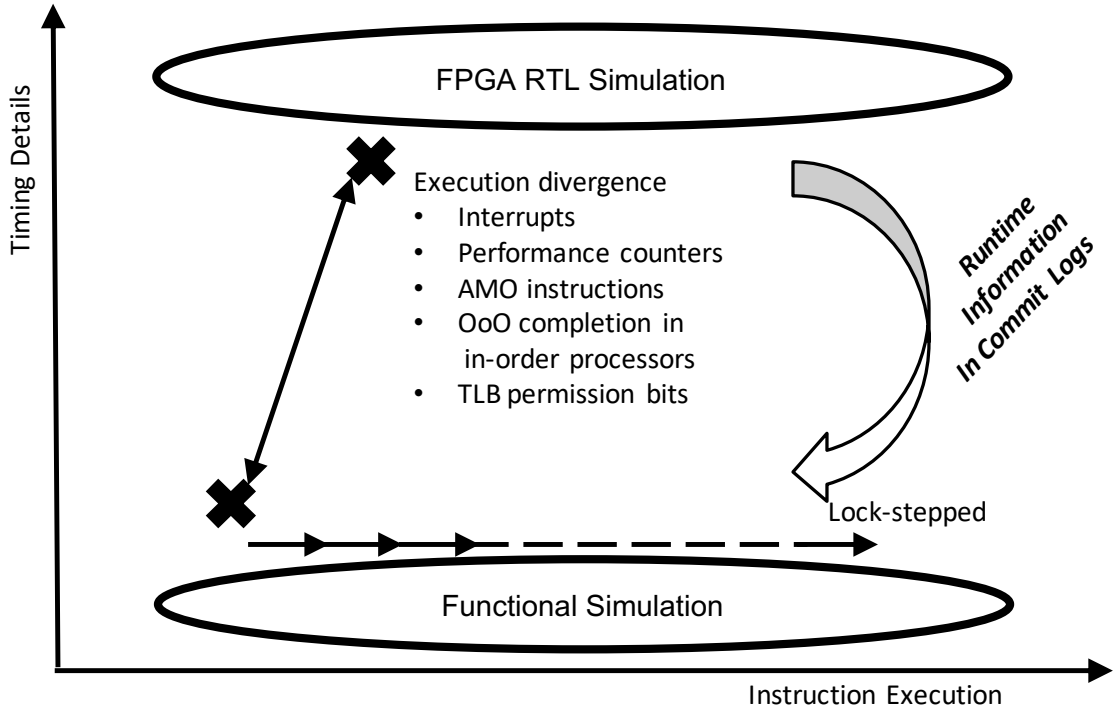


Figure 4.11: State synchronization between the function simulator and the FPGA

port are accessed. To select the up-to-date value among them, the βth element of the select vector is also read and provided to the jth mux.

Each memory array has one read port and one write port, and thus, these memory arrays are efficiently mapped to distributed RAMs. This custom transform is also implemented using the compilers-in-a-pass technique with a parameterized Chisel module (Section 2.2.1).

4.3.5 State Synchronization between the Golden Model and the FPGA

DESSERT is a general methodology that can be applied to any hardware designs. As such, for software-based error checking, logs generated from FPGAs are compared against a software golden model of any RTL. However, if we use DESSERT for microprocessor verification, the state of the software functional simulator must be carefully maintained to prevent divergence from the RTL implementation. Figure 4.11 depicts a high-level idea of state synchronization between the golden model and the RTL implementation.

First, the physical memory and device configurations of the functional software simulator and the RTL implementation should be identical. This ensures the memory

zones of Linux are the same in both implementations, resulting in the same page allocation.

Next, interrupts in both implementations must be synchronized. It is incredibly difficult to make interrupts happen simultaneously in both implementations since the functional simulator has no timing model. Instead, interrupts in the functional simulator are disabled by default. Whenever an interrupt is raised from the RTL implementation, the interrupt cause is passed along with the commit log from the FPGA to the functional simulator. Then, the functional simulator is forced to handle the interrupt on the same instruction as the RTL.

In addition, microarchitecture-dependent state needs to be synchronized. Examples include performance-counter reads, atomic memory operations, and memory-mapped I/Os. Performance-counter reads and atomic memory operations are easily identified by their instruction encoding while memory-mapped I/Os are identified by their memory addresses. Whenever such events happen, the destination register values of the functional simulator are updated with those in the FPGA's commit log.

Some processors support out-of-order completions for long-latency instructions using a scoreboard to maintain register dependencies (e.g. the Rocket processor [8]). In this case, the destination register values may not be available even though instructions have retired. We cannot ignore these instructions due to microarchitecture-dependent state. Therefore, the commit log also includes the information of whether or not the scoreboard is set by each instruction. When the scoreboard is set, the destination register value is not compared immediately. Instead, the functional simulator saves the destination register value with its address. When the instruction completes in the FPGA, its destination register value as well as the register address are delivered from the FPGA to the functional simulator and compared. For microarchitecture-dependent state, the destination register value of the functional simulator is updated with the value from the FPGA.

Finally, the permission bits in TLBs are modeled in the functional simulator. This is because TLB flushes can be delayed by an OS as a performance optimization, resulting in accesses to stale page-table entries. Thus, whenever the TLBs in the FPGA are refilled, the functional simulator updates its TLB model by using the TLB tag and the permission bits of the page-table entry from the FPGA. Memory accesses in the functional simulator also go through the TLB model to match page faults between the function simulator and the FPGA.

Other forms of complex golden functional model, such as out-of-order memory systems, will require similar strategies to track cycle-level interleaving of the RTL design.

4.3.6 Ganged-Simulation for Rapid Error Replays

Redundant multi-threading is a popular technique for fault-tolerant computing and post-silicon validation (e.g. [56]) where two identical threads are required for error

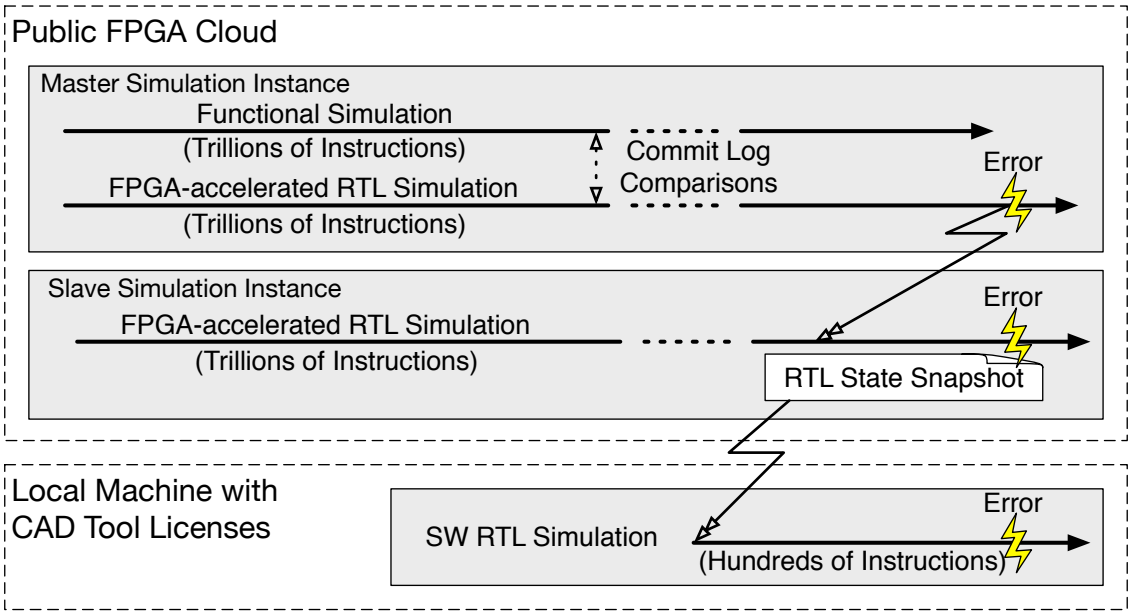


Figure 4.12: Ganged-simulation for rapid error relays

checking. In our case, only a single simulation run is necessary for error checking. We, instead, need another simulation instance that always runs behind for *pre-error state snapshotting*.

To detect and replay errors efficiently, we exploit the determinism of our FPGA-accelerated simulation by running two identical simulators concurrently: a leading *master* instance, which detects the target RTL bugs, and a lagging *slave*, which checkpoints the target RTL state (Figure 4.12).

The leading master checks for simulation errors by detecting either an assertion failure or a mismatch between the golden model and the simulator-generated log (Section 4.3.2). The master controls the advance of the slave by periodically sending it packets over TCP, each of which contains a target cycle timestamp and an error detection bit, indicating whether or not the master has encountered an error at the timestamped target cycle[6].

The slave cannot proceed until it receives a timestamped message from the master. When it receives a message with a clear error bit, it can safely advance up to the timestamped target cycle of the message. On the other hand, when the slave receives the message with a set error bit, it advances up to the timestamped target cycle minus L cycles to capture an L-cycle snapshot of the ROI (Section 4.3.3). Since simulations are deterministic (Section 4.3.1), the same error, which is detected by the master, also is captured by the slave at the same target cycle.

[6] Alternatively, we may use `f1.4xlarge` instances with 2 FPGAs, which are recently provided by Amazon.

Finally, the captured RTL state snapshot can be replayed L cycles in software RTL simulation until the same error appears, thus providing a fully-visible error waveform of the target over the ROI. This waveform dramatically improves debuggability, helping RTL designers find and fix the cause of the bug.

To mitigate the monetary costs, we use FPGAs in the cloud. This provides a cheap, elastic source of very large FPGAs, without the large initial capital expense. On the other hand, commercial CAD tools are not allowed to run in the public cloud, and thus, error snapshots are copied to and replayed in the local machine with the CAD tool licenses.

4.4 Results

We demonstrate the effectiveness of our methodology with a case study of two RISC-V processor core designs and report on the types of bugs found.

4.4.1 Target Designs, Golden Model, Benchmarks, and Host Platform

Target Designs: We apply DESSERT to two open-source RISC-V processors implemented with Chisel [10]: Rocket [8], a productized scalar in-order processor, and BOOM-v2 [34], an industry-competitive, open-source out-of-order processor. Table 4.2 shows the processor configurations used for this study with the number of assertions and the size of log entries. Log entries are generated when instructions are committed. The processor and L1 cache represent the design under test (DUT) and are supplied as RTL, while the supporting L2 cache and DRAM are implemented as abstract timing models, which can be configured at runtime.

Software Golden Model: We employ Spike [137] as a golden model for the RISC-V ISA, which is modified for commit log comparison (Section 4.3.5). For software-based checking, commit logs generated by Rocket or BOOM-v2 from the FPGA are compared against Spike.

Benchmarks: We execute the SPECint2006 benchmark suite on the target processors hosted on the FPGA. All benchmarks are compiled using gcc version 6.1.0, and run on Linux kernel version 4.6.2. For each benchmark, we built a BusyBox image including all necessary files for a given benchmark within an initramfs.

Host Platform: We use Amazon F1 instances (`f1.2xlarge`) as simulation host platforms. An `f1.2xlarge` instance is equipped with Xilinx UltraScale+ VU9P and 1.5GB/s FPGA-CPU DMA.

Parameter	Rocket	BOOM-v2
Fetch-width	1	2
Issue-width	1	4
Issue slots	-	60
ROB size	-	80
Ld/St entries	-	16/16
Physical registers	32(int)/32(fp)	100(int)/64(fp)
Branch predictor	-	gshare: 16 KiB history
BTB entries	40	256
RAS entries	2	4
MSHR entries	2	2
L1 $ capacities	16 KiB or 32 KiB	
ITLB and DTLB reaches	128 KiB / 128 KiB	
L2 $ capacity and latency	1 MiB / 23 cycles	
DRAM capacity and latency	2 GiB / 80 cycles	
Assertions	123	601
Commit log entry width	60 B	64 B

Table 4.2: Target processors verified with DESSERT

4.4.2 FPGA Quality of Results

We compiled bitstreams using Vivado 2017.1 targeting the Xilinx UltraScale+ VU9P parts present in Amazon EC2 F1 instances. Pure FPGA mappings for both designs close timing at 62.5 MHz, which is bounded by the unretimed double-precision FMA in both cores[7]. The compile time is about 2 hours for Rocket and 4 hours for BOOM on `c4.8xlarge` (about $1 and $2 with spot instances, respectively).

Table 4.3 shows the total utilization of the VU9P after place and route, with varying levels of instrumentation enabled:

- *Prototype*: just the processor without transformations
- *FAME1*: FAME1 simulator for deterministic simulation (Section 4.3.1)
- *Debug*: FAME1 simulator with assertion and print synthesis (Section 4.3.2)
- *Scan*: FAME1 simulator with scan chain insertion (Section 4.3.3)
- *All*: FAME1 simulator with all transforms and instrumentation

[7]Vivado cannot retime the 3-cycle double-precision FMA present in both cores. Manually pipelining the unit increases f_{max} to 190 MHz.

Processor	Resource	Prototype	FAME1	Debug	Scan	All
Rocket	Logic LUTs	18.0%	18.4%	18.5%	24.6%	24.7%
	Registers	10.8%	10.8%	10.9%	13.6%	13.7%
	BRAMs	18.1%	19.6%	24.9%	21.2%	26.6%
BOOM-v2	Logic LUTs	28.0%	28.4%	30.7%	51.5%	52.1%
	Registers	12.9%	12.8%	13.4%	22.4%	22.5%
	BRAMs	19.4%	20.9%	27.4%	22.6%	30.1%

Table 4.3: FPGA utilization versus instrumentation level

Processor	Verilator	VCS	
Rocket	6.9 kHz	6.1 kHz	
BOOM-v2	1.8 kHz	0.2 kHz	
Processor	**FPGA No-Checking**	**FPGA Assertion**	**FPGA Log**
Rocket	52.7 MHz	52.6 MHz	21.3 MHz
BOOM-v2	52.3 MHz	52.1 MHz	13.7 MHz

Table 4.4: Simulation rates for various simulators

LUTRAMs, DSP48s, and URAMs are omitted as they are lightly used (<1%, <5% and 0%).

The FAME1 transform has marginal overhead over the prototype due to FPGA tool optimizations. The debug instrumentation uses slightly more LUTs for assertion synthesis and more BRAMs for log buffers. As expected, the scan chain instrumentation has large overhead on both LUTs and Registers. However, the DESSERT framework can be extended to adopt more resource-efficient checkpoint implementation as discussed by Koch et al [75].

4.4.3 Simulation Performance

Table 4.4 compares the simulation rates of software RTL simulators and a single instance of FPGA-accelerated simulation with no error checking (*FPGA No-Checking*), hardware-based checking from assertion synthesis (*FPGA Assertion*), and software-based checking comparing logs from the FPGA against a golden model (*FPGA Log*).

In software simulation, waveforms are recorded for debugging. In this evaluation, intermediate signals generated by Chisel and FIRRTL are not traced in Verilator, while all signals are traced in VCS. As we can see, software RTL simulation does not ensure sufficient simulation performance for debugging with real-world applications and is even slower with complex hardware designs.

Benchmark	Assertion Failure	Cycle (B)	Simulation Time (mins)
483.xalancbmk.test	Invalid writeback in ROB	1.9	3.4
464.h264ref.test	Pipeline hung	3.2	3.8
471.omnetpp.test	Pipeline hung	3.3	3.9
445.gobmk.test	Invalid writeback in ROB	14.9	9.0
471.omnetpp.ref	Pipeline hung	62.6	22.2
401.bzip2.ref	Wrong JAL target	473.7	164.6

Table 4.5: Assertion triggers from BOOM-v2 running the SPECint2006 benchmark suite.

On the other hand, FPGA-accelerated RTL simulation guarantees high simulation rates regardless of design complexities. In addition, hardware-based assertion checking has almost no performance overhead as the assertion checker is infrequently polled by the software driver (Section 4.3.2.3).

Software-based checking decreases simulation rates because, in this case study, the functional simulator must be run and compared in lock step (Section 4.3.5). As a result, the log buffer is not quickly drained, resulting in frequent simulation stalls. Notably, software-based checking has a larger performance impact on BOOM-v2, which has greater IPCs, and thus, generates more commit log entries per cycle. However, exhaustive software-based checking is still worthwhile as it can discover subtle bugs not found by hardware-based assertion checking (Section 4.4.5). We believe the simulation performance can be further improved with decoupling and speculation of functional simulation, to reduce synchronization frequency.

4.4.4 BOOM-v2 Assertion Failure Bugs Found

BOOM-v2 is a major microarchitectural update of the original BOOM processor to improve its physical realizability [34]. BOOM-v2 passes all ISA tests, random instruction tests, microbenchmark tests, and boots Linux. However, we noticed that some of the SPECint2006 benchmarks that passed in BOOM-v1 failed in BOOM-v2. Therefore, we used DESSERT to debug BOOM-v2.

Table 4.5 shows assertions caught from BOOM-v2 when running the SPECint2006 benchmarks. Note that assertion messages were shown in FPGA-accelerated RTL simulation when these assertions were triggered. In addition, RTL state snapshots were taken before the assertions were triggered (Section 4.3.6) and replayed in software RTL simulation for full visibility of the internal signals.

With the waveform from the 1024-cycle error replay, we quickly tracked down the cause of the *invalid writeback in ROB* assertion to a buggy interaction between back-pressure queuing and branch misspeculation that did not correctly kill instruc-

tions moving data from the integer register file to the floating-point register file. In general, the *pipeline hung* assertion was caused by pipeline resource scarcities for various reasons, which were not found in the 1024-cycle window, suggesting assertions describing more specific properties be necessary. Also, the waveform from the 1024-cycle error replay revealed that the *wrong JAL target* assertion, which was triggered at almost a half trillion target cycles, was caused by incorrectly handled signed arithmetic in computing jump target addresses, which is latent until the processor touches instructions allocated in a high-address memory region.

We caught all these assertion triggers and obtained full visibility within 3 hours using two Amazon EC2 F1 instances. Therefore, the total cost to catch and replay these errors is roughly \$2 (compilation) $+2 \times$ \$1.56 (simulation) = \$5.12 with spot instances, which is extremely economical compared to commercial emulation tools.

4.4.5 BOOM-v2 Commit Log Bugs found

Software-based error checking, which compares logs from an FPGA against a software golden model, can discover subtle bugs that may not immediately affect the results of applications. We verified Linux boot in Rocket and BOOM against the software golden model using commit logs from the FPGA (Section 4.3.5). Linux boot in Rocket was successfully verified against the golden model[8]. However, Linux boot in BOOM-v2 failed with the following message:

```
Instruction mismatch at cycle: 669432906
    PRIV        PC               INST                 REG
Last: 0 0x0000000000069ce0 (0x00100793) x15 0x0000000000000001
SW  : 0 0x0000000000069ce4 (0x1404272f) x14 0x0000000000000000
FPGA: 1 0xffffffff80422a9c (0x14011173) x 2 0xffffffffcc54000
```

This shows BOOM jumped into Linux's exception handler (PC = `0xffffffff80422a9c`) while executing `lr.w a4, zero, (s0) (0x1404272f)`. The waveform from the 1024-cycle replay showed BOOM incorrectly triggered a store access fault for load-reserved instructions. After fixing this bug, Linux boot in BOOM-v2 fully matched against the golden model. This bug was found in less than three minutes including target memory initialization, but would have taken a month using VCS.

Commit log comparisons are also helpful to catch bugs that are not easily discovered by assertions. For example, `403.gcc.test` fails in BOOM without assertion triggers. However, from commit logs, the following mismatch is found:

```
Instruction mismatch at cycle: 2909587019
    PRIV        PC               INST                 REG
Last: 0 0x00000000001d15fc (0x14d76e63)
SW  : 0 0x00000000001d1600 (0x03079793) x15 0x0000000000000000
```

[8] We could not easily match floating-point loads due to what was a legally valid ambiguity due to microarchitectural implementation differences between Rocket Chip and the golden model (Spike). Newer versions of the RISC-V ISA close this specification ambiguity.

```
FPGA: 0 0x00000000001d1600 (0x01813483) x 9 0x00000000004322e8
```

Note that this bug is found at 2.9 billion cycles in just *6 minutes*; Verilator would have taken nearly *three weeks* to reach this bug.

The commit log shows BOOM fetching the wrong instruction at PC = `0x1d600`. The waveform from the 1024-cycle replay shows that BOOM's fetch buffer is unable to accept more instructions and applies back-pressure to the instruction cache, which experiences a cache miss at the same time. Once the cache miss is resolved, the wrong instruction is returned from the instruction cache. BOOM-v2 shares the frontend and the instruction cache with RocketChip, and we used an old version of RocketChip [9] on which the current version of BOOM-v2 [10] is based. The frontend and the instruction cache in the current version of RocketChip has since been completely rewritten. We will verify BOOM-v2 again with a newer RocketChip code base in the future.

4.5 Summary

In this chapter, we presented DESSERT, an effective RTL debugging methodology using FPGAs. Motivated by the fact that RTL debugging can be extremely challenging and painstaking, we developed a framework that helps rapidly catches and helps fix bugs that only manifest after hundreds of billions of target clock cycles, with little developer effort and at extremely low cost, by taking advantage of cloud-hosted FPGA platforms. With automatic transforms and instrumentation of the target RTL design, DESSERT ensures deterministic simulation on the FPGA; supports both quick error checking with assertion failures on the FPGA and more exhaustive error checking with commit log comparisons between the golden model and the FPGA; and captures RTL state snapshots for error replays in software simulation for full visibility. DESSERT also quickly provides the error trace from pre-error RTL state snapshots captured by two identical simulations that are run in parallel. We also demonstrated the usefulness of DESSERT with debugging of BOOM-v2 for errors encountered at up to half trillion cycles, when running the SPECint2006 benchmark suite.

[9] Commit Hash: 8c8d2af7141102adf8ccc65b929e740ce7ce189, Date: Feb 9th, 2017
[10] Commit Hash: 70b94eefe6658a1444ca420ab86953c25665dae8, Date: Sep 12th, 2017

Chapter 5

Sample-Based Energy Modeling

This chapter presents Strober, a sample-based energy modeling methodology for average power and energy estimation. Section 5.1 motivates why RTL-based power and energy evaluation is necessary for computer architecture research. Section 5.2 describes existing design-time power modeling methodologies. Section 5.3 overviews our sample-based energy evaluation methodology. Section 5.4 presents the Stober framework as an implementation of sample-based energy evaluation. Section 5.5 shows the evaluation results of Strober with Rocket and BOOM. Section 5.6 concludes this chapter.

5.1 Motivation: Why RTL-based Power/Energy Modeling?

Energy efficiency has become the primary design metric for both low-power portable computers and high-performance servers. As technology scaling slows down, computer architects must use architectural innovation rather than semiconductor process improvement to improve energy efficiency. This trend necessitates accurate and fast energy evaluation of various long-running applications on novel designs for architectural design-space exploration.

The most accurate way to evaluate energy efficiency is by running applications on a silicon prototype with power consumption measured directly. Prototyping is accurate and can run large workloads rapidly, but each prototyping cycle is expensive and has a long latency, prohibiting extensive design-space exploration.

Computer architects instead mostly rely on analytic power models calibrated against representative RTL designs [26, 134, 92, 90, 120]. These must be driven by activities from micro-architectural simulation [19, 109, 143]. This approach helps designers gain some intuition in early design phases, but is limited to microarchitectures resembling those for which the abstract model was built, and requires long simulation

times to gather microarchitectural activities. As power model validation depends on the existence of representative RTL, constructing abstract power models is more difficult for non-traditional architectures such as application-specific accelerators.

When complete RTL designs are available, they can be used to evaluate not only energy efficiency, but also cycle time and area using commercial CAD tools. Although existing commercial CAD tools provide extremely accurate performance and power estimates from detailed gate-level simulation, the simulation runtime of complex designs is painfully slow, preventing large architecture studies of many hardware configurations.

This chapter describes a sample-based RTL energy-modeling methodology, which enables fast and accurate energy evaluation of long-running applications. First, a design's performance is evaluated using full-system RTL simulation, during which a set of replayable RTL snapshots is captured randomly over the course of a program's execution. Next, the design's average power is estimated by replaying the samples on a gate-level power simulator, which also provides the confidence interval for the average power estimate.

This chapter also presents the open-source Strober framework, an example implementation of sample-based energy simulation. Strober is implemented with custom transforms in the FIRRTL compiler [64] to automatically generate an FPGA-accelerated FAME1 simulator from any RTL design for rapid performance modeling. The FAME1 simulator is enhanced with the ability to capture a full replayable RTL snapshot at any sample point, which can then be replayed on a commercial gate-level simulator to obtain power numbers. The Strober framework is evaluated using the in-order processor Rocket [8] and the out-of-order processor BOOM [34].

The main contributions of this chapter are as follows:

- *General and Easy-to-Use Framework:* Strober automatically generates FPGA-accelerated FAME1 simulations from any RTL design including the ability to snapshot simulation state for replay on gate-level simulation, thus minimizing designers' manual effort. We present results using RTL designs of in-order and out-of-order processors, but note that the approach applies to any RTL including application-specific accelerators.

- *Accurate Estimation:* Performance measurement is truly cycle-accurate, since it is based on the RTL design modeled using a token-based timing simulation. For average power, we can achieve less than 5% error with 99.9% confidence against commercial CAD tools. This indicates Strober can be a framework to provide ground truth for other models.

- *Fast Simulation:* We achieve more than two orders of magnitude speedup over existing microarchitectural simulators and four orders of magnitude speedup over commercial Verilog simulators. This implies Strober can support large

design-space exploration using long-running applications on complex hardware designs.

5.2 Existing Methodologies for Design-Time Power and Energy Evaluation

Analytical power modeling [26, 134, 92, 90, 120] combined with microarchitectural software simulators [19, 109, 143] is widely-used for computer architecture research. This method enables early architecture-level design-space exploration, helping designers gain high-level intuitions before RTL implementation. However, microarchitectural software simulators execute far more slowly than real systems, requiring application runs to be subset. Moreover, the power models should be strictly validated against real systems or detailed gate-level simulations, which is difficult when exploring new non-traditional designs. This chapter also suggests sample-based energy simulation as a way of obtaining accurate ground truth to train abstract power models rapidly.

There are significant efforts to validate analytic power models. Shafi et al. [119] validate an event-driven power model against the IBM PowerPC 405GP processor. Mesa-Martinez et al. [98] validate power and thermal models by measuring the temperature of real machines. The authors measure temperature using an infrared camera and translate temperature to power using a genetic algorithm. Xi et al. [149] validate McPAT against the IBM POWER7 processor and illustrate how inaccuracies can arise without careful tuning and validation. Lee et al. [83] propose a regression-based calibration of McPAT against existing processors to improve its prediction accuracy. McKeown et al. [96] characterize power and energy of an open-source 25-core processor from its silicon implementation. However, these methodologies can only be applied using existing machines or proprietary data. Jacobson et al. [65] suggest a power model from pre-defined microarchitectural events and validate it against RTL simulation. However, the approach relies on designer annotations and microbenchmarks exploiting familiarity with a particular family of processor architectures. In contrast, Strober can be used for validation of novel hardware designs and long-running real world applications.

There are a number of significant attempts to accelerate power estimation using an FPGA. Sunwoo et al. [127] generate power models from manually specified signals, which requires designers' intuition. This technique also requires additional manual efforts to instrument existing FPGA simulators with power models. Bhattacharjee et al. [17] also manually implement event counters in FPGA emulators to speed up event-driven power estimation. Coburn, Ravi, & Raghunathan [39] implement detailed power models directly on the FPGA, which suffers from large FPGA resource overhead. Ghodrat et al. [49] extend Coburn et al. by employing a software/FPGA co-emulation approach to reduce FPGA resource overhead, but introduces commu-

Population		Sample	
size	N	*size*	n
mean	$\overline{X}$	*mean*	$\bar{x}$
variance	σ^2	*variance*	s_x^2
		sampling mean	$\overline{X}$
		sampling variance	$Var(\bar{x})$
		confidence level	$(1-\alpha)$
		confidence interval	$\bar{x} \pm z_{1-(\alpha/2)}\sqrt{Var(\bar{x})}$

Table 5.1: Statistical parameters

nication overhead between the software and FPGA, which can bottleneck emulation performance without careful partitioning. Atienza et al. [9] implement a special module to monitor selected signal activities on FPGA.

Our Strober framework differs in that the hardware design is automatically instrumented to generate samples instead of manually implementing power models on an FPGA, while still minimizing FPGA resource overhead.

5.3 Methodology Overview

In this section, we present our sample-based energy simulation methodology using RTL designs for fast and accurate energy estimation. First, we present a brief theoretical background of statistical sampling in Section 5.3.1 with parameters in Table 5.1. Next, we describe how statistical sampling is applied to RTL energy simulation in Section 5.3.2.

5.3.1 Statistical Sampling

A population P of size N is the set of all elements (e_1, e_2, ... e_N) which could be selected in an experiment. Each element e_i has a corresponding measurable quantity, X_i. A population's parameters such as its mean, $\overline{X}$, and its variance, σ^2, can be exactly calculated if all elements within the population are measured:

$$\overline{X} = \frac{\sum_{i=1}^{N} X_i}{N} \tag{5.1}$$

$$\sigma^2 = \frac{\sum_{i=1}^{N} (X_i - \overline{X})}{N} \tag{5.2}$$

Unfortunately, evaluating every element in P is usually infeasible due to any number of resource constraints. Instead, a subset of the population, a sample, is

selected according to a sampling strategy, and is used to estimate some parameters of the original population.

While there are many sampling strategies, the most statistically robust strategy is random sampling without replacement, where every e_i in P has an equal probability of being selected in a sample. For simplicity and clarity, the following assumes this specific sampling strategy.

To estimate population parameters, every element in a sample of size n is measured (x_i), and the sample mean $\bar{x}$ and sample variance s_x^2 are calculated. These sample values are used to estimate the corresponding true population values.

$$\overline{X} \approx \bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \tag{5.3}$$

$$s_x^2 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1} \tag{5.4}$$

$$\sigma^2 \approx \frac{(N-1)s_x^2}{N} \tag{5.5}$$

Population parameter estimates depend entirely on which sample, out of all possible samples, was selected in the experiment. To address this, statistical procedures have been developed to judge the quality of an estimated parameter.

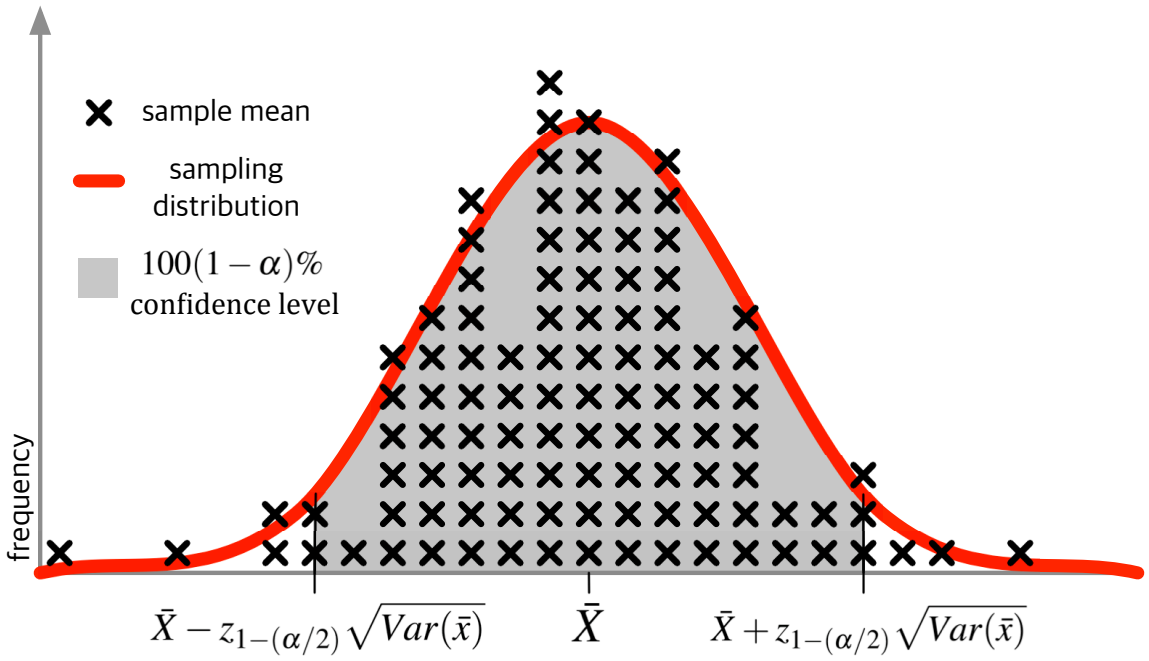


Figure 5.1: Theoretical sampling distribution

Suppose the mean for each possible sample of size n of our population (totaling $\frac{N!}{n!(N-n)!}$ possible samples) was calculated and plotted as a histogram (Figure 5.1). The distribution of these sample means (sampling distribution) has a variance $Var(\bar{x})$ (sampling variance) and a mean (sampling mean) that is equivalent[1] to $\overline{X}$.

Like σ^2, directly computing $Var(\bar{x})$ is too expensive but can be accurately estimated.[2]

$$Var(\bar{x}) \approx \frac{s_x^2(N-n)}{Nn} \tag{5.6}$$

Once an estimator and its estimated accuracy have been computed, we can use normal theory to obtain approximate confidence intervals under a given confidence level $(1-\alpha)$ for the unknown parameter being estimated. The constant $z_{1-(\alpha/2)}$ is the $100[1-(\alpha/2)]$th percentile of the standard normal distribution.

$$\bar{x} \pm z_{1-(\alpha/2)}\sqrt{Var(\bar{x})} \tag{5.7}$$

A confidence interval interpretation is if n elements are sampled from a population repeatedly, with a given sampling strategy, $100[1-(\alpha/2)]\%$ of each sample's confidence interval would include the true (but unknown) population parameter.

A critical assumption of confidence intervals is of normality, or that the sampling distribution is Gaussian in shape. Fortunately, the central limit theorem of statistics guarantees that for large enough sample sizes ($n > 30$), sampling distributions tend to be normal, regardless of the underlying distribution of the element characteristics in the sample.[3]

In other words, given random sampling, enough samples, and no measurement error, calculated confidence intervals are *always* representative of the accuracy of an estimator.

To determine the minimum sample size, the previous equations can be analyzed to derive the following approximate relationship, where ϵ represents the maximum relative difference allowed between the estimated parameter and the unknown true population parameter.

$$n \geq \max\left\{\left(\frac{z_{1-(\alpha/2)}^2 s_x^2}{\epsilon^2 \bar{x}^2}\right), 30\right\} \tag{5.8}$$

By using this equation, we can validate whether our sample size was large enough to give adequate accuracy.

[1] Assuming no measurement error, which is a valid assumption given our simulation technique.

[2] This estimation again assumes no measurement error, as well as a sample size greater than 30.

[3] This guarantee of normality is only for linear estimators (e.g. a mean estimator).

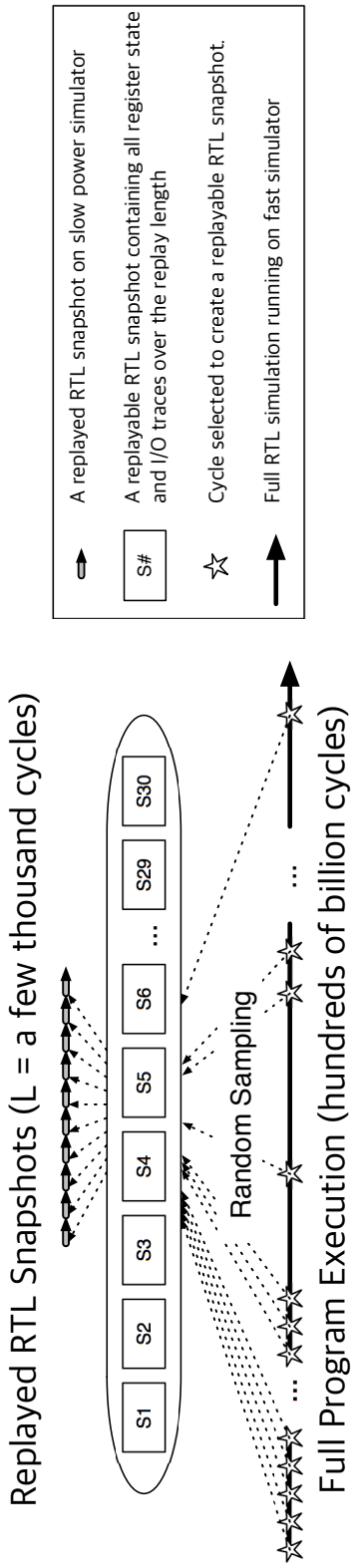


Figure 5.2: Sample-based energy simulation methodology

5.3.2 Sample-based Energy Modeling Methodology

Our sample-based RTL energy simulation methodology quickly and accurately estimates both performance and power of long running applications on arbitrary hardware designs. This methodology obtains random sample points from a fast simulator and replays them on a slow but detailed simulator. Figure 5.2 shows the basic idea behind our methodology.

First, a design's performance is evaluated by an accelerated full-system RTL simulation, during which a set of replayable RTL snapshots is obtained. A replayable RTL snapshot, at cycle c, of a given replay length L, consists of all information necessary to replay from c to $c + L$ on a very slow but extremely detailed RTL/gate-level simulation. More specifically, a replayable RTL snapshot contains all RTL state at cycle c and a trace of all I/O signals of length L starting at cycle c. As an optimization, the I/O traces of a given replayable RTL snapshot are read out from the simulation only when the next replayable RTL snapshot is sampled.

We can obtain the best statistical properties when the replayable RTL snapshots are randomly captured over the course of the program's execution (Section 5.3.1). Since knowing the length of a full program execution is impossible a priori, we employ reservoir sampling [136] to address this problem. With this algorithm and a desired sample size n, the first n replayable RTL snapshots are recorded with the sample size. The kth element where $k > n$ is recorded with the probability of n/k, and then randomly replacing one of the existing replayable RTL snapshots. Note that the probability of selection decreases with longer execution, thus diminishing the sampling overhead. At the end of the program execution, we have n replayable RTL snapshots that were selected at random, without replacement. The simulation time of very long-running applications with sampling is very close to the simulation time without sampling.

In order to replay each replayable RTL snapshot, the RTL state is loaded into the detailed simulator. For each cycle in the replay, the inputs from the I/O trace are fed to the input of the target design, and outputs are verified against the output values of the design. Note that unlike the previous statistical simulation sampling techniques [148], there is no state warming problem due to the exactness of the replayable RTL snapshot. In addition, all replayable RTL snapshots are independent, so we can parallelize their replays on multiple instances of the detailed simulator.

To estimate power, the detailed simulator is a gate-level simulation of the given RTL design. The simulation computes the signal activities of the gate-level design, accounting for detailed timing from floorplanning, placement and routing. An industrial power analysis tool computes the power of each replayable RTL snapshot from the detailed signal activities. By aggregating the power of all replayable RTL snapshots, we can predict the average power and corresponding confidence interval of a full execution of benchmarks.In general, the derived confidence intervals are very small with a small number of replayable RTL snapshots and 99.9% confidence, regardless

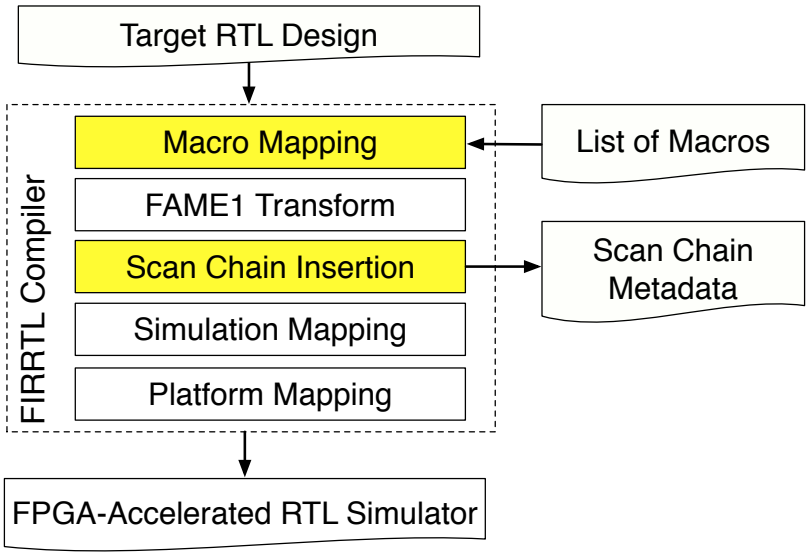


Figure 5.3: FIRRTL compiler passes for sample-based energy modeling

of the length of simulation.

5.4 The Strober Framework

In this section, we describe the Strober framework, our implementation of the sampling-based energy-modeling methodology for RTL designs. Section 5.4.1 shows additional FIRRTL transforms that are necessary for energy modeling in addition to custom passes in Section 3.2.2. Section 5.4.2 describes how RTL snapshots are replayed on gate-level simulation using commercial CAD tools. Section 5.4.3 explains how to estimate DRAM's power consumption using activity counters. Lastly, a simple analytic performance model for the Strober framework is introduced in Section 5.4.4.

5.4.1 Custom Transforms for Sample Replays

Section 3.2.2 describes the FIRRTL compiler passes minimally necessary for FPGA-accelerated RTL simulation. To enable sample-based energy modeling proposed in Section 5.3.2, we need additional custom transforms (yellow boxes in Figure 5.3.2) in the FIRRTL compiler.

By default, the FIRRTL compiler instantiates flip-flops for memory arrays in the RTL design. FPGA tools automatically infer these memory arrays and map them into FPGA-specific block RAMs. However, ASIC tools do not have this capability, and thus, designers need to manually instantiate technology-dependent macro blocks for memory arrays. Unfortunately, this manual mapping should be repeated whenever the target technology is changed, deteriorating the productivity of hardware designs.

Instead, a compiler pass can automatically map technology-independent memory arrays into technology-dependent macros. Specifically, *Macro Mapping* automatically finds the optimal mapping of each memory array within a pool of macros in the target technology and generates required logic for the mapping, greatly reducing designers' manual effort.

Note that, *Macro Mapping* is necessary for both FPGA-accelerated RTL simulation and the replay flow (Section 5.4) to ensure the same RTL is used for both flows, which is important for RTL state snapshot loading on RTL/gate-level simulation. On the other hand, *Macro Mapping* instantiates flip-flops in macro block that are mapped to block RAMS for FPGA-accelerated RTL simulation, while it leaves them as black boxes for the ASIC tool flow.

The *FAME1 Transform* and *Simulation Mapping* are necessary to pause simulation before taking RTL state snapshots. By shutting down the timing token flow toward the target RTL hosted on the FPGA, the simulation can stall at the desired cycle, which is randomly determined by reservoir sampling.

To take RTL state snapshot from FPGAs, scan chains are automatically instrumented from parameterized Chisel RTL using the compilers-in-a-pass technique (Section 2.2.1). As shown in Section 4.3.3, *Scan Chain Insertion* inserts basic scan chains that copy register values immediately after the simulation stalls. On the other hand, special scan chains, which sequentially read each element of memory arrays through address generation, are inserted for large register files and RAMs to preserve their structures. *Scan Chain Insertion* also emits the meta data file that contains the decoding information for the data from scan chains. Unlike RTL debugging, sample-based energy modeling only requires read capabilities of scan chains.

5.4.2 Sample Replays on Gate-Level Simulation

The FPGA-accelerated RTL simulators generated by compiler passes in Section 5.4.1 provide cycle-exact performance estimates, but the replayable RTL snapshots must be simulated on a RTL/gate-level simulator to compute average power. Figure 5.4 shows the tool flow to replay RTL snapshots on gate-level simulation.

We use the same RTL design for both the ASIC tool flow and FPGA-accelerated RTL simulation. The FIRRTL Verilog backend generates Verilog RTL from a given RTL design (e.g. Chisel RTL) for the ASIC tool flow. Next, a gate-level design is generated from the Verilog RTL using a synthesis tool[4] as well as place-and-route tool[5]. Gate-level simulation[6], with very detailed timing, simulates the post place-and-route (PnR) design to compute signal activities for replayable RTL snapshots.

Replayable RTL snapshots are randomly sampled from the FPGA-accelerated RTL simulator, as explained in Section 5.4.1. The RTL state is loaded into the

[4]For synthesis, we used Synopsys Design Compiler J-2014.09-SP4.
[5]For place-and-route, we used Synopsys IC Compiler J-2014.09-SP4.
[6]For gate-level simulation, we used Synopsys VCS H-2013.06.

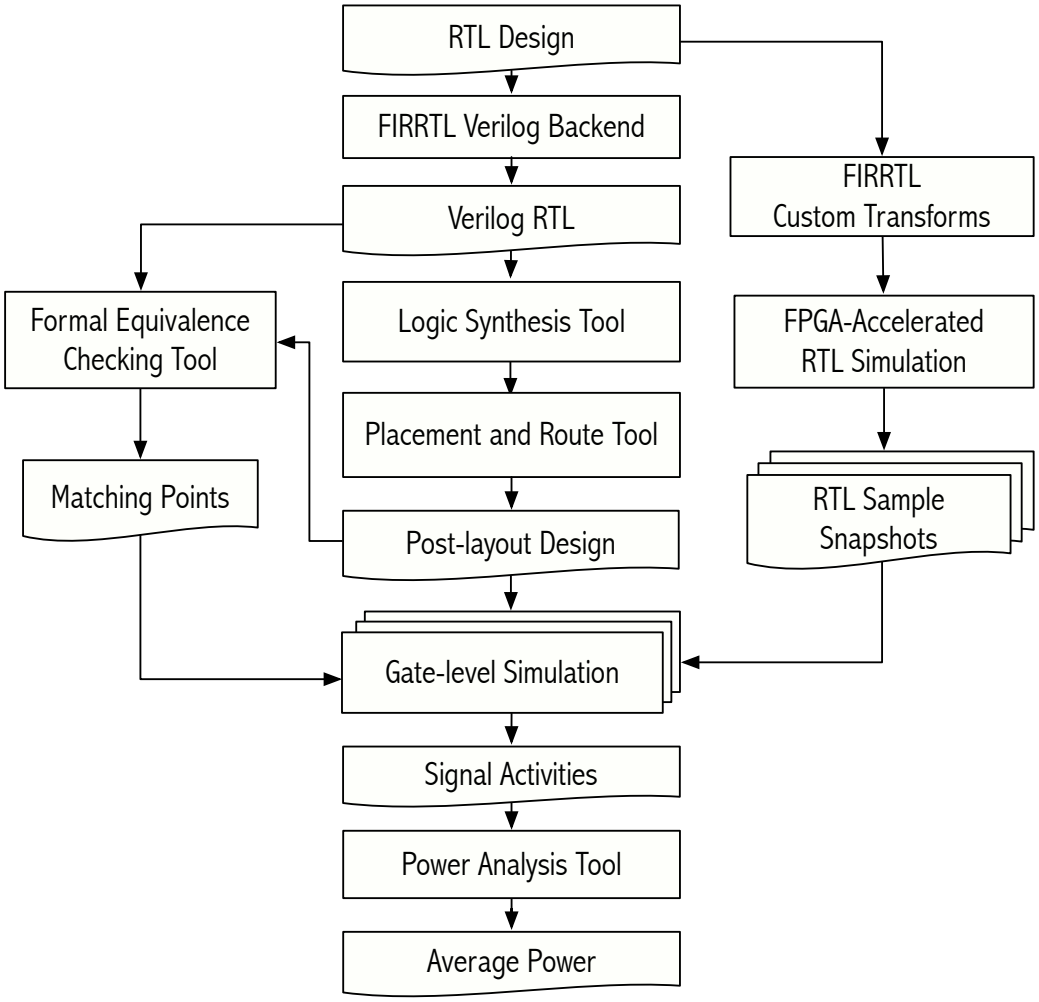


Figure 5.4: RTL snapshot replays with CAD tools for average power estimation

gate-level simulation, and the input traces are fed to the inputs of the design for each sample replay. Moreover, *the output values of the design are compared with the output traces, which ensures samples are replayed correctly.* Samples are independent of one another, so we can replay them on multiple instances of gate-level simulation in parallel.

The generated signal activities are consumed by the power analysis tool[7] to estimate total power consumption for that replayable RTL snapshot. By calculating the mean of each power result, we can obtain the average power of all replayable RTL snapshots. As explained in Section 5.3.1, this average is an accurate estimation of the total application's power consumption on the given RTL design.

However, there are three key challenges to replay samples on gate-level simulation, addressed in the following subsections.

5.4.2.1 Signal Name Mangling in the Gate-level Netlist

One difficulty in initializing the RTL state is that register signal names are mangled by the optimizations performed by CAD tools. Because we cannot use the RTL signal names to load the state snapshots on gate-level simulation, we use a commercial formal equivalence checking tool[8] to match nodes between RTL designs and gate-level designs (Figure 5.4).

The synthesis tool generates information about optimizations applied to a designs to help formal equivalence checking. By using this information, the formal verification tool first finds the matching points between RTL and the gate-level design (including registers) and then verifies the equivalance of the two designs. The matching results of this tool enable us to construct a name mapping table and translate RTL names into gate-level netlist names.

5.4.2.2 State Snapshot Loading on Gate-level Simulation

To load the register values into the gate-level simulation, we originally translated the values into scripts that were read by our commercial Verilog simulator. Unfortunately, this simulator could only execute 400 commands per second, which for a design of 35k flip-flops with 30 replayable RTL snapshots takes 40 minutes to load. While this is unacceptably slow for Strober's framework, writing a customized testbench for each design configuration is very cumbersome and error-prone.

We address this issue by writing a custom state snapshot loader that uses the Verilog Programming Language Interface [128]. The commercial Verilog simulators are compiled with this loader, which handles the snapshot loading commands efficiently. With this implementation, gate-level simulation can handle 20000 commands

[7] For power analysis, we used Synopsys PrimeTime PX J-2014.12-SP2.

[8] We used Synopsys Formality J-2014.09-SP4.

per second, reducing runtime to only 54 seconds for 30 samples with the example in-order processor.

5.4.2.3 Register Retiming

Another big challenge in loading state snapshots is handling register retiming. Register retiming is a technique to move datapath registers, reducing the critical path, area, or both [89]. For example, RTL designers often depend on this technique for writing floating-point units (FPUs), relying on CAD tools to automatically balance the stages in a datapath pipeline. Unfortunately, we cannot easily reconstruct the values of retimed registers from the RTL state snapshot.

Instead, we can capture the I/O values of the retimed datapath. First, note the retimed datapaths are annotated by the designers with the desired latency. For the n-cycle-latency datapath, a custom transform adds shift registers which capture the I/O values for the last n cycles (and the corresponding scan chains). The I/O signals of the retimed datapaths are forced externally in the simulation for n cycles before loading the snapshots to recover their internal state. By starting replays at this point, we can simulate each sample snapshot with fully-recovered state.

5.4.3 DRAM Power Modeling

DRAM power consumption is affected by the DRAM's internal operations (which can be triggered by memory access requests) and its internal state. For example, DRAM's internal read and write operations trigger data transfer through DRAM's I/O bus, causing dynamic power consumption. However, knowing the physical address mapping, the DRAM controller's policies, and all memory access requests is enough to predict any given DRAM's internal operations, and thus predict its power consumption. As in the experimental settings specified in Kim et al. [74], we use Micron's LPDDR2 SDRAM S4 [101] with eight banks, and 16K (16×1024) rows for each bank. We assume a bank-interleaved memory mapping where adjacent memory addresses are distributed across different banks. Finally, we assume an open-page policy, where DRAM banks are kept active after a row access.

To capture the DRAM memory requests, we attach counters to the memory request output ports. Using the known memory mapping, the physical address of each emory request is translated into the bank number and the row number. The previously accessed row and bank numbers are stored with the counter data to enable determining whether the row activation operation will occur. From the counter values, we know the number of read/write operations and the number of row activation operations. With this information and DRAM configurations, the DRAM power can be calculated using a spreadsheet power calculator provided by Micron [100].

5.4.4 Simulation Performance Model

To demonstrate the opportunity for significant speedup over the existing CAD tools, we present a simple analytic performance model of the Strober framework in this section. To estimate the overall time, we should consider (1) the synthesis time for the FPGA-accelerated RTL simulator, (2) the FPGA-accelerated RTL simulation time, (3) the ASIC tool chain time (logic synthesis, placement, routing, and formal verification), and (4) the replay time for sample snapshots. Note that (3) is independent from (1) and (2), so the overall time is expressed as follows:

$$T_{overall} = max(T_{FPGAsyn} + T_{FPGAsim}, T_{ASIC}) + T_{replay}$$

The ASIC tool chain time, T_{ASIC}, tends to be long for complex designs. However, we run very long-running application on the FPGA simulator, thus resulting in $T_{ASIC} < T_{FPGAsyn} + T_{FPGAsim}$. In this chapter, the synthesis time for the FPGA simulator, T_{FPGA_syn}, can be up to one hour with a two-way out-of-order processor while T_{ASIC} is around three or four hours. Also note that $T_{FPGAsyn} \ll T_{FPGAsim}$ for real-world long-running applications.

To estimate $T_{FPGAsim}$, assume the FPGA simulation runs at K_f Hz. Let N and L be the total simulation cycles and the replay length respectively. Reservoir sampling [136] ensures that the number of elements recorded during the simulation is roughly $2nln((N/L)/n))$ with the sample size n. The FPGA simulation time, $T_{FPGAsim}$, is therefore:

$$T_{FPGAsim} = T_{run} + T_{sample} \approx N/K_f + T_{rec} \times 2nln(N/nL)$$

where T_{run}, T_{sample}, T_{rec} are the simulation running time, the total sampling time, and the time to read out a single replayable RTL snapshot, respectively.

T_{replay} is decomposed into (1) the snapshot loading time, (2) the snapshot replay time, and (3) the power analysis tool time. The snapshot loading time is considered because it can be very slow without a proper implementation (Section 5.4.2.2). For the snapshot replay time, suppose the RTL/gate-level simulation runs at K_g Hz. In addition, only L cycles are replayed for each sample snapshot. We provide the switching activity interface format (SAIF) files to the power analysis tool for the average power of each sample snapshot, and thus, the power analysis time is independent of the length of each sample snapshot. Lastly, each snapshot replay is independent one another, and thus, can be parallelized. Therefore, assuming P instances of gate-level simulation, the total replay time is:

$$T_{replay} = \frac{n \times (T_{load} + (L/K_g) + T_{power})}{P}$$

where T_{load} is the time to load each RTL state into the gate-level simulation, and T_{power} is the time to run the power analysis tool for a single sample snapshot.

For the example two-way out-of-order processor used in this chapter, the FPGA synthesis time with Strober was around one hour[9], the FPGA simulation runs at 3.6 MHz, and the gate-level simulation runs at 12 Hz. In addition, the recording time per replayable RTL snapshot is 1.3 seconds, the sample loading time on gate-level simulation is 3 seconds, and the time for power analysis[10] is around two and a half minutes. Suppose we simulate a benchmark whose execution length is 100 billion cycles on the two-way out-of-order, has a sample of 100 replayable RTL snapshots (with replay length of 1000 cycles), on 10 instances of gate-level simulation. Plugging these numbers to the equations, we can calculate the overall simulation time:

$$T_{FPGAsyn} = 3600\ s$$

$$T_{run} = \frac{10^{11} cycles}{3.6 \times 10^{6} Hz} = 27778\ s$$

$$T_{sample} = 1.3 \times 100 \times 2 \times ln(\frac{10^{11}}{100 \times 10^{3}})) = 3592\ s$$

$$T_{replay} = \frac{100 \times (10^{3} cycles/12Hz + 150)}{10} = 2333\ s$$

Thus, $T_{overall} = T_{run} + T_{sample} + T_{replay} = 33703$ seconds or 9.4 hours. Note that it will take $10^{11} cycles/300KHz = 3.86$ days even on fast microarchitectural software simulators and $10^{11} cycles/12Hz = 264$ years on gate-level simulation!

5.5 Evaluation

5.5.1 Target Designs

To demonstrate Strober's ability to augment arbitrary Chisel RTL, we evaluated two different synthesizable open-source cores, both which leverage the open-source *Rocket-Chip* SoC generator [8]. The first core is *Rocket*, a 5-stage single-issue in-order core. The second core is *BOOM*, a parameterized, superscalar out-of-order core [34]. Both cores implement the full 64-bit scalar RISC-V ISA, which includes support for atomics, IEEE 754-2008 floating-point, and page-based virtual memory.

Note that the Strober framework is built upon commercial CAD tools, which report accurate timing and area for RTL designs. Figure 5.5 shows a sample floorplan of the two-way superscalar out-of-order processor. We synthesize and place-and-route the designs in TSMC 45nm. For this evaluation, both cores were simulated at 1 GHz frequency, however silicon implementations of Rocket have been demonstrated to reach 1.3 GHz [88] and 1.65 GHz [126] in an IBM 45nm SOI technology.

[9]For FPGA synthesis, we used Vivado ® 2014.4.

[10]For power analysis, we again used PrimeTime ® PX J-2014.12-SP2.

	Rocket	BOOM-1w	BOOM-2w
Fetch-width	1	1	2
Issue-width	1	1	2
Issue slots	-	12	16
ROB size	-	24	32
Ld/St entries	-	8/8	8/8
Physical registers	32(int)/32(fp)	100	110
L1 I$ and D$	16KiB/16KiB	16KiB/16KiB	16KiB/16KiB
DRAM latency	100 cycles	100 cycles	100 cycles

Table 5.2: Target designs evaluated with Strober

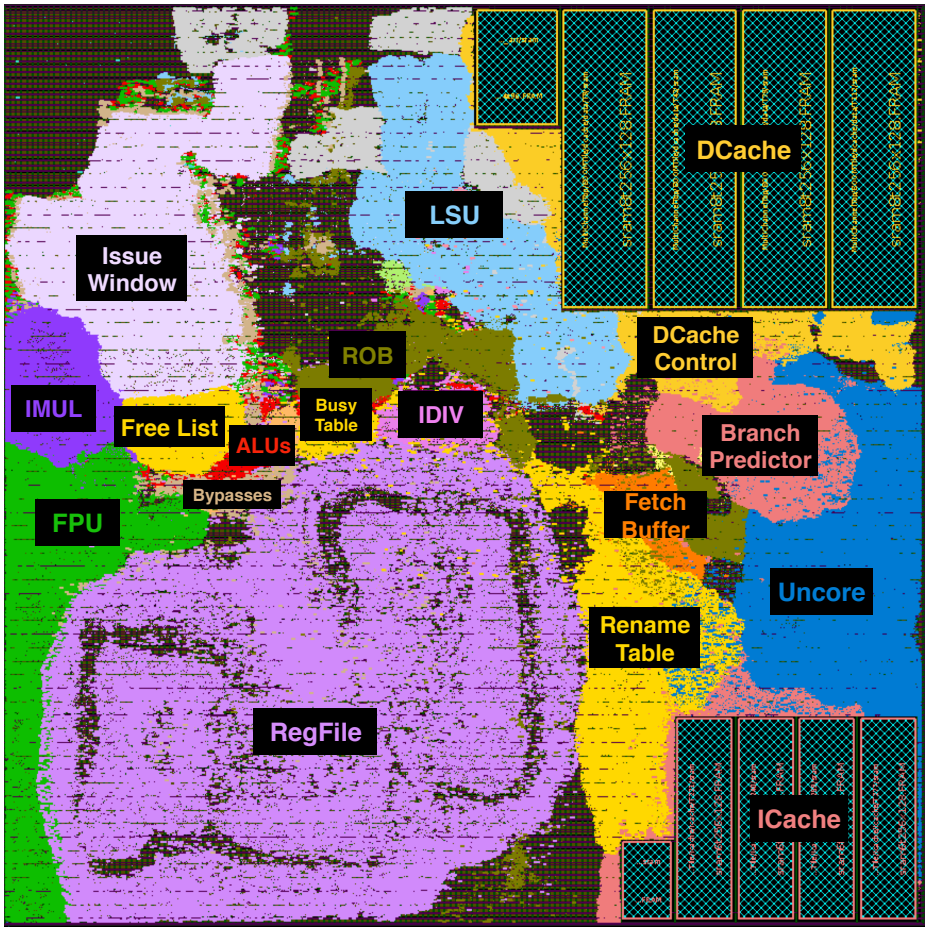


Figure 5.5: Floorplan of BOOM-2w

	LinuxBoot	**Coremark**	**gcc**
Simulation Cycles (10^9)	0.5	3.92	73.39
Record Counts	980	1116	1497
Simulation Time with Sampling (min)	12.88	32.80	344.00
Simulation Time without Sampling (min)	3.68	11.00	312.25

Table 5.3: Simulation performance for BOOM-2w

5.5.2 Benchmarks

We chose three disparate workloads to demonstrate Strober's ability to measure target design performance, power, and energy usage. The first is CoreMark, a benchmark designed to stress processor pipelines [3]. The second workload boots the RISC-V port of Linux on a small BusyBox disk image, executes the `uname` and `ls` commands, and then powers down. The third workload executes the SPECint benchmark `403.gcc` [2] on Linux. For `gcc`, we execute the first 20B instructions (or 20%) of the SPECint reference input workload "gcc 166.in".

5.5.3 Simulation Performance

For Rocket Chip target systems running under Strober, target I/O devices are mapped to software on the host CPU, not the FPGA, causing a communication overhead that stalls the simulator every 256 cycles. The target simulator is also stalled while capturing a replayable RTL snapshot.

Table 5.3 shows the performance evaluation of Strober with BOOM-2w running long benchmarks showed in 5.5.2. The record counts, the number of sample recording during each simulation run, only moderately increases as explained by reservoir sampling. Therefore, the sampling overhead is very small for long-running simulations.

For the `gcc` runs of 70 billion cycles, Strober achieved a simulation speed of around 3.56 MHz. For comparison, the unmodified Rocket and BOOM cores both can be synthesized at 50 MHz on the same ZC706 FPGA.

5.5.4 Power Validation

To validate our Strober framework and the sample-based RTL energy modeling methodology, we run the microbenchmarks included in the Rocket-Chip framework to completion on a gate-level-simulation of Rocket. The switching activity for he entire benchmark is used to calculate the actual average power. Also, we obtain 30 random sample snapshots of 128 cycles from the FPGA simulation, and by replaying these,

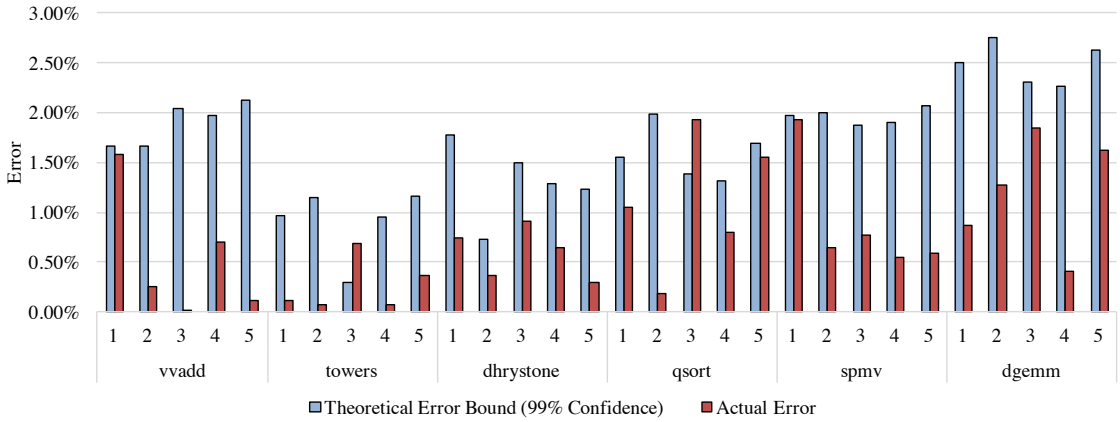


Figure 5.6: Confidence intervals (theoretical error bounds) vs. actual errors

Benchmark	**Simulated Cycles**	**Replayed Cycles**	**Coverage**
vvadd	200521	30×128	1.92%
towers	410752	30×128	0.93%
dhrystone	396790	30×128	0.97%
qsort	187160	30×128	2.05%
spmv	927144	30×128	0.41%
dgemm	1833075	30×128	0.21%

Table 5.4: Simulated and replayed cycles for each benchmark on Rocket

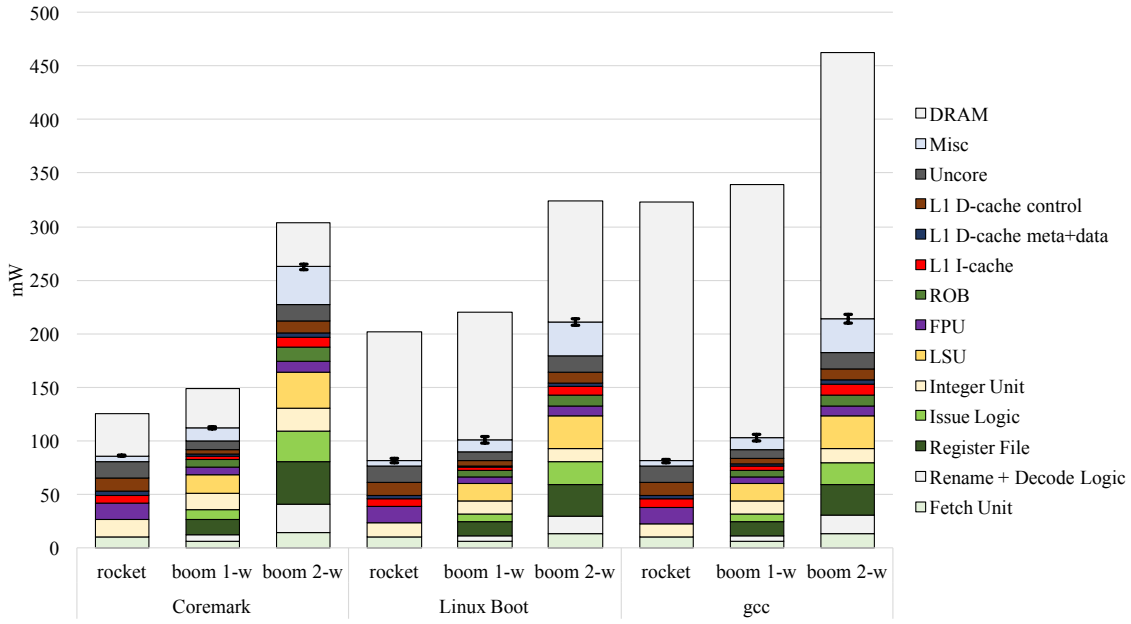


Figure 5.7: Power breakdown with error bounds using 30 random samples from CoreMark, LinuxBoot, and `403.gcc`

we calculate the average power as well as their error bounds with 99% confidence. Then, we compare those error bounds over the actual errors as in Figure 5.6. We repeated this process five times for each benchmark.

Note that even though the samples cover only less than 2.1% of the cycles as shown in Table 5.4, the errors tend to be very small. Moreover, in most cases, the actual errors are within the error bounds computed from the samples. This also shows that *the errors are independent of the length of execution.* While the third sampling of *towers*, and the third of *qsort* are slightly outside their error bounds, this result is somewhat expected due to the probabilistic nature of statistical sampling. Nevertheless, their actual errors are still very small, less than 2%.

5.5.5 Case Study

Figure 5.7 compares the energy breakdown of the Rocket, BOOM-1w, and BOOM-2w cores using 30 random sample snapshots for each benchmark. The performance differences between the cores is easiest to see when running CoreMark, a small benchmark designed to fit in L1 caches and stress processor's integer pipelines. BOOM-1w is 9.8% faster than Rocket, and BOOM-2w is 58% faster. However, BOOM-2w uses 3× the power, while Rocket is the most energy-efficient.

The other benchmarks use a much larger memory footprint than CoreMark, as seen in the increased DRAM power usage. On Linux-boot, clock for clock, Rocket's

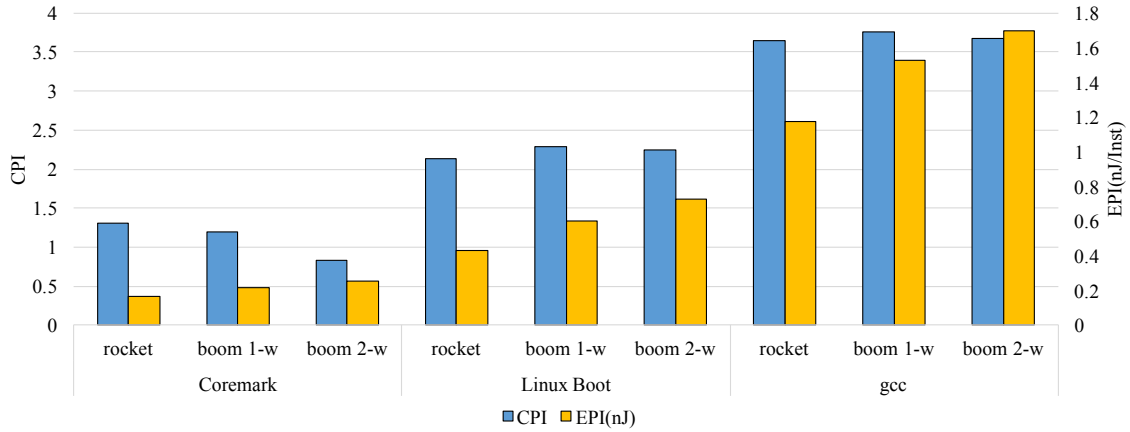


Figure 5.8: Performance and energy efficiency for CoreMark, LinuxBoot, and `403.gcc`

shorter branch resolution latency allows it to outperform BOOM, which has only a simple branch predictor in the version used in this case study.

Details aside, this case study shows the validity of using Strober as a basis for design-space exploration in architecture research. With Strober, researchers now have the ability to run real programs on RTL with a full evaluation of energy, area, and performance. In addition, each sample snapshot contains a timestamp, so by using performance counters we can correlate performance and power at a specific point as shown in Figure 5.9. The CPI is sampled every 100M cycles by a separate user program running on Rocket. Grey vertical lines denote when a Strober snapshot was taken. Using this case study as an example, the turn-around time for evaluating 70 billion cycles on BOOM-2w is approximately 7 hours for a complete evaluation. We believe this is fast enough to enable realtime feedback in the RTL design loop.

5.5.6 Power and Energy Efficiency for SPECint2006

With significant improvements on FPGA-accelerated RTL simulation as shown in Chapter 3, we can evaluate power and energy efficiency of Rocket and BOOM with each benchmark in SEPCint2006 [2] to completion. Specifically, with *I/O endpoints* that significantly reduce the communication overhead between the FPGA and the CPU (Section 3.2.3), the average simulation rate of BOOM-2w was *18 MIPS* with Xilinx ZC706 for the SPECint2006 benchmark suite. In this section, the simulation setup is the same as in Section 3.3. For power estimation, we used the Synopsys 32nm Educational Technology and randomly sampled 50 RTL state snapshots from each benchmark.

We can check whether or not the designs are realistic by examining their power consumption. Figure 5.10 shows the power breakdowns for Rocket and BOOM-2w with 32 KiB L1 caches for SPECint2006 with 95% confidence intervals. We can see

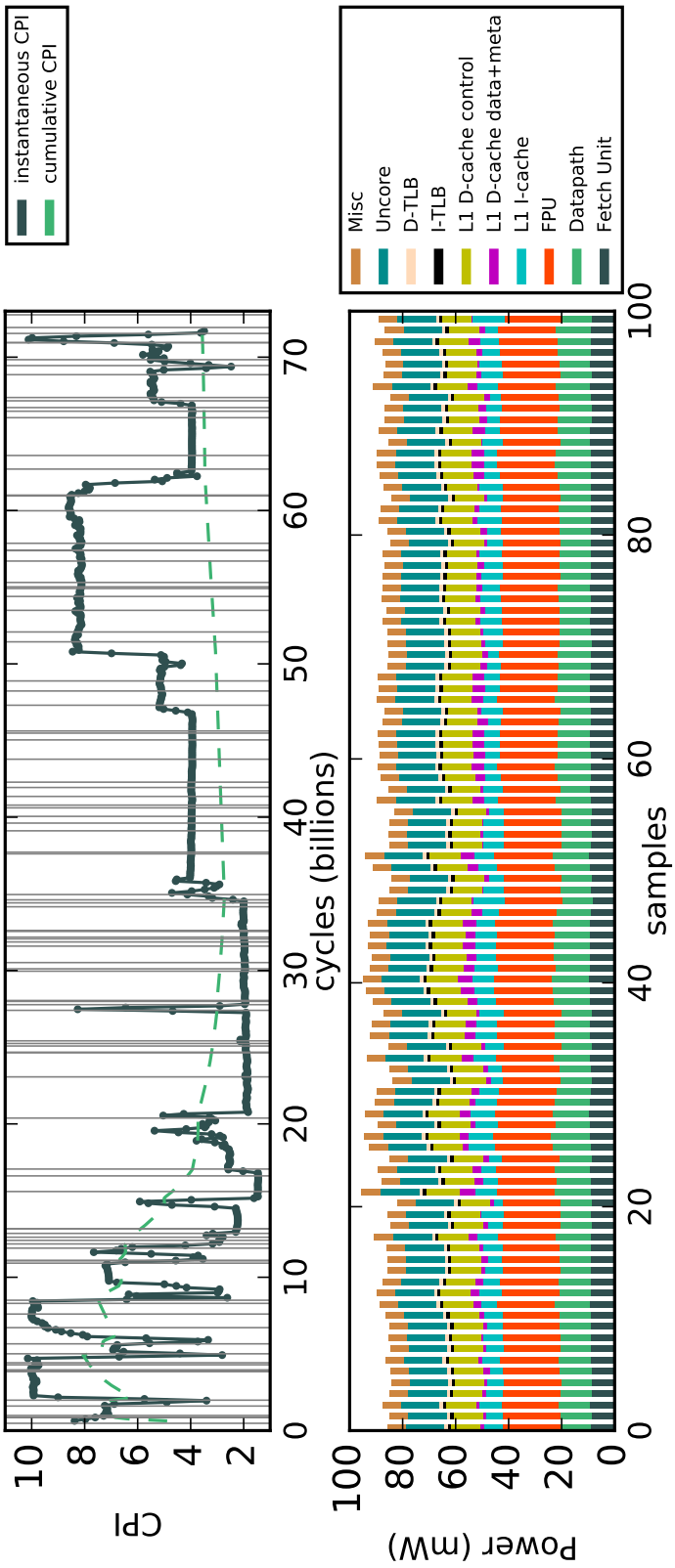


Figure 5.9: The CPI of the first 20B instructions (or 20%) of `403.gcc` as executed on Rocket

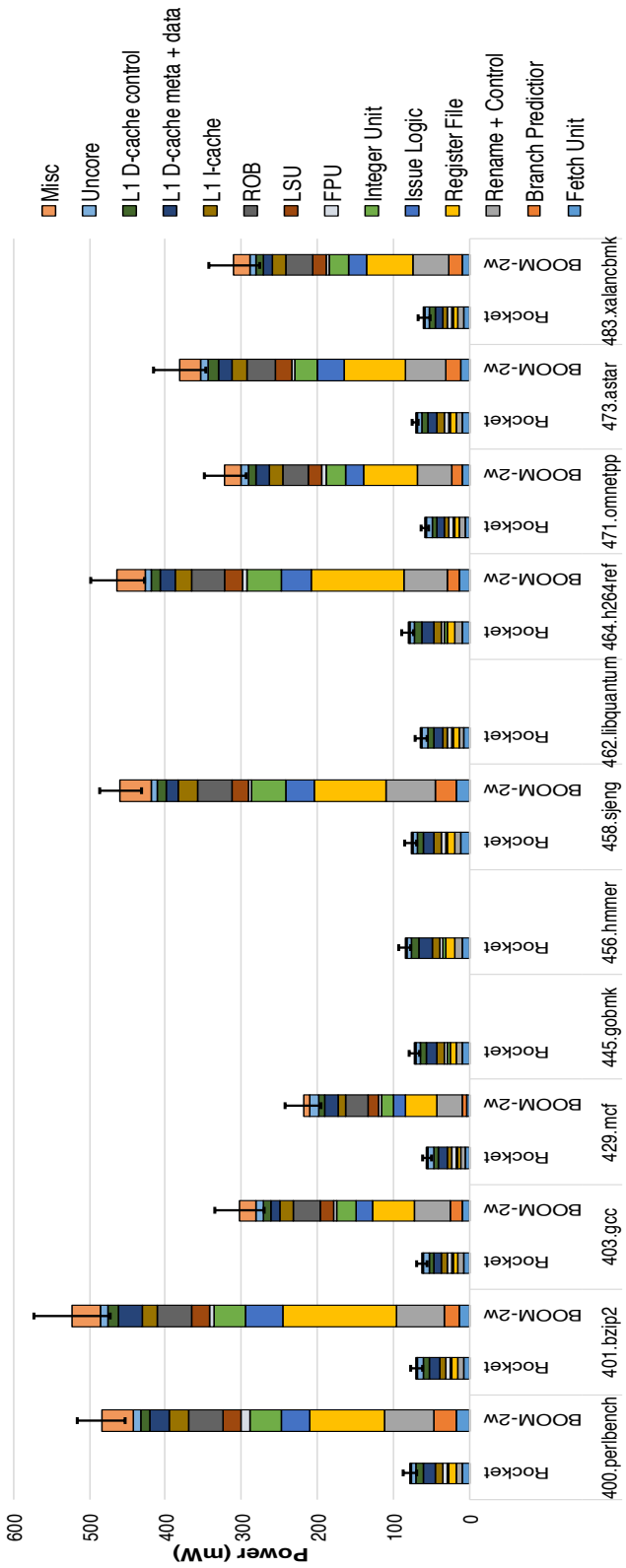


Figure 5.10: Power estimates of Rocket and BOOM-2w with 32 KiB L1 caches for the SPECint2006 benchmarks

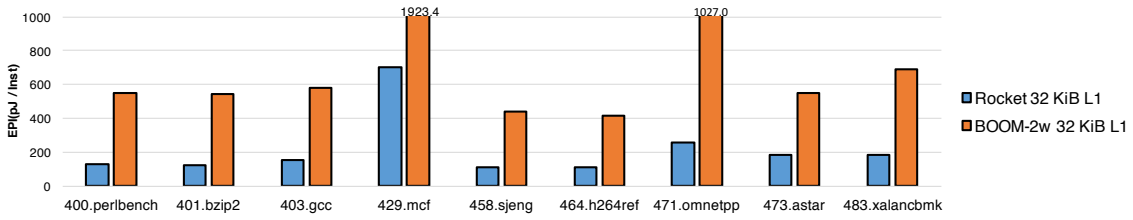


Figure 5.11: Energy efficiencies of Rocket and BOOM-2w with 32 KiB L1 caches for the SPECint2006 benchmarks

out-of-order processors can easily be power inefficient. Specifically, the register file and the rename logic are unrealistic as they consume up to 40% of the total power. Improving energy efficiency is a key motivation of a new version of BOOM [33], which improves its microarchitecture from BOOM v1 [34].

Figure 5.11 shows the energy efficiencies of Rocket and BOOM-2w with 32 KiB L1 caches. BOOM-2w is much less energy efficient than Rocket: BOOM-2w burns more power (Figure 5.10) without proportionally increasing IPC (Figure 3.5). Notably, the energy efficiency of `429.mcf` is worse than any other benchmarks although it consumes the least power. Therefore, evaluating just either performance or power but not both is insufficient if energy efficiency is a primary concern, as it is for most classes of computer systems.

5.6 Summary

In this chapter, we presented a sample-based RTL energy modeling methodology that captures replayable RTL snapshots from a fast performance simulation and replays them on a detailed power simulation. We showed the statistical robustness of this methodology, including the ability to generate confidence intervals for any power prediction.

Next, we introduced Strober, a framework for taking existing RTL designs written in the Chisel hardware construction language, and generating a cycle-accurate, decoupled simulator that can be executed on an FPGA. The instrumented simulator can be used to not only measure the cycle-accurate performance of the RTL design, but to generate random RTL snapshots that can be replayed (in parallel) in a detailed gate-level simulator. We also demonstrated significant theoretical speedups using an analytical model for simulation performance.

We then validated our methodology and framework for simulation performance, and power accuracy. Finally, we demonstrated our framework by running three complex RTL designs through our toolchain to obtain timing, area, performance, and average power for a variety of benchmarks. These case studies serve as an example of how Strober can not only provide ground truth for building faster and more flexible

abstract power models, but can in and of itself be a tool for design-space exploration at the RTL level.

Strober is open-source and freely available [11]. The commercial CAD tools used in this chapter are industry-standard, and widely available to academics through academic licensing programs.

[11] strober.org

Chapter 6

Runtime Power Modeling

This chapter presents Simmani, an activity-based runtime power modeling that automatically identifies key signals for power dissipation of any RTL design. Section 6.1 motivates why runtime power modeling is necessary for computer systems. Section 6.2 covers existing runtime power modeling methodologies. Section 6.3 describes how Simmani selects key signals for power dissipation with signal clustering and then trains model-level runtime power models with regression against power traces from CAD tools. Section 6.4 explains how Simmani automatically instruments activity counters collecting runtime statistics to enable runtime power analysis with FPGA-based simulation. Section 6.5 and shows the evaluation results of Simmani with Rocket and BOOM. Section 6.6 shows the evaluation results of Simmani with Hwacha. Section 6.7 summarizes this chapter.

6.1 Motivation: Is Activity-Based Runtime Power Modeling Necessary?

As power and energy efficiency has been the primary concern for both low-power portable computers and high-end servers, runtime power estimation plays an important role not only in validation of hardware design ideas during the design process but also in effective runtime power, energy, and thermal optimizations and management. As a result, there has been significant prior work on various power-modeling methodologies.

Power modeling using performance counters has been widely adopted for runtime power and thermal management for real microprocessors [14, 93, 60, 20, 15, 125]. Power models are constructed in terms of statistics from existing performance counters, and calibrated against power measurement from real systems. These power models provide quick power estimates by profiling full execution of applications, which can be further taken advantage of by runtime power and thermal optimizations such as

dynamic voltage and frequency scaling (DVFS). However, this method has been only successful for well-known traditional microprocessors with their existing prototypes. With a novel hardware design, designers should manually identify microarchitectural activities highly correlated with dynamic power dissipation, which is also extremely difficult for non-traditional hardware designs.

With the slow down in historical transistor scaling, the only way to sustain performance gain is through specialization with application-specific accelerators. Indeed, RTL implementation has become a standard procedure in computer architecture research to estimate the area, power, and energy for novel design ideas. However, dynamic power dissipation is not one-dimensional and cannot be statically determined as it heavily depends on signal activities that can vary across different workloads. Moreover, runtime power, energy, and thermal-management techniques should be studied for novel hardware designs to improve their energy efficiency. For this reason, a general, accurate, and efficient runtime power modeling methodology is required for future architecture research.

In this chapter, we present Simmani, a novel activity-based runtime power modeling methodology using automatic signal selection for any RTL designs. Our methodology is developed from the observation that *signals showing similar toggle patterns have similar effect on dynamic power dissipation*. In the power modeling flow, the toggle pattern matrix, where each RTL signal is represented as a high-dimensional point, is constructed from VCD dumps generated from RTL simulation of the training set. As similarities of signals are quantified by the Euclidean distances between two points, an optimal number of signals are selected through clustering with dimensionality reduction. Then, the power model is trained through regression against cycle-accurate power traces from industry-standard CAD tools.

6.2 Existing Runtime Power Modeling

6.2.1 Power Modeling with Performance Counters

Power modeling based on performance-monitoring counters is also popular for power estimation [14, 93, 60, 20, 15, 125]. This method provides a quick power estimate, which is also useful for runtime power/thermal optimizations, by profiling full execution of applications. On the other hand, LeBeane et al. [77] shows a power modeling methodology that maps platform-specific event counters to McPAT's event counts.

There are also studies on phase/kernel-based power modeling. Isci et al. [62] characterized power phases with event counter statistics collected with dynamic binary instrumentation. Zheng et al. [152] present a cross-platform phase-based power modeling methodology that predicts the target design's power from the host platform's counter statistics. Wu et al. [147] and Greathouse et al. [51] develop a GPGPU

performance and power modeling methodology that clusters training kernels based on performance scaling behaviors and classifies the group of a new kernel with neural nets based on performance counter values.

However, these methodologies are limited to well-known microprocessors with existing silicon implementations. For novel hardware designs, computer architects need intuition to define representative microarchitectural events highly correlated with power dissipation, which is extremely difficult without collecting empirical data from real silicon implementations. Even if we can identify important events for power consumption, it is very hard to collect their statistics for long-running applications without existing implementations.

Unlike the previous work on event-based power modeling, Simmani trains high-fidelity runtime power models by automatically selecting an optimal number of signals for any RTL designs. In addition, activity counters collecting signal statistics are also automatically instrumented to provide quick power estimates with FPGAs, which enables various power/thermal case studies in the SW/HW co-design flow for novel hardware designs with non-trivial applications.

6.2.2 Statistical Modeling with Microarchitecture Parameters

There are a significant amount of previous work on statistical performance/power modeling for uniprocessors [78, 79, 67, 66, 42] and chip multiprocessors [59, 73, 80]. Regression [78, 79, 80, 67, 42] or neural network [59, 66, 73] models in terms of microprocessor parameters are trained from simulations of a small number of configurations to predict performance and power for unseen configurations without detailed simulations.

However, all these models are constructed in the microprocessor context. For non-traditional hardware designs, high-level microarchitectural parameters should as be carefully identified with designers' intuition. In contrast, Simmani inspects all signal activity in the RTL design and selects a small number of signals with statistical techniques without requiring manual effort.

6.2.3 Cycle-Level RTL Power Modeling

Activity-based cycle-level RTL power modeling is also explored in previous work [97, 52, 24]. Metha et al. [97] build table-based power models for small-size modules by clustering their input transitions resulting in similar energy dissipation to reduce the number of entries in the table. Our approach is different in that we cluster signals based on their toggle patterns to choose a small number of signals as regression variables. Gupta et al. [52] construct four-dimensional table-based macro models for combinational logic indexed by input/output signal switching activities.

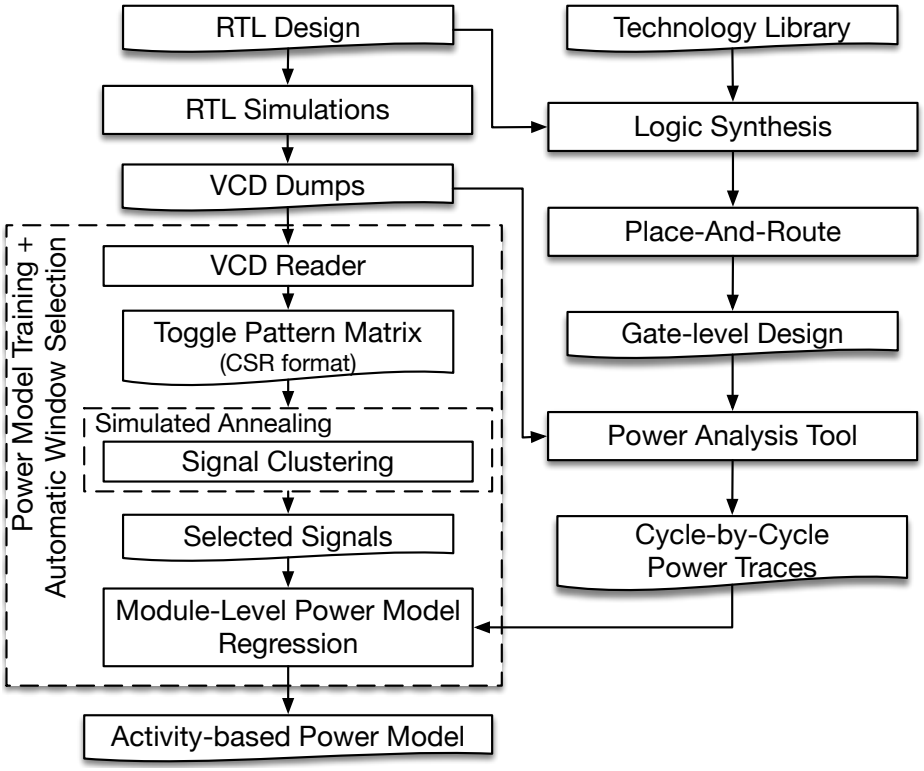


Figure 6.1: Tool flow for runtime power modeling

Bogliolo et al. [24] build regression-based RTL power models in terms of input and output signals of combinational macro blocks divided by registers. This approach is not scalable since all switching activity of registers needs to be tracked, which is intractable for complex hardware designs. In contrast, Simmani is scalable as it automatically selects a small number of signals from a large-scale design for power-model regression.

Zoni et al. [153] select signals from input/output signals in the module boundaries of the design hierarchy, construct a linear power model in terms of these signals, and instrument the runtime power model visible by software. In general, input and output signals are not the most correlated with power dissipation of a given module, and thus, a smaller number of internal signals are preferred to a larger number of input and output signals for accurate power modeling.

6.3 Power Model Training

In this section, we present the Simmani framework that automatically selects signals most correlated with power dissipation and trains power models in terms of the selected signals for any RTL design. The core idea is to *cluster signals showing*

similar toggle patterns to choose distinctive signals, and then, *train power models in terms of these signals using cycle-accurate power traces*. The intuition is that signals showing similar toggle patterns have similar effect on dynamic power dissipation and can be *factored* to share the same coefficient in the power model, minimizing modeling error. Figure 6.1 describes the overall power modeling flow in the Simmani framework.

Section 6.3.1 introduces the power modeling background. Section 6.3.2 explains how the toggle patten matrix is constructed from VCD dumps. Section 6.3.3 describes how important signals for power dissipation are found through clustering. Section 6.3.4 explains how the optimal number of signals is determined through model selection with simulated annealing. Section 6.3.5 explains how to obtain detailed cycle-accurate power traces from commercial CAD tools. Section 6.3.6 shows how power models in terms of selected signals are trained through regression against cycle-accurate power traces. Section 6.3.7 presents how the optimal window size for the toggle pattern matrix is automatically selected.

6.3.1 Power Modeling Background

CMOS power consumption can be decomposed into three major factors:

$$P_{total} = P_{dyn} + P_{dp} + P_{leak} = \alpha f(C_L V_{DD}^2 + V_{DD} I_{peak} t_s) + V_{DD} I_{leak}$$

The dynamic power, P_{dyn}, is consumed when the capacitance, C_L, is charged or discharged, while the direct-path power, P_{dp}, is consumed during rise/fall times due to short-circuit current, $I_{peak} t_s$, when transistors are switching. Both cause power dissipation when signals *toggle*, the ratio of which is captured by the activity factor, α. The leakage power, P_{leak}, is, on the other hand, consumed due to leakage current, I_{leak}, even when transistors are not switching.

We may assume leakage power is constant under the condition that the temperature is well-controlled and the threshold voltage does not change dynamically. In this case, the leakage power can be statically computed by CAD tools. In addition, the direct-path power is minimized by CAD tools, and thus, much smaller than dynamic power. However, dynamic power, a primary factor in power dissipation, is hard to determine statically since the activity factor is highly workload-dependent. Therefore, we should collect activity statistics from simulations to measure the dynamic power dissipation.

Dynamic power can be computed by summing signal toggle densities over all CMOS gates [104]:

$$P_{dyn} = \frac{1}{2} V_{DD}^2 \sum_{g \in \{gates\}} C_g D_g$$

where C_g and D_g are the load capacitance and the toggle density of gate g, respectively. Unfortunately, such toggle densities are only available through extremely

detailed gate-level simulation, which is not practical for collecting related statistics from real-world workloads running on complex hardware designs.

Therefore, for large-scale designs, we approximate the dynamic power in terms of event statistics associated with their effective capacitances:

$$P_{dyn} \approx \frac{1}{2} V_{DD}^2 \sum_{e \in \{events\}} C_e D_e$$

where C_e and D_e are the effective capacitance and the statistics of event e, respectively.

Microarchitectural power models such as Wattch [26] and McPAT [92] analytically compute capacitances for regular structures [108] and collect manually identified event statistics from microarchitectural software simulators [19, 143, 109] before RTL implementation. Performance-counter-based power modeling [14, 93, 60, 20, 15, 125] uses existing counters in the system, and finds the effective capacitance of each counter event through regression against power measurement of the real machine. These methodologies have been effective for well-known traditional microarchitectures.

However, for arbitrary novel designs, these approaches are very challenging as 1) manually selecting important signal/event activities is difficult and 2) finding the effective capacitance can be also hard. In the following sections, we tackle both problems for any RTL design automatically and systematically.

6.3.2 Toggle Pattern Matrix from VCD Dumps

The first step for power-model training is to construct the *toggle pattern matrix* using VCD dumps from RTL simulations of the training set. For accurate power modeling, we carefully choose small workloads that represent real-world applications. If the training set is too small, the trained model cannot accurately predict power consumption of unseen workloads. If the training set is too large, the model training is bottlenecked by RTL simulation and power analysis tools that need to process a large volume of VCD dumps. In this paper, we choose ISA tests and microbenchmarks and replays of random sample snapshots from long-running applications.

The toggle-pattern matrix is a collection of toggle-density vectors of all signals in the RTL design. Each element of this matrix is constructed as follows:

$$v_{ij} = \frac{\text{total number of toggles of signal } i \text{ over window } j}{\text{width of signal } i \times \text{window size}}$$

where v_{ij} is the element at row i and column j of the toggle pattern matrix.

Figure 6.2 shows a simple example of how the toggle-pattern matrix is constructed from VCD dumps with a window size of two cycles. For single-bit signals, the number of toggles is just the number of value transitions. For example, the total number of value transitions of signal `a` over window 1 is 2, and thus, $v_{a0} = 2/2 = 1.0$. The other elements for signal `a` and `b` are computed in the same way.

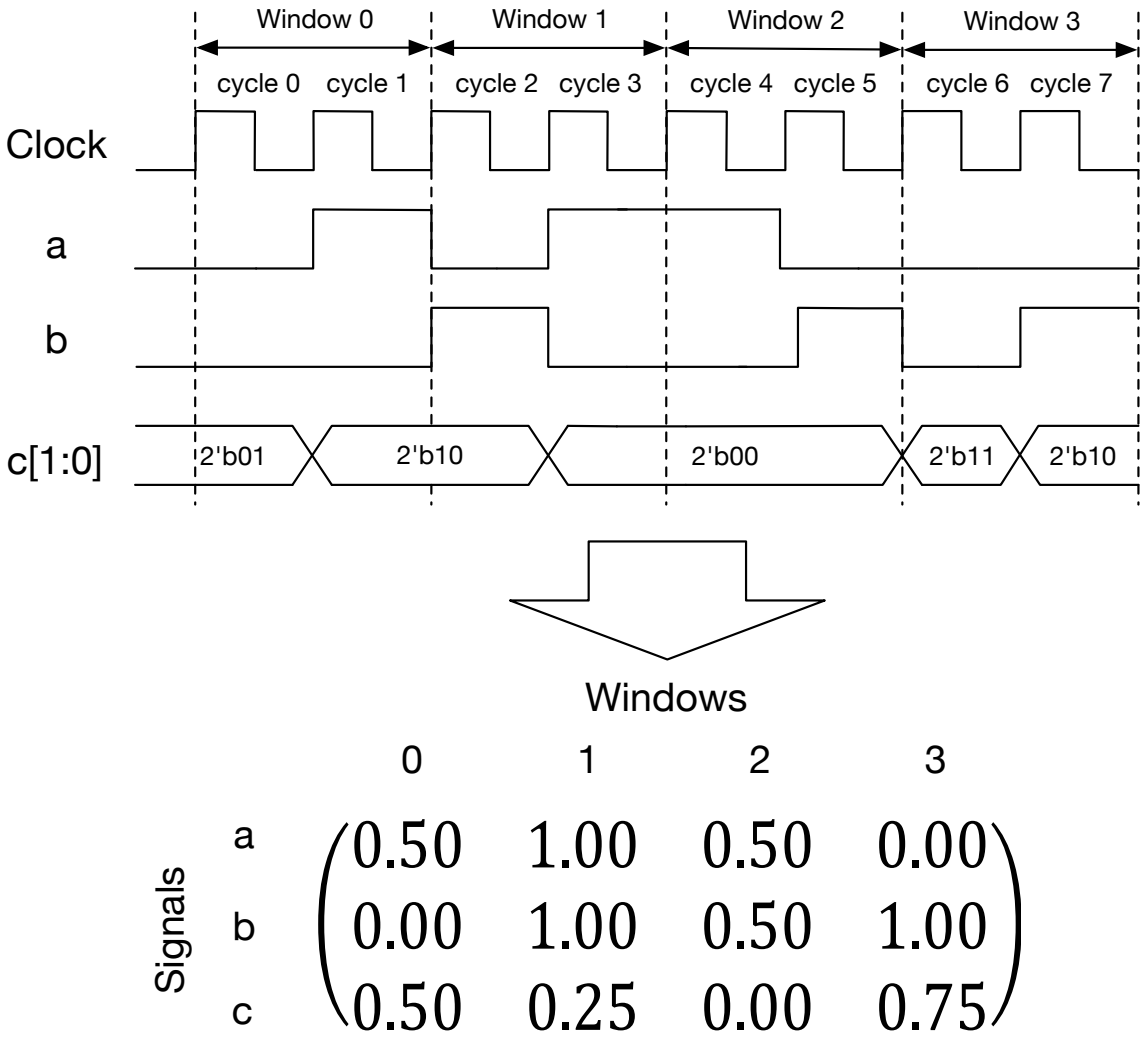


Figure 6.2: A simple example to construct toggle pattern matrix

For multi-bit busses, the number of toggles is the Hamming distance between the value at the previous cycle and the value at the current cycle. For example, the Hamming distance of signal `c` at cycle 1 is 2. The reason each matrix element is divided by the width of the signal is we want to group busses of different widths together in the same cluster if they show similar toggle patterns. Hence, $v_{c0} = 2/(2 \times 2) = 0.5$.

The similarity matrix is very large for a complex hardware design. But most entries in this matrix are zeros for a typical hardware design, since only a small number of signals tend to be active in a given time slot. Therefore, the similarity matrix is represented as a sparse matrix using the compressed sparse row (CSR) format.

The similarity of two signals is measured by the Euclidean distance between two vectors. It is intuitive that two signals having a short distance between them have a similar effect on power dissipation. In this case, the window size plays an important role in quantifying similarities. Selecting the optimal window size is discussed in Section 6.3.7

6.3.3 Automatic Signal Selection through Clustering

Once the toggle pattern matrix is constructed, we want to partition signals into a handful of groups, each of which collects signals showing similar toggle patterns. Since the similarity is measured as the Euclidean distance between two signal vectors, this problem is identical to a *clustering problem*.

There are several challenges in signal clustering. First, exact clustering is known as an NP-hard problem, and thus, a randomized algorithm such as `k-means` should be used, where clustering results are different with different initial seeds. Moreover, with a non-trivial hardware design, there are a large number of signals, each of which is represented as a very high-dimensional point, which makes clustering more challenging. Specifically, if we have signal traces of N cycles with window size w, the dimension of each signal is N/w, which can be easily a very large number with a long trace.

Spectral clustering is a class of algorithms for clustering of high-dimensional data points through dimensionality reduction [23]. In this paper, we reduce the dimension of data by projecting data into principal components from singular vector decomposition (SVD). The algorithm to partition the data set into k clusters is as follows:

1. Find the space V spanned by the top k right singular vectors from SVD.
2. Project the data points into V through matrix multiplication.
3. Cluster the projected points through `k-means++` [6], which selects better initial seeds than random initial centroids.

4. Repeat multiple times and select the clustering with the best score.

It is also proven that the projection brings the data points closer to their cluster centers [1]. In addition, this algorithm can be efficiently implemented with high-performance linear algebra libraries.

Once signal clustering is done, *the signals that are the closest to the center of each cluster are selected*, which will be regression variables in power model training. The rationale is these signals have the smallest variance of similarities to other points in the same cluster, and thus, we expect them to introduce the smallest errors in regression than any other signals.

6.3.4 Finding the Optimal Number of Signals

The clustering algorithm in Section 6.3.3 finds the optimal clustering when the number of signals is given, but does not determine the number of clusters. In many cases, it is hard to know in advance what is the optimal number of signals for power modeling with an arbitrary hardware design. We want to select as many signals as possible for accurate power modeling, but not too many signals to avoid model overfitting and to enable power model instrumentation (Section 6.4).

Finding the optimal number of signals is the same as a *model selection problem*. We want to select the best clustering among candidate models for a given data. The idea is we run the clustering algorithm with different numbers of clusters and find the one having the best objective score.

For model selection, we use the Bayesian Information Criterion (BIC) [116], which is commonly used beyond the hypothesis tests. The BIC is a penalized model-fit statistic as it prefers a model having less parameters to a model having more parameters but only fitting marginally better. [2] Formally, for model M_j, the BIC is formulated as follows:

$$BIC_j = p_j ln(n) - 2ln(L_j)$$

where n is the number of points in the data, and p_j and L_j are the size and the likelihood of model M_j, respectively. The absolute value of the BIC is barely interpretable. However, the difference of values, $\Delta BIC = BIC_{new} - BIC_{old}$, is of interest. For example, we determine the new model is very strong compared to the old model if $\Delta BIC < -10$ [3].

For clustering, we use the formula derived by Pelleg and Moore [112], assuming underlying distributions are spherical Gaussians. The maximum likelihood estimate

[1] For the theorem and its proof, refer to [23]

[2] Compared to the Akaike information criterion (AIC), the BIC assigns more penalties on the number of parameters, having higher chances to reject models with more signals.

[3] The Bayes factor is equal to $\exp(-\Delta BIC/2)$

for the variance is:

$$\hat{\sigma}^2 = \frac{1}{n-k}\sum_{i=1}^{n}\|x_i - \mu(x_i)\|^2$$

where k is the number of clusters and $\mu(x_i)$ is the cluster center of x_i. Intuitively, this quantity explains how far points in each group are scattered away from their cluster center.

Then, the log likelihood of model M_j is the summation of log likelihoods of all clusters:

$$ln(L_j) = -\frac{n}{2}ln(2\pi) - \frac{nd}{2}ln(\hat{\sigma}^2) - \frac{n-k}{2} + \sum_{i=1}^{k} n_i ln(\frac{n_i}{n})$$

where d is the dimension of points and n_i is the number of points in cluster i. The number of parameters, p_j, is $(k-1)+dk+1$ for $(k-1)$ cluster probabilities, k centroids of dimension d, and one variance estimate. Intuitively, this quantity expresses how well signals in each group are clustered around their cluster center. This metric is also used by SimPoint [121] to find the optimal clustering for program phases.

To select the optimal number of signals, we keep track of ΔBIC by increasing the number of clusters, k. To avoid getting stuck at local minima, we employ *simulated annealing* as follows:

1. Run the clustering algorithm with the initial k, and compute the BIC, which is the initial best clustering.
2. Increase k, and run the clustering algorithm and compute the BIC.
3. If $\Delta BIC = BIC_{cur} - BIC_{best} < -10$, update the best clustering, and go to 2.
4. Otherwise, decrease the temperature, T, and go to 2 with the probability of $exp(\frac{\Delta BIC}{T})$.

This algorithm starts with a high temperature, which gradually decreases over iterations. Therefore, this algorithm is not likely to terminate in early iterations even if no better clustering is found, helping escape from local minima. However, with low temperatures, the algorithm has a very high probability to terminate in later iterations as the current best clustering is very likely to be the global optimum.

6.3.5 Obtaining Cycle-Accurate Power Traces

For accurate power modeling for RTL designs, cycle-accurate power traces are necessary. We obtain these power traces using commercial CAD tools as shown in Figure 6.1. We first obtain the gate-level design from target RTL using logic synthesis[4]

[4]We used Synopsys Design Compiler version O-2018.06-SP4

with a target technology library[5]. Clock gating is also automatically inferred during synthesis. Since commercial SRAM compilers were not available, we characterize SRAMs used in the target design with CACTI 6.5 [103], and generate library files using commercial library compilers[6]. To obtain accurate estimates for the timing, area, and the floorplan of the final silicon, we also place and route the post-synthesis design[7].

After place-and-route, the commercial power analysis tool[8] can compute cycle-accurate power traces from RTL VCD dumps. In this detailed power analysis, RTL signal activities are propagated into gate-level signals and the cycle-by-cycle power for modules in the design is computed. Since the full cycle-by-cycle power traces are generated instead of just the average power, this process tends to be the bottleneck for power modeling. For example, it takes a couple of days to obtain the training and test power data for the target designs used in this chapter.

Throughout this chapter, we assume the cycle-accurate power traces obtained in this section are the true power for training and evaluation.

6.3.6 Power-Model Regression

Once n signals are automatically selected (Section 6.3.3), we train the power model in terms of these signals against the cycle-accurate power traces from commercial CAD tools (Section 6.3.5). As discussed in Section 6.3.1, power model training finds the effective capacitances for the signal activities. We are also interested in module-level power modeling for thermal analysis [122].

Formally, for each module k, we want to find a function f_k that accurately approximates the actual power dissipation in terms of signal activities:

$$p_{kj} \approx f_k(x_{1j}, x_{2j}, \cdots, x_{nj})$$

for all time window j, where p_{kj} and x_{ij} are the power consumption of module k and the toggle density of signal i in window j, respectively. The total power consumption of the target design in window j is just the sum of power consumptions in window j over all modules.

Power models need to be as simple as possible to minimize the computation overhead for runtime power and thermal analysis. On the other hand, power models need to be more complex than linear regression since, in general, power dissipation is not a linear function of the activities of the selected signals. To cope with non-linearity, we use linear regression with interactions and high-order terms. One justification is that, theoretically, a non-linear function can be approximated with its Taylor expansion

[5] We used the TSMC 45nm technology

[6] We used Synopsys Library Compiler version J-2014.09-SP4 and Synopsys Milkyway version J-2014.09-SP4

[7] We used Synopsys IC Compiler version O-2018.06-SP4

[8] We use Synopsys PrimeTimePX version O-2018.06-SP4

with polynomial terms. There is also a large amount of empirical evidence that linear regression with polynomial terms of manually selected events and signals is reasonably accurate for microprocessors [60, 20, 80, 127, 65, 15]. Lastly, these interactions and high-order terms can be viewed as an approximation to *hidden activities* that are not solely captured by the selected signals.

Therefore, we also assume power dissipation is a function of the following form:

$$\begin{aligned} p_{kj} = \alpha + & \beta_1 x_{1j} + \beta_2 x_{2j} + \cdots \beta_n x_{nj} + \\ & \beta_{11} x_{1j}^2 + \beta_{22} x_{2j}^2 + \cdots + \\ & \beta_{12} x_{1j} x_{2j} + \cdots + \beta_{123} x_{1j} x_{2j} x_{3j} + \cdots \end{aligned}$$

where α and β's are parameters to be trained. As there are an infinite number of terms in the Taylor expansion, we limit the order of terms to two. However, the number of terms still grows exponentially on the number of signals. For instance, if 50 signals are selected, there will be 2550 terms in the model. Linear regression with this many terms tends to be unstable and suffers from high variance, losing prediction accuracy.

Moreover, models with a large number of variables are less interpretable, as well as increasing the compute overhead. From the perspective of activity-based power modeling, each regression variable represents a certain activity in the design and its coefficient is its effective capacitance. However, all these activities are not equally important for power modeling across different modules. Indeed, we want to systematically select most of the single-order terms but only a small number of higher-order terms to correlate between signal activities and power consumption without prior knowledge of these signals.

The previous two issues can be viewed as a problem of *regularization* and *variable selection* in linear regression. Prediction accuracy can be improved by shrinking coefficients through regularization with penalized regression, which reduces the variance of coefficients while trading off the bias. Variable selection can further improve the prediction accuracy, preventing overfitting, as well as the interpretability.

In this chapter, we employ the elastic net [154] for both regularization and variable selection. The elastic net is penalized regression with a convex combination of the L1 and L2 penalties of coefficients. As a result, the elastic net behaves mostly like LASSO [133], while preserving the prediction power of ridge regression.

We assume the training data is standardized having the zero mean and the unit standard deviation before regression. Then, to find the optimal coefficients β with given power trace $\mathbf{p}$ and toggle densities $\mathbf{X}$, the elastic net solves the following optimization problem:

$$\min_{\beta} \left\{ \frac{1}{2n} ||\mathbf{p} - \mathbf{X}\beta||^2 + \lambda \left(\frac{1-\rho}{2} ||\beta||^2 + \rho ||\beta||_1 \right) \right\} \tag{6.1}$$

where ρ and λ are determined by K-fold cross-validation, a technique that splits the training data into K groups for both training and validation. We also restrict

that all coefficients are non-negative[9]. This optimization can be efficiently solved by coordinate descent [48]. Notice that ridge regression and LASSO are special instances of the elastic net when $\rho = 1$ and $\rho = 0$, respectively.

When we apply the elastic net for power modeling, many unimportant variables are desirably eliminated as shown in Section 6.6.2

6.3.7 Finding the Optimal Window Size

As alluded in Section 6.3.2, the window size plays an important role in quantifying similarities in the toggle pattern matrix. If the window size is too small, two very similar signals (e.g. the input and the output of shift registers) may have a long distance between them. On the other hand, if the window size is too large, two distinctive signals can have a very small distance. Therefore, the window size affects the number of selected signals and in turn prediction accuracies.

The optimal window size is dependent on the target design *and* the training data set. We also observed that the clustering algorithm tends to select more signals with a larger window size. This is because a shorter window size dramatically increases distances between points, and thus, having more clusters does not help improving the quality of clustering.

Indeed, manual selection of the window size is another challenge and requires many trials and errors. Instead, we propose automatic signal selection as follows:

1. For a given window size,
   (a) Select signals with clustering (Section 6.3.3 and 6.3.4).
   (b) Train a power model for each submodule (Section 6.3.6) and compute its BIC.
2. Select the window size that minimizes the total score of power models.

Note that Step 1 can be parallelized for different window sizes to minimize runtime overhead as trainings are independent of one another. For the score of each power model in Step 2, we use the BIC for linear regression:

$$BIC = \frac{\sum_i^N error_i^2}{\sigma^2} + ln(N) \cdot df$$

where $error_i$ is the error of each data point i, df is the degree of the freedom of the model, which is a function of λ in Equation (6.1), and N and σ^2 are the size and the variance of data, respectively. Intuitively, the BIC finds the model with small errors as well as a small number of variables.

[9] We do not have this constraint on uncore in our example target design, whose power is given by subtracting the sum of power of all other modules from the total power.

Computing exact *df* for the elastic net is computationally expensive. However, when λ in Equation (6.1) is small, which is the case when only a small number of variables are selected, *df* is very close to the degree of the freedom of LASSO, which is equal to the number of nonzero coefficients [155]. Therefore, we approximate *df* with the number of nonzero coefficients in the model when we compute the BIC.

Since we have multiple power models for each submodule in the design, we may want to select the new window over the old window if the geometric mean of all Bayes factors [72] of each model is greater than 1, which is expressed as follows:

$$\sqrt[K]{\prod_{k=1}^{K} \exp\left(\frac{-\Delta BIC^k}{2}\right)} = \exp\left(\frac{-\sum_{k=1}^{K} \Delta BIC^k}{2K}\right) > 1$$

$$\Leftrightarrow \sum_{k=1}^{K} \Delta BIC^k = \sum_{k=1}^{K} BIC^k_{new} - \sum_{k=1}^{K} BIC^k_{old} < 0$$

where K is the number of models and BIC^k is the BIC of model k. Therefore, we select the window size that minimizes the sum of all BICs of each model.

Section 6.6.3 presents evaluations on how window sizes affect the number of selected signals and the accuracy of power models.

6.4 Power Model Instrumentation

Once the power model is trained as in Section 6.3, the target RTL needs to be instrumented for runtime power analysis and evaluation. The target RTL design is automatically instrumented with the power model using custom transforms inserted in the FIRRTL compiler as shown in Figure 6.3. Section 6.4.1 shows how activity counters collecting toggle activities of the selected signals are automatically inserted into the target RTL by a custom compiler pass. Section 6.4.2 shows how runtime power traces are obtained from FPGA-accelerated RTL simulation for various case studies.

6.4.1 Activity Counter Insertion

As shown in Figure 6.3, activity counters are automatically inserted by the compiler pass using the information from the power model. Figure 6.4 shows components instrumented by the compiler pass to collect toggle activities of the selected signals.

For each selected signal, the HD unit is inserted to compute the Hamming distance between the value at the current cycle and the value at the previous cycle. For a single-bit signal, it is just an XOR gate. For a multi-bit bus, the HD unit computes XORs of individual bits and counts the number of 1's. If the selected signal is a wire, a shadow register that keeps the value at the previous cycle is also inserted, and

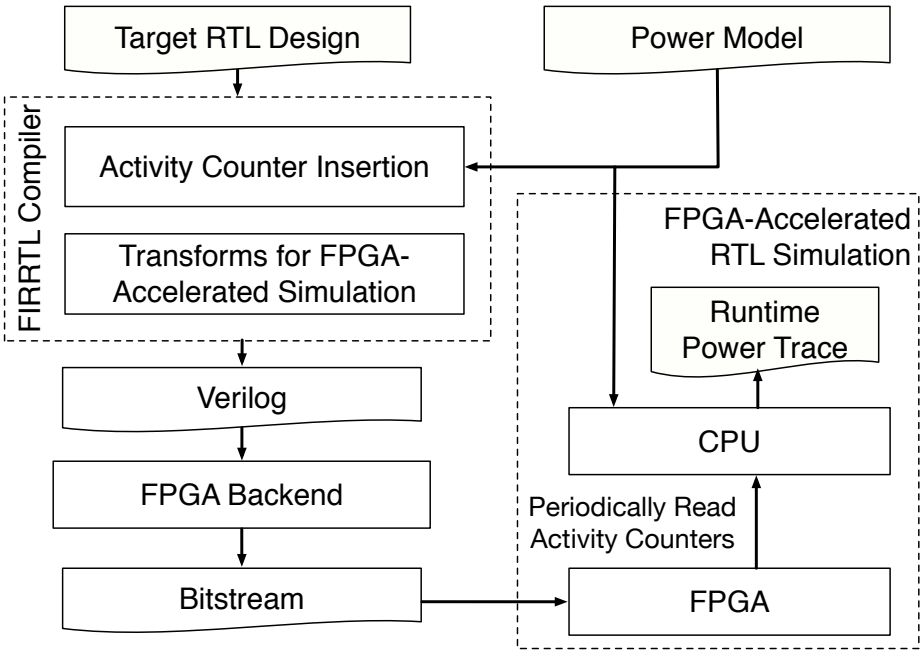


Figure 6.3: RTL instrumentation flow for runtime power analysis with FPGAs

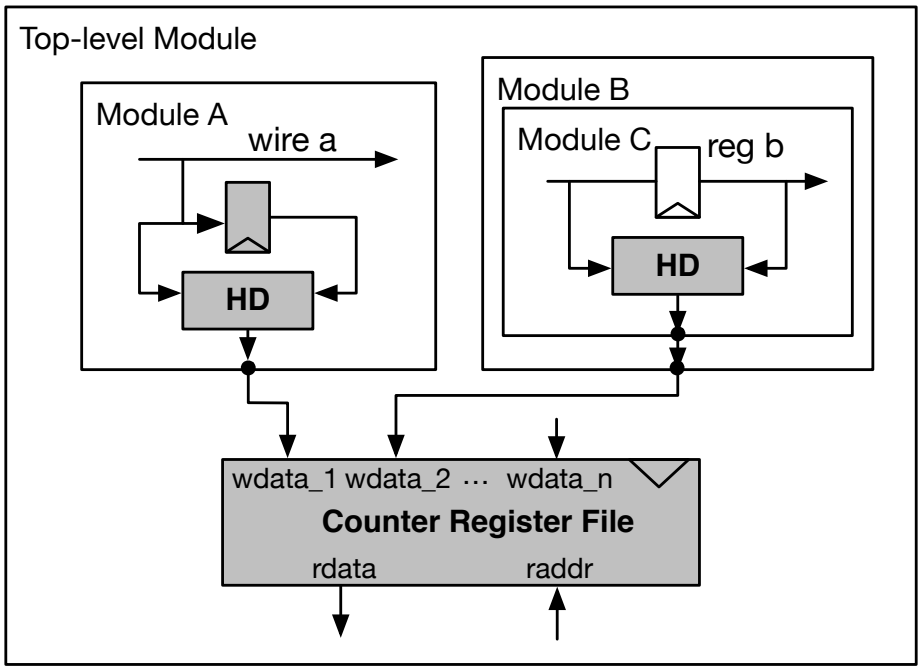


Figure 6.4: Activity counter instrumentation for runtime power analysis
Gray boxes are automatically instrumented by the compiler pass.

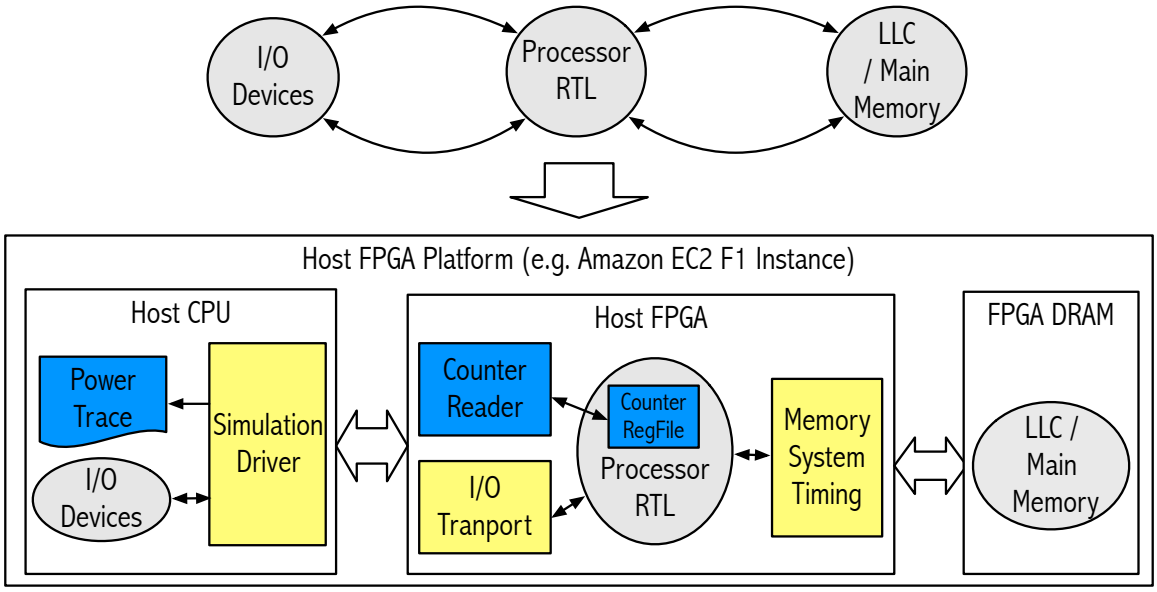


Figure 6.5: Mapping the target system to the host platform for runtime power analysis

then the input and the output of this shadow register are fed into the HD unit. On the other hand, if the selected signal is a register, we do not need a shadow register. Instead, the input and the output of the selected register are connected to the HD unit.

We also need counters that increment by the Hamming distance on each cycle. For this purpose, a counter register file is instantiated in the top-level module, with the number of write ports equal to the number of selected signals. The HD units in submodules are connected to the counter register file across different module hierarchies, and thus, the compile pass creates module ports along the connections to the write ports.

The counter register file has one read port and can be visible as architectural state for software running in the target design. Alternatively, this read port can be directly connected to the top-level I/O for FPGA-accelerated RTL simulation as explained in Section 6.4.2.

6.4.2 Runtime Power Analysis with FPGAs

FPGA-accelerated RTL simulation is the only viable way for performance, power, and energy evaluation of complex RTL designs running real-world software before tape-out. We extend MIDAS v1.0 (Chapter 3) to automatically generates FPGA-accelerated RTL simulators from any RTL designs with custom compiler passes. We use this framework to obtain runtime power traces from FPGAs. Once the activity counters are inserted, the instrumented target design is consumed by the following

custom transforms for FPGA-accelerated RTL simulation (Figure 6.3).

After the FPGA-accelerated RTL simulator is compiled into the bitstream, it is run on the FPGA along with the software simulation driver. Figure 6.5 shows how the transformed target design is mapped to the host platform. The processor RTL is mapped into the FPGA while the data for the last-level cache (LLC) and the DRAM are mapped into the FPGA DRAM. For the timing of the memory system, we have an abstract timing model that only keeps the tags of the LLC on the FPGA. The software driver with abstract I/O devices runs on the host CPU. The processor RTL infrequently communicates with the I/O devices through the I/O transport unit on the FPGA only when necessary (e.g. console I/O), minimizing simulation slowdown.

To obtain power traces, the simulation driver periodically reads the activity counter values through the activity counter unit on the FPGA, which is connected to the read port of the counter register file. When the counter values are read, the simulation is stalled so that it does not change the behavior of the target system. Counter sampling is infrequent, and thus, the simulation infrequently polls the counter read unit that only pauses the simulation when the counters are sampled.

After activity statistics are collected from the FPGA, the software driver, which has the power model information, does the rest of computations for model-level power and dumps runtime power values to a file. As such, we obtain the power traces over the whole execution of real-world applications at the end of the simulation.

We also estimate the power dissipation of the LLC and the DRAM with event counters in the memory timing model. For the LLC, we characterize its read and write energy per access as well as its static power with CACTI [103], and collect the number of read/write accesses to the LLC. For the DRAM, we assume Micron's LPDDR2 SDRAM S4 [101] and use the spreadsheet power calculator provided by Micron [100] with statistics on read/write operations and row activations of the DRAM.

6.5 Evaluation: Rocket and BOOM

6.5.1 Experimental Setup

The Simmani framework is first demonstrated with the Rocket in-order processor [8] and the BOOM out-of-order processor [34]. Table 6.1 shows their configurations with the power modeling results. We have abstract timing models for the L2 cache and the DRAM since we do not have corresponding RTL implementations for now. However, power dissipation of these regular array structures can be analytically modeled associated with their primary activities [103]. The cycle time, area, and the floorplan of each processor are obtain from Synopsys Design Compiler (logic synthesis) and Synopsys IC Compiler (place-and-route) with the TSMC 45nm technology.

For this evaluation, we empirically selected the window size for toggle matrices (Section 6.3.2) without automatic window selection. We trained power models with

Parameter	Rocket	BOOM
Fetch width	1	2
Decode width	1	1
Issue width	1	4
Issue slots	-	12
ROB entries	-	16
Ld/St entries	-	8/8
Physical registers	32(int)/32(fp)	56(int)/48(fp)
L1 I$ and L1 D$ capacities	32 KiB / 32 KiB	
L1 D$ MSHR entries	2	
ITLB and DTLB reaches	128 KiB / 128 KiB	
L2 TLB reaches	4 MiB	
L2 $ capacity and latency	1 MiB / 23 cycles	
DRAM latency	80 cycles	
Cycle time	1 ns	1 ns
Area	1 mm x 1 mm	1.2 mm x 1.2 mm
Total number of RTL signals	42494	64540
Window size for toggle matrices	80 cycles	140 cycles
Number of selected signals	47	56

Table 6.1: Parameters for Rocket and BOOM evaluated with Simmani

Microbenchmark	Small Input	Large Input
`median`	400 elements	10000 elements
`multiply`	100 elements	1000 elements
`vvadd`	300 elements	10000 elements
`towers`	7 disks	11 disks
`dhrystone`	500 runs	1000 runs
`qsort`	2048 elements	500 elements
`dgemm`	24x25x24 matrix	36x36x36 matrix
`spmv`	500x500 matrix with 2399 nonzero elements	2000x1000 matrix with 4666 nonzero elements

Table 6.2: Small and large inputs for microbenchmarks for evaluation

LASSO, a special case of the elastic net (Section 6.3.6).

For FPGA-accelerated RTL simulation, we use AWS F1 instances. Theses simulators are synthesized at the frequency of 90 MHz, but the simulation rate for SPEC benchmarks is 42.3 MHz on average due to the overhead of counter sampling for power analysis. For accurate validation, we carefully matched the timing of the memory system and the I/O devices between FPGA-accelerated RTL simulation and software RTL simulation.

The training data set consists of 1) the RISC-V ISA tests, 2) microbenchmarks with their small input sets, and 3) 30 random sample snapshots of 1024 cycles from each benchmark of SPECint2006 and SPECrate/speed 2017 Integer with their test input sets. The test data set consists of 1) microbenchmarks with their large input sets for validation of fine-grained runtime power prediction (Section 6.5.2), and 2) 50 random sample snapshots of 1024 cycles from selective benchmarks of SPECint2006 and SPECrate/speed 2017 Integer with their reference inputs for validation with realistic workloads (Section 6.5.3). Table 6.2 compares the small and large input sets for each microbenchmark.

6.5.2 Fine-Grained Power Prediction

In this section, we validate the capability of fine-grained power prediction. We ran microbenchmarks with their large input sets on the FPGA, sampled activity counter values every 128 cycles, and obtained power traces over the entire execution. These power traces were validated against cycle-accurate power traces from Synopsys PrimeTime PX (Section 6.3.5).

Figure 6.6 shows the normalized mean-squared errors (NMSREs) and the average errors (AVGEs) for Rocket and BOOM. For N samples, NRMSEs and AVGEs are calculated as follows:

$$NRMSE = \frac{\sqrt{\sum_i^N (p_i - p_i^{pred})^2 / N}}{p_{avg}}$$

$$AVGE = \frac{|p_{avg} - p_{avg}^{pred}|}{p_{avg}}$$

The NRMSE accounts for sample-by-sample errors while the AVGE cares the average values only.

Figure 6.7 highlights the power traces for `spmv` in which power modeling has large errors as shown in Figure 6.6. Large errors are caused by the fact that *the predicted power tends to be less variable than the actual power*. Note that this is expected due to regularization and variable selection (Section 6.3.6). To improve the accuracy, we can use more flexible models like the elastic net and employ automatic window selection (Section 6.3.7).

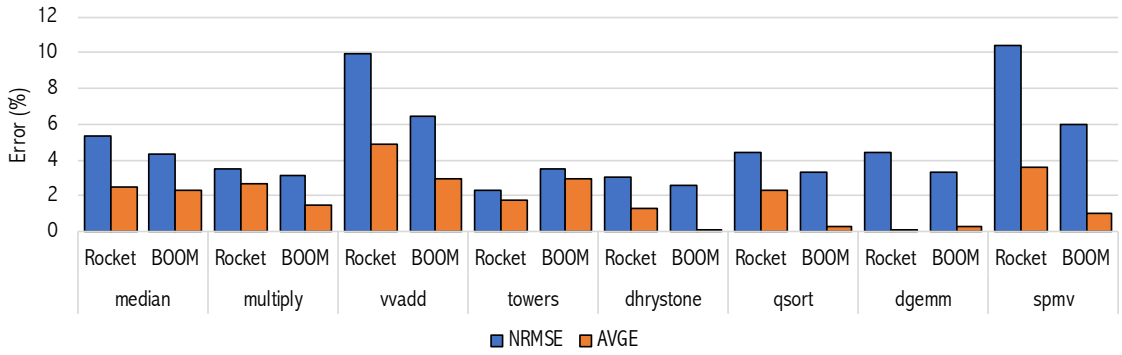


Figure 6.6: Fine-grained power prediction errors for microbenchmarks on Rocket and BOOM

6.5.3 Case Study: SPEC2006 and SPEC2017

We also validate the capability of power prediction for realistic workloads that execute trillions of cycles. Since we cannot obtain the cycle-accurate power traces over the entire execution from commercial CAD tools, the validation strategy is as follows.

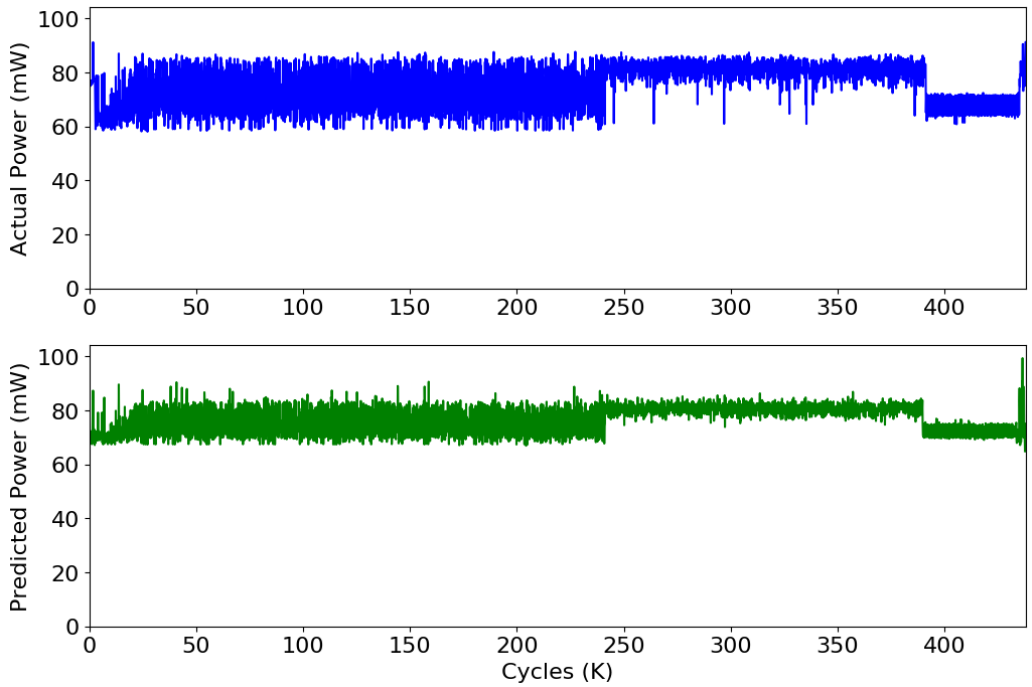
First, we take random 50 sample snapshots of 1024 cycles from each benchmark. When each of the random snapshots is taken, the simulation driver can also predict the power dissipation of this period of 1024 cycles from activity counters. As a result, we can obtain snapshot-by-snapshot power estimates with random snapshots as well as the full power traces at the end of the simulation.

Then, we obtain power estimates for each snapshot from commercial CAD tools through state replays. Using these power estimates and the power predictions from the FPGA, we can compute the NRMSE for 50 sample points. For the AVGE, we compare the average over the whole power trace against the average power from the replays with confidence intervals.

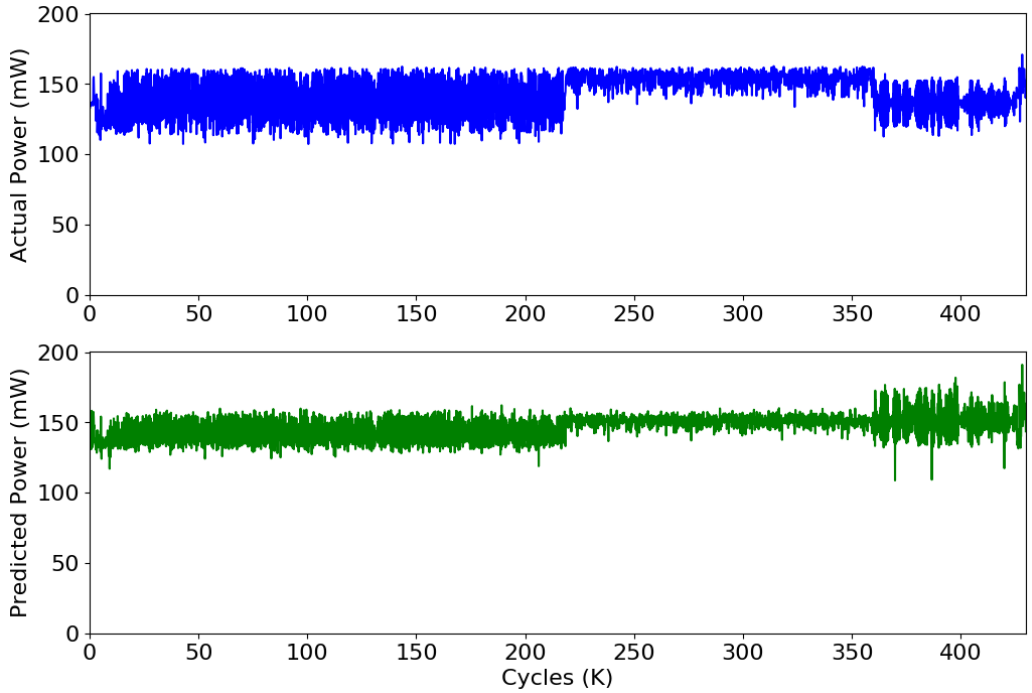
Figure 6.8 shows the prediction errors with the 95% confidence intervals across benchmarks in SPECint2006 and SPECrate/speed 2017 Integer. Unfortunately, we could not run `403.gcc` and `502.gcc` to completion on BOOM. In general, we have good prediction accuracy for both processors.

Figure 6.9 compares power breakdowns from the sample replays (Strober) and the power traces from the FPGA (Simmani). It seems that we have large errors when the processor utilization is low such as `429.mcf`. We believe this is because the training set only consists of workloads that highly utilize processors. Interestingly, Rocket has larger errors than BOOM even though it is a simpler processor. We believe this is because fewer signals are selected for Rocket and automatic window selection (Section 6.3.7) can mitigate such errors.

Figure 6.10 shows example power traces. We chose `600.perlbench` from SPEC-



(a) Rocket



(b) BOOM

Figure 6.7: Power traces for `spmv` on Rocket and BOOM

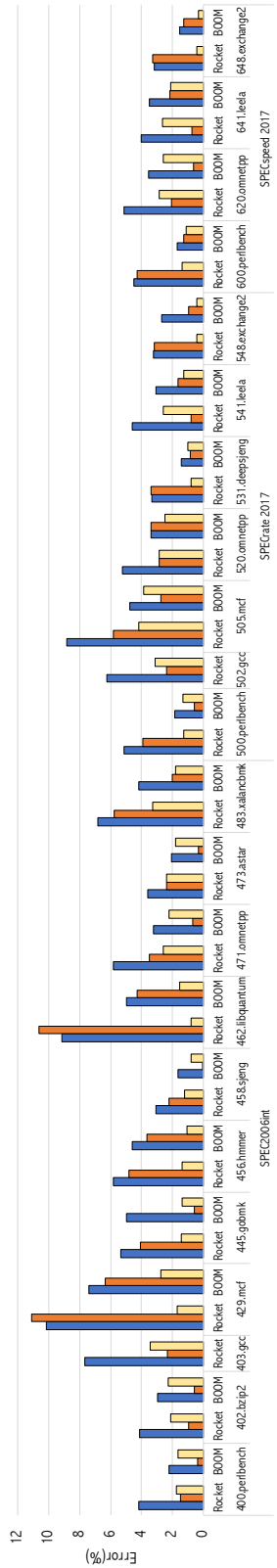


Figure 6.8: Prediction errors for the SPEC2006 and SPEC2017 integer benchmark suite

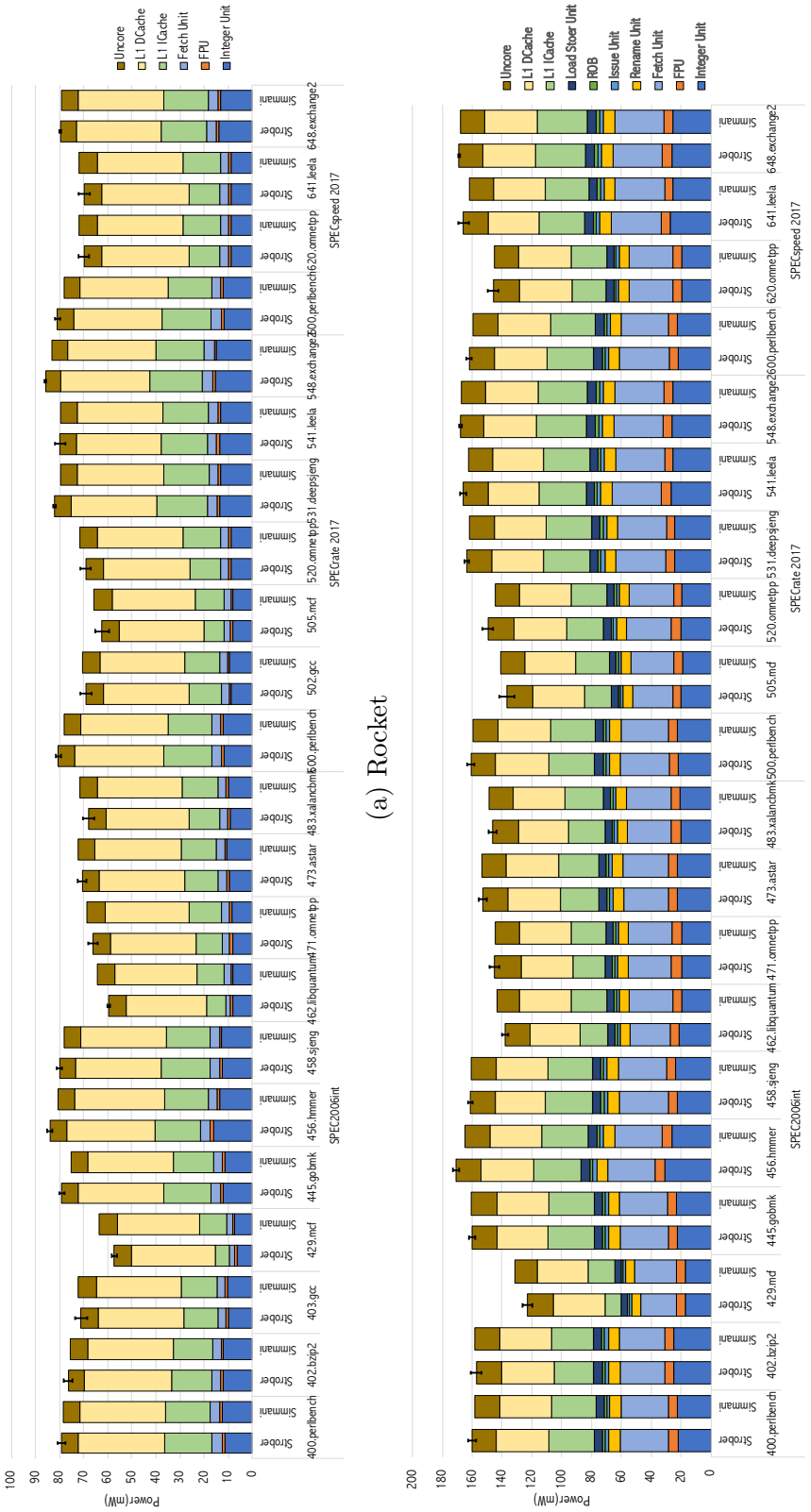
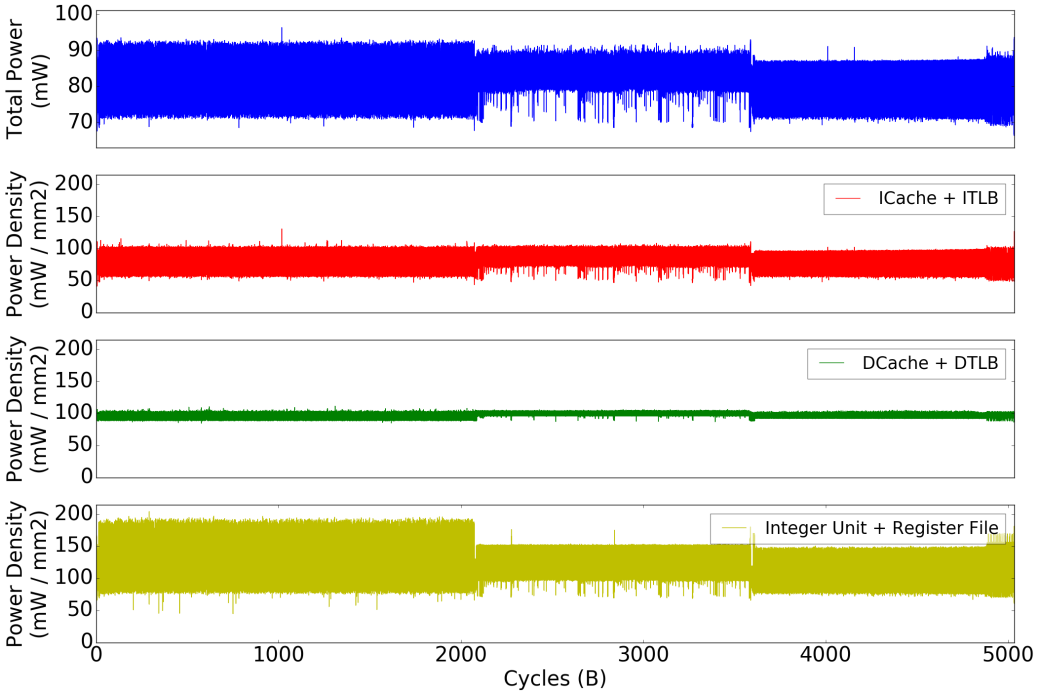
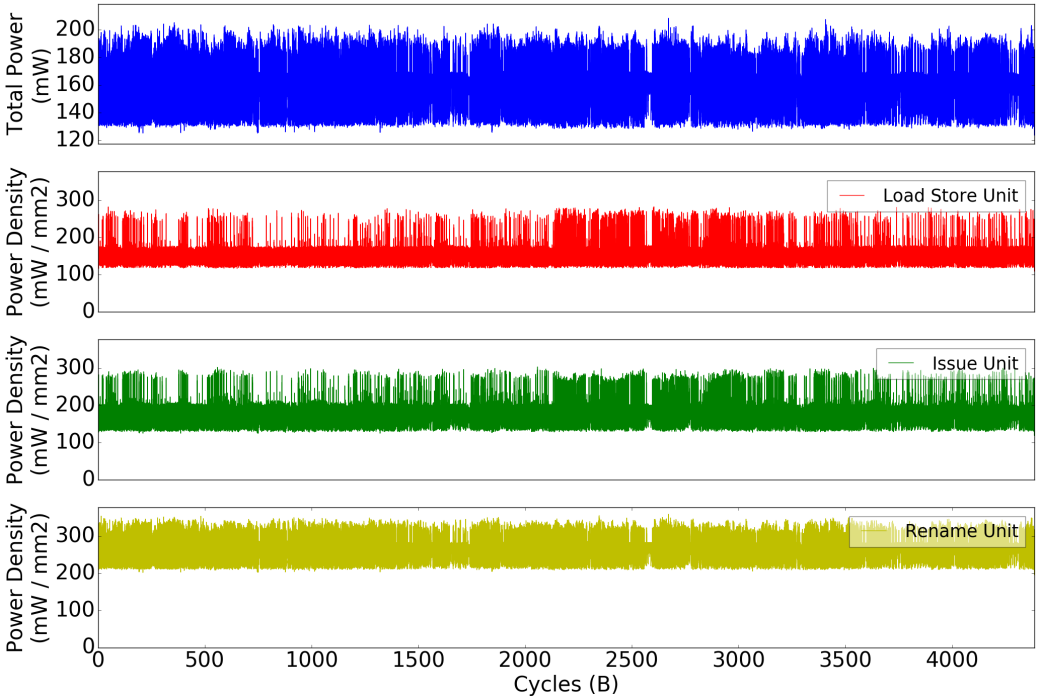


Figure 6.9: Power breakdowns for the SPEC2006 and SPEC 2017 integer benchmark suite



(a) `600.perlbench` running on Rocket



(b) `445.gobmk` running on BOOM

Figure 6.10: Power traces for `600.perlbench` on Rocket and `445.gobmk` on BOOM

Parameter	Rocket+Hwacha
Processor	Rocket 5-stage in-order processor
Accelerator	2048-bit wide vector unit and 64-bit wide scalar unit
Registers	32(int)/32(fp)/64(scalar)/256(vector)
L1 I and D $	Capacity: 32 KiB, Associativity: 8 ways
ITLB & DTLB	Reach: 128 KiB, Associativity: fully-associative
L2 TLB	Reach: 4 MiB, Associativity: direct-mapped
Cycle time	1 ns
Area	1.79 mm x 1.52 mm
L2 $	Capacity: 1 MiB, Latency: 23 cycles, Read energy: 116.1 pJ, Write energy: 95.9 pJ, Leakage power: 21.0 mW
DRAM	Latency: 80 cycles, Number of banks: 8, Number of rows in each bank: 16K, Open-page policy

Table 6.3: Parameters for Rocket+Hwacha evaluated with Simmani

speed 2017 Integer for Rocket and `456.gobmk` from SPECint2006 for BOOM. The topmost graphs show the total power over time, while the graphs below show the power densities of selected modules over time. We sampled activity counters every 100K cycles over trillions cycles of execution to obtain these traces.

6.6 Evaluation: Hwacha

6.6.1 Experimental Setup

Most of the power training algorithm (Section 6.3) is implemented in Python with the `SciPy`'s sparse matrix libraries, while the toggle matrix construction algorithm (Section 6.3.2) is implemented in C++. We import `k-means++` (Section 6.3.3) and the elastic net solver (Section 6.3.6) from `scikit-learn` [110].

The Simmani framework is demonstrated with the heterogenous processor composed of the Rocket in-order core[10] [8] and the Hwacha vector accellerator[11] [87, 85] (Rocket+Hwacha). Table 6.3 shows its configuration for the evaluation. We have abstract timing models for the L2 cache and the DRAM since we do not have corresponding RTL implementations for now.

The cycle time, area, and the floorplan of each processor are obtained from Synopsys Design Compiler (logic synthesis) and Synopsys IC Compiler (place-and-

[10] Commit: 50bb13d7887e5f9ca192431234b057ae9d8edb6c
[11] Commit: 519ed1642674909d89769eae1bd4fc35fa383e49

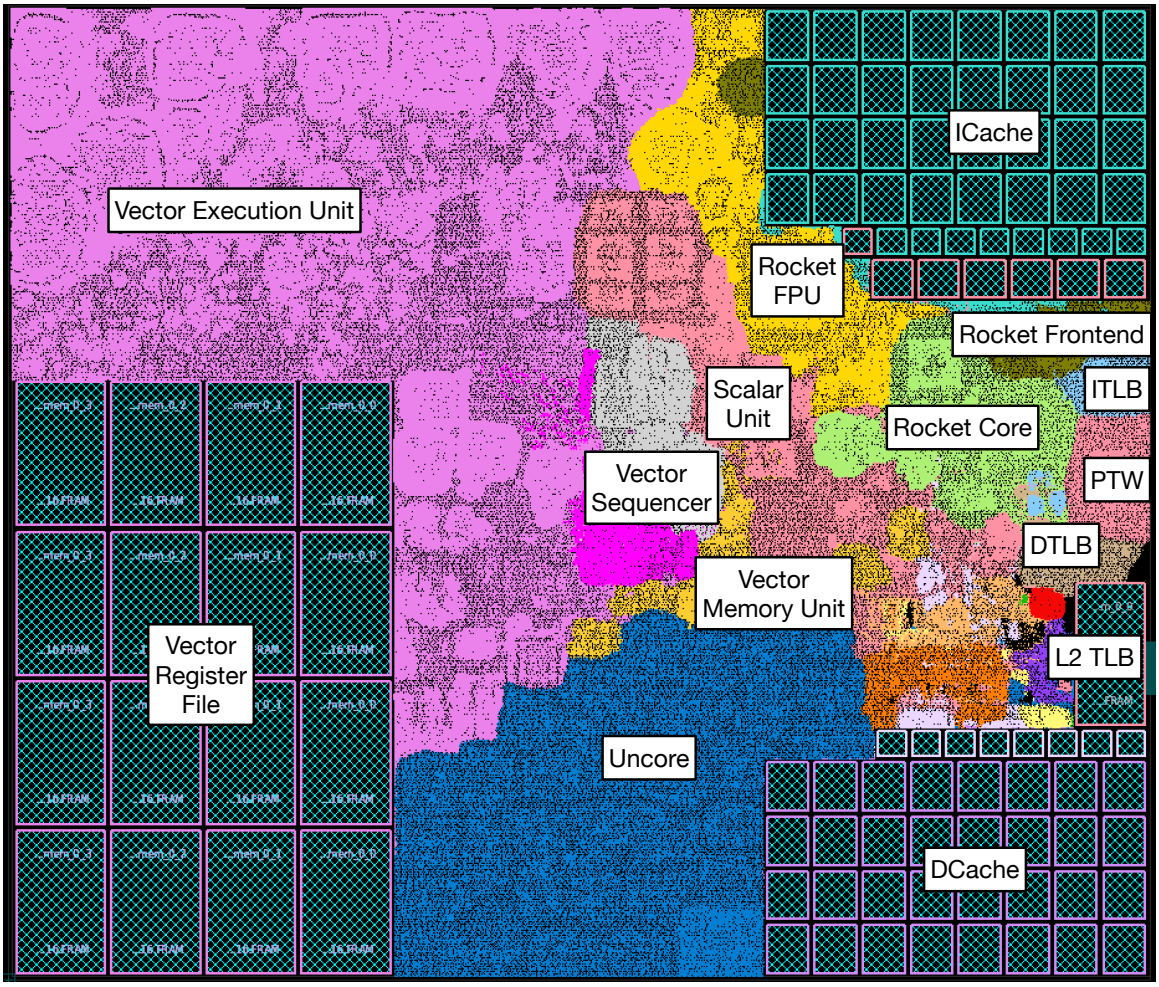


Figure 6.11: Floorplan of Rocket+Hwacha

Total number of RTL signals		115,285			
Module	**Selected Signals**	**Single-order Terms**	**Second-order Terms**	**Average Power (mW)**	**Standard Dev (mW)**
Total	113	72	599	143.22	14.40
Rocket Fetch Unit	18	14	123	3.84	1.19
Rocket Core	27	30	134	7.55	2.24
Rocket FPU	1	7	72	2.00	1.87
Scalar Unit	1	16	70	3.43	1.16
Vector Sequencer	17	13	16	4.69	2.80
Vector Register File	1	24	74	36.30	4.57
Vector FP MulAdd	7	14	41	1.76	3.74
Vector Execute Unit	10	27	114	24.48	3.25
Vector Memory Unit	-	13	22	2.24	0.48
L1 ICache & ITLB	8	15	79	16.38	6.31
L1 DCache & DTLB	19	12	72	9.76	4.69
Uncore	4	15	75	30.77	3.58

Table 6.4: Results of automatic signal and variable selection for Rocket+Hwacha

route) with the TSMC 45nm technology. Figure 6.11 shows the floorplan result of Rocket+Hwacha.

For FPGA-based simulation, we use AWS F1 instances. The FPGA-based simulator is synthesized at the frequency of 75 MHz, but the simulation rate for the case study was 39.3 MHz on average due to the overhead of counter sampling. For accurate validation, we carefully matched the timing of the memory system and the I/O devices between FPGA-based RTL simulation and software RTL simulation.

The training data set consists of 1) ISA tests, 2) microbenchmarks with their small input sets, and 3) 200 random sample snapshots from each benchmark of SqueezeNet with two images (dog and mousetrap).

For power model validation in Section 6.6.4, we used microbenchmarks with their large input sets. For case study in Section 6.6.5, we used different 11 images for each benchmark of SqueezeNet.

6.6.2 Signal and Variable Selection

Table 6.4 shows the results of automatic signal selection by the algorithm in Section 6.3.3 with the average power and the standard deviation of each module for the training set. We counted all signals and data busses shown in the VCD dump except intermediate signals generated by the FIRRTL compiler (starting with `_GEN_`). Our signal clustering algorithm selected 113 signals out of 115 thousand signals.

At glance, it is surprising only one signal was selected for the vector register file even though it dissipates a significant amount of power. If we had selected signals manually, the enable signals for this module would have been our primary choice.

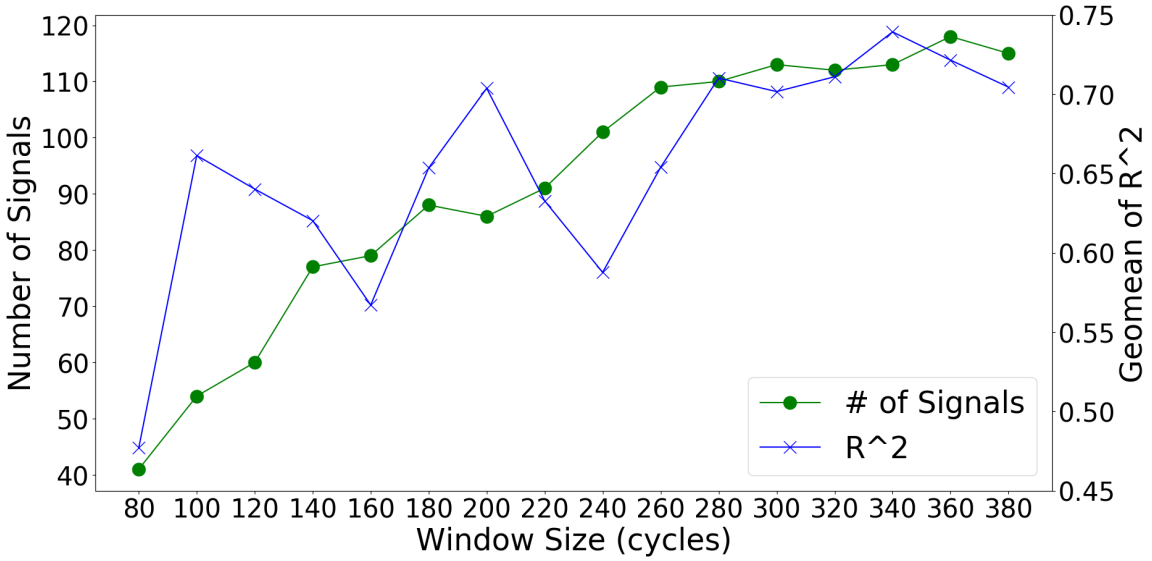


Figure 6.12: The number of selected signals and the geometric mean of R^2 across module-level power models for different window sizes

However, it turns out that those signals were clustered together with related signals in other modules but are not selected as representative signals. For example, its write enable and mask signals were clustered with a control signal of the floating point multiply-add unit, which is the representative signal of this cluster. Similarly, signals in the vector memory unit are clustered with signals in the load-store units of the vector execution unit, while signals in the FPU are clustered with signals in the Rocket core.

Table 6.4 also presents the variable selection results from power-model regression (Section 6.3.6). Note that some of terms appear across multiple modules, and thus, the total number of terms is smaller than the summation of the number of terms from each submodules. Our power modeling keeps 671 terms in total out of 6554 candidate terms for training.

We also notice that cross-order terms can capture events across different modules. For example, our power modeling finds the interaction between a predicate signal in the vector execute unit and a hit signal in the data cache, which has a positive effect on the power dissipation of the vector register file.

6.6.3 Automatic Window Size Selection

Figure 6.12 shows how the window size affects the number of selected signals and the geometric mean of the R^2 values, a statistic for how well the model fits the training data, across module-level power models. First of all, more signals are selected as the window size increases. This is because a bigger window size makes data points

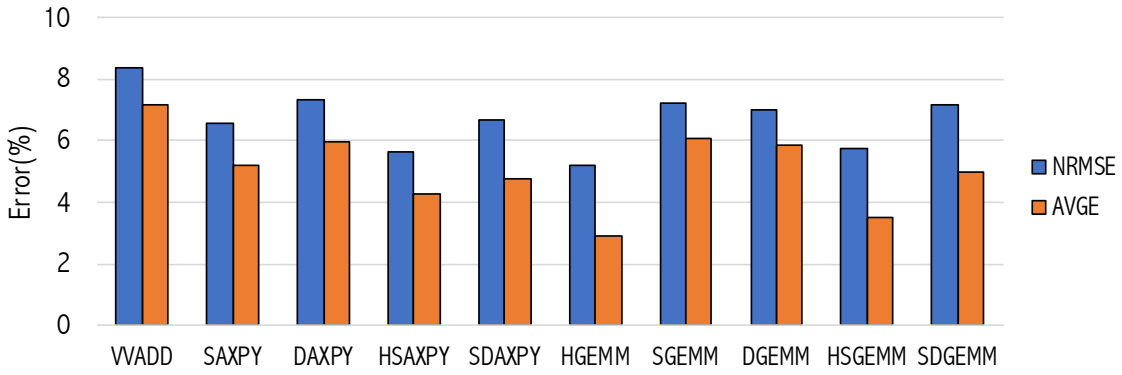


Figure 6.13: Power prediction errors for microbenchmarks on Rocket+Hwacha

closer, and therefore, having more clusters improves the quality of clustering. We can also see that this effect diminishes as the window size gets bigger.

Another trend is selecting more signals does not necessarily result in more accurate models. A model fits the training set well if its R^2 value is closer to 1.0[12]. As seen in Figure 6.12, the geometric mean of R^2 is the max at the window size of 340 cycles, and the sum of BICs, the score for window size selection in Section 6.3.7, is also the smallest at this point. Therefore, our training algorithm selects 113 signals with the window size of 340 cycles.

6.6.4 Power Model Validation

In this validation, we used well-known floating-point microbenchmarks vectorized for Rocket+Hwacha with various precisions. We estimated runtime power by sampling activity counters from the FPGA every 128 cycles, and compared it against power traces from Synopsys PrimeTime PX.

We computed the normalized mean-squared errors (NMSREs) and the average errors (AVGEs) across benchmarks. For N samples, NRMSEs and AVGEs are calculated as follows:

$$NRMSE = \frac{\sqrt{\sum_i^N (p_i - p_i^{pred})^2 / N}}{p_{avg}}$$

$$AVGE = \frac{|p_{avg} - p_{avg}^{pred}|}{p_{avg}}$$

The NRMSE accounts for sample-by-sample errors while the AVGE cares the average values only. The NRMSE also tends to be bigger than the AVGE.

[12]However, we do not use R^2 for model selection because a high R^2 may result from overfitting

Benchmark	Cycles per inference (B)		Speedup
	Scalar	Vector	
`SqueezeNet`	22.89	1.58	14.45
`SqueezeNet-8bits`	26.53	1.57	16.94
`SqueezeNet-Comp`	16.02	1.37	11.72

Table 6.5: Performance of Rocket+Hwacha for SqueezeNet

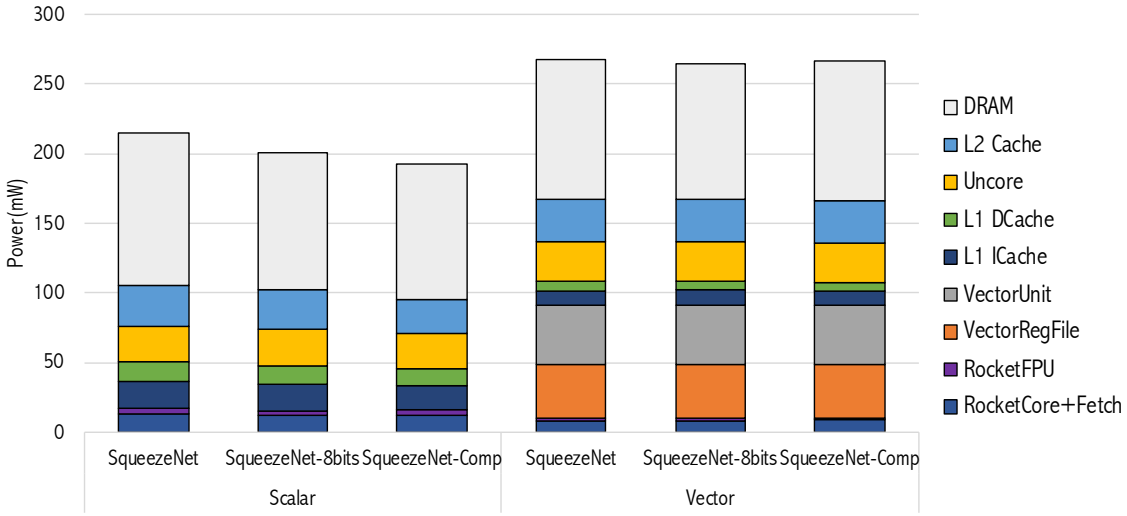


Figure 6.14: Power breakdown for SqueezeNet on Rocket+Hwacha

Figure 6.13 shows both errors are within 9 % and our power modeling is reasonably accurate for these microbenchmarks.

6.6.5 Case Study: SqueezeNet

In this section, we demonstrate how Simmani can be used for custom accelerators targeting emerging applications in the HW/SW co-design flow. For this case study, we use SqueezeNet [57], a neural network for image classification that achieves AlexNet-level accuracy with very small models. We evaluated three versions of SqueezeNet. In addition to the base variant (`SqueezeNet`), we evaluated a variant with 8-bit weight quantization (`SqueezeNet-8bits`), and a variant with both quantization and compressed weight storage (`SqueezeNet-Comp`).

We ported and vectorized SqueezeNet so that these three benchmarks can run on Rocket+Hwacha. For FPGA-based simulation, we made initramfs Linux images that contain SqueezeNet binaries as well as 11 images for inference. To obtain power traces, we sampled counters from the FPGA every 100K cycles. For comparison, we

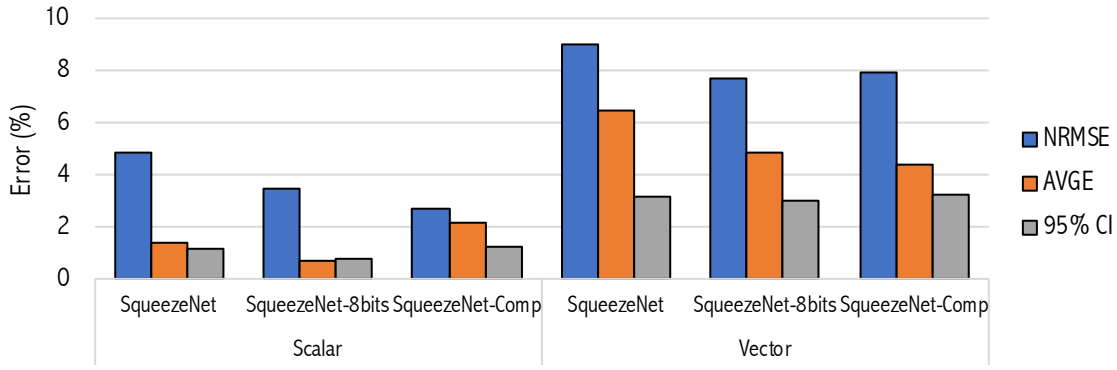
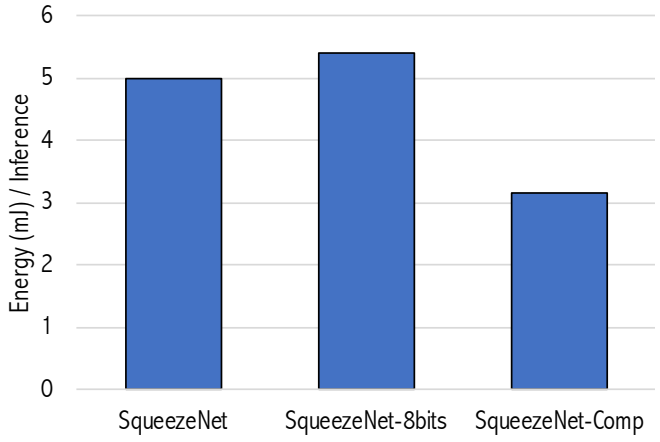
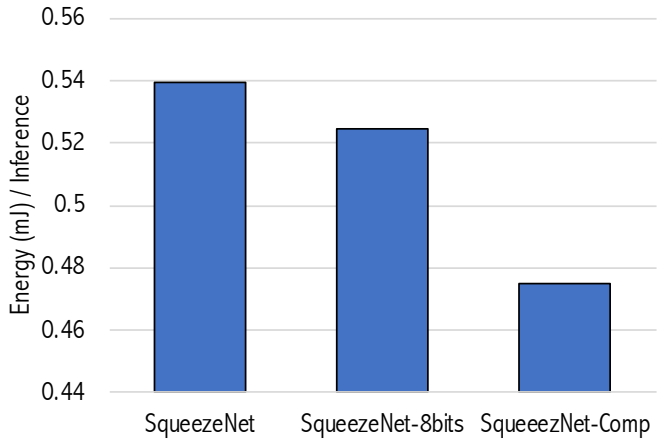


Figure 6.15: Power prediction errors for SqueezeNet on Rocket+Hwacha



(a) Scalar



(b) Vector

Figure 6.16: Energy efficiency of Rocket+Hwacha for SqueezeNet

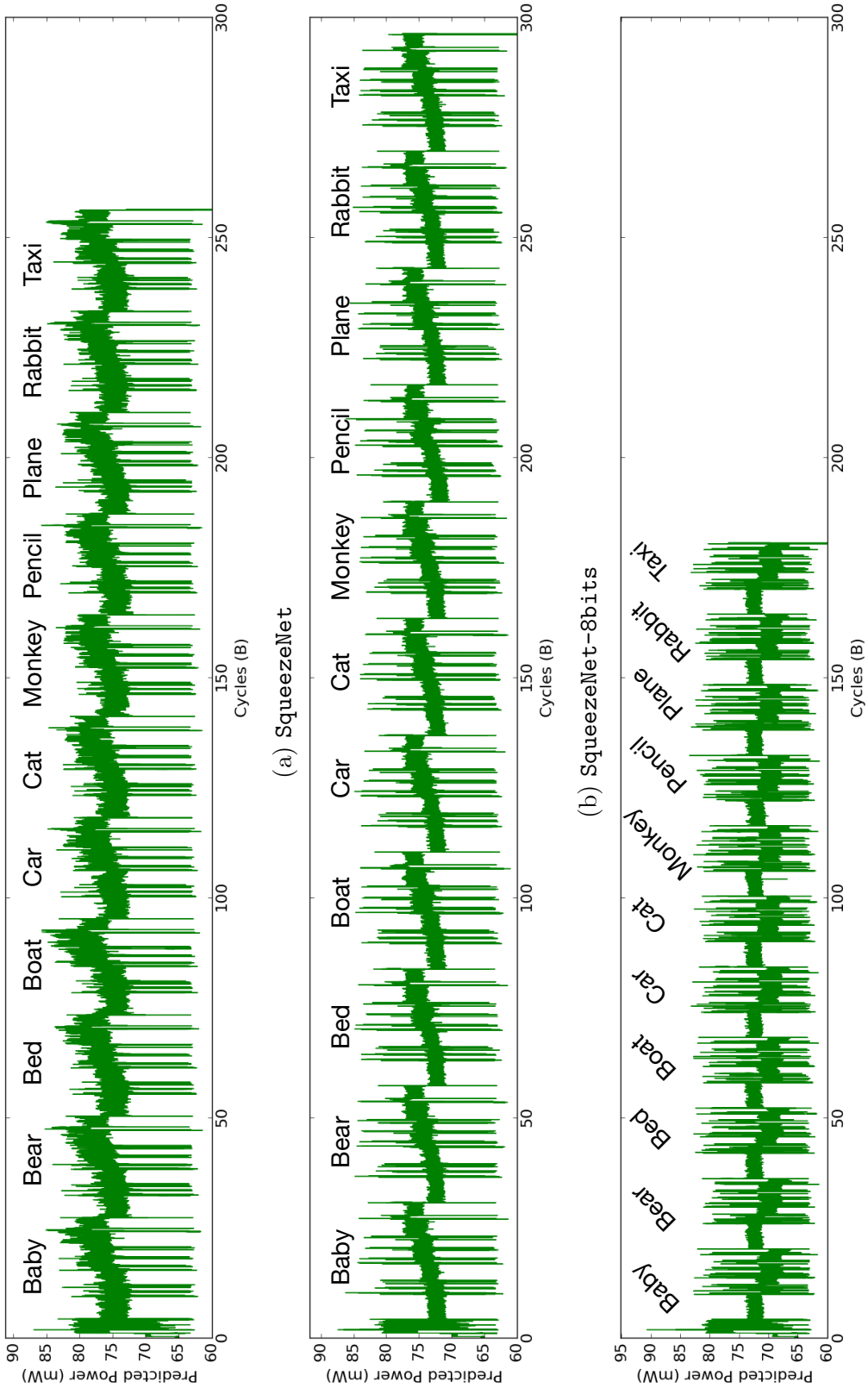
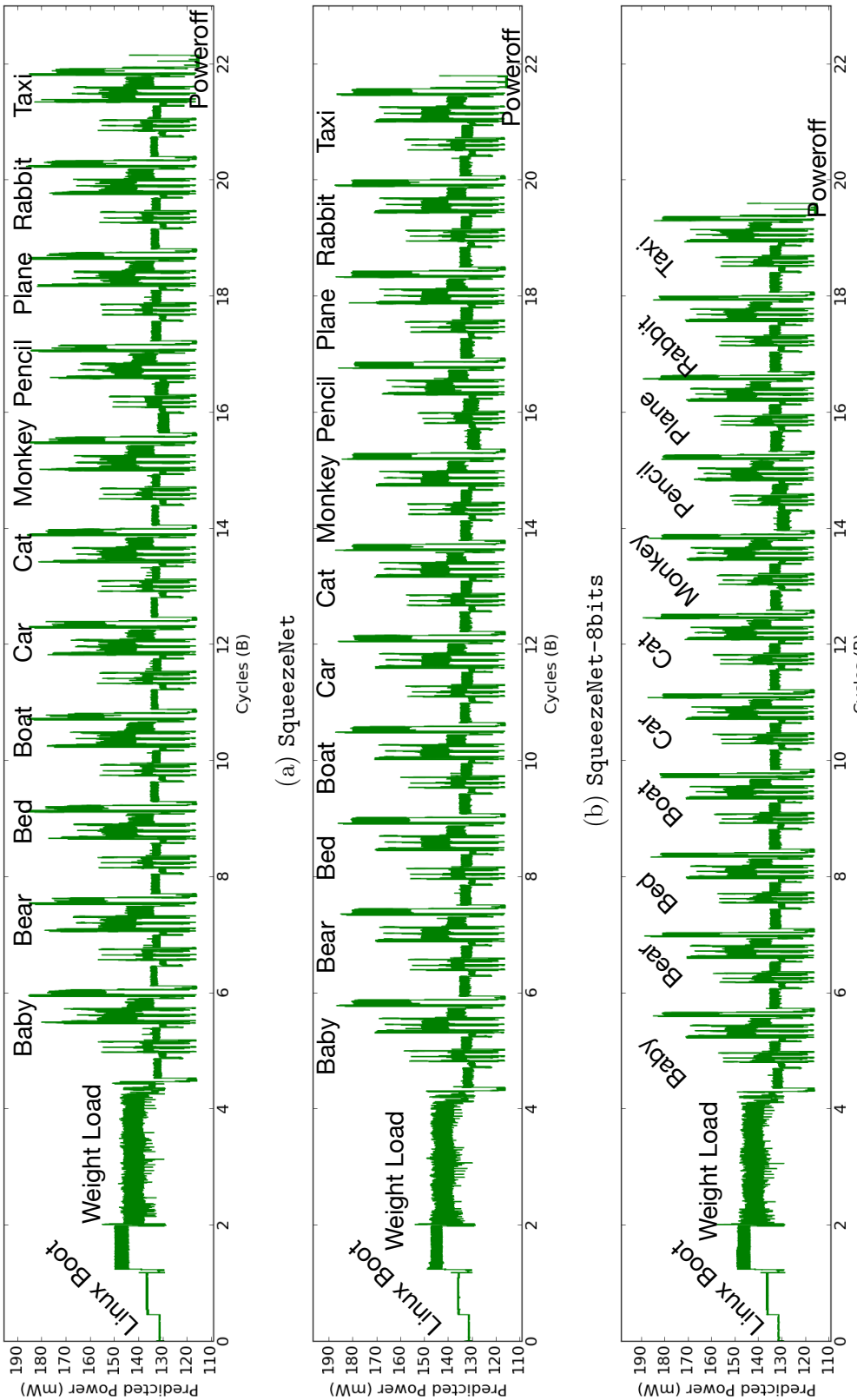


Figure 6.17: Power traces for the scalar SqueezeNet benchmarks on Rocket+Hwacha



(a) `SqueezeNet` (b) `SqueezeNet-8bits` (c) `SqueezeNet-Comp`

Figure 6.18: Power traces for the vectorized SqueezeNet benchmarks on Rocket+Hwacha

also evaluated unoptimized scalar SqueezeNet benchmarks, which do not utilize the vector unit at all.

Table 6.5 shows the performance of Rocket+Hwacha for these three benchmarks without and with vectorization. First of all, vectorization significantly improves its performance. Without vectorization, quantization decreases the performance, while with vectorization, it marginally improves the performance. However, in both cases, there is a significant performance improvement with compression on top of vectorization.

Figure 6.14 shows the average power breakdowns for SqueezeNet. For the scalar benchmarks, we assume the vector accelerator is perfectly power-gated. Without vectorization, both quantization and compression reduce power consumption as they require less memory accesses. Surprisingly, Simmani reveals that Rocket+Hwacha consumes almost the same power across the vectorized benchmarks. This is because the vector accelerator is highly utilized during inferences thanks to small model sizes.

Figure 6.15 shows the power prediction errors with the 95 % confidence intervals. To validate the power estimates in Figure 6.14, we took random 50 sample snapshots of 1024 cycles from each benchmark. When each of these random snapshots was taken, its runtime power was also estimated by sampling activity counters for this period of 1024 cycles. After the FPGA-based simulation was done, the power estimate of each sample point was obtained from sample replays, which was in turn compared against its runtime power estimate from the FPGA to compute the NRSE for 50 sample points. For the AVGE, we compared the average over the whole power trace against the average power estimate from sample replays, which also provided its confidence interval. From this validation, we can see that our power modeling also achieves good prediction accuracy for SqeeuzeNet.

Figure 6.16 shows the energy efficiency for SqueezNet. First of all, we can significantly improve energy efficiency with vectorization, which achieves significant speedups despite the increase in power consumption. Also, with vectorization, both quantization and compression gain energy efficiency as they require the same-level of power consumption.

Figure 6.17 and Figure 6.18 shows the entire power traces without L2 and DRAM power for the scalar and the vectorized SqueezeNet benchmarks, respectively, while booting Linux, loading the model weights, and running inferences for 11 images. We can also detect power phase changes over the whole execution of the benchmark, which will be useful for runtime power/energy managements. For example, we can observe more power variance in the later layers than in the earlier layers for each inference because the later layers are more memory-intensive.

As demonstrated in this case study, we believe Simmani will bring up future research opportunities on performance/power/energy optimizations for emerging applications with both software and hardware techniques.

6.7 Summary

In this chapter, we presented a novel runtime power modeling methodology for any RTL designs by automatically selecting key signals for power dissipation. Our observation was that RTL signals with similar toggle patterns have similar effects on power dissipation. We identified similar signals through clustering with high-dimensional points from the toggle pattern matrix, which is constructed from the VCD dumps of the training set. We selected representative signals from each cluster and constructed the power model with these signals through polynomial regression with regulation and variable selection to avoid overfitting.

We also automatically instrumented the target design with activity counters using custom compiler passes to collect statistics from FPGAs, without requiring manual effort. As such, Simmani enables various case studies for complex hardware designs running non-trivial applications by quickly providing their runtime power estimates.

We validated power modeling accuracy with a heterogeneous processor composed of an in-order core and a custom accelerator with microbenchmarks. We also demonstrated the capability of Simmani with a case study of HW/SW co-design for machine learning applications.

Simmani is truly general and easy-to-use for any RTL designs. We believe Simmani will provide more research opportunities on runtime performance, energy, and thermal optimizations with novel hardware designs in the HW/SW co-design process. Simmani will be open-source soon so that our methodology can be easily integrated into various accelerator research projects.

Chapter 7

Conclusion

This thesis demonstrated fast and accurate RTL simulation methodologies using FPGAs for performance, power, energy evaluation as well as verification and debugging in the hardware/software co-design flow. Section 7.1 summarizes the contributions of this thesis. Section 7.2 suggests potential future work based on the progress of this thesis.

7.1 Contributions

The contributions of this thesis are summarized as follows:

- **MIDAS v1.0** (Chapter 3): This thesis presented MIDAS v1.0, a open-source framework that automatically generates FPGA-accelerated performance simulators from any RTL designs. This framework is built upon custom compiler passes in the FIRRTL compiler, significantly improving productivity by minimizing manual effort. Performance simulators generated by MIDAS v1.0 are truly cycle-accurate and orders of magnitude faster than software-based simulators. Therefore, MIDAS v1.0 enables accurate performance evaluation of RTL designs by executing real-world workloads to completion.

  MIDAS v1.0 is also improved by a more realistic DRAM timing model hosted on the FPGA [18]. In addition, MIDAS v1.0 is extended for the datacenter simulator, FireSim [70], which connects RocketChip nodes with hardware network interface cards and software-based switch models deployed on the FPGA cloud.

- **DESSERT** (Chapter 4): This thesis demonstrated DESSERT, an effective framework for RTL debugging using FPGAs. The target RTL is automatically transformed and instrumented to enable deterministic simulation on the FPGA with state initialization and state snapshot capture. Assert statements, which are already present in target RTL for software simulation, are automatically

synthesized for quick error checking on the FPGA, while print statements are automatically synthesized for more exhaustive error checking from commit log comparisons between the golden model and the FPGA. To rapidly provide pre-error waveforms for effective debugging, two parallel deterministic simulations are run simultaneously to capture and replay state snapshots immediately before an error. DESSERT helps finding and fixing bugs in BOOM-v2, which only appear after billions of cycles up to a half trillion cycles.

- **Strober** (Chapter 5): This thesis presented Strober, a sample-based energy modeling for any RTL designs. Strober uses FPGAs for fast simulation of workloads, from which random RTL state snapshots as well as I/O traces are captured. RTL state snapshots are loaded into RTL/gate-level simulation for sample replays, during which input traces are fed into the input ports of the target design, while output traces are compared against the output ports of the design. The power analysis tool such as PrimeTime PX uses the signal activities from sample replays for average power estimation across the whole execution. This methodology achieves significant speedups over commercial CAD tools while providing accurate power and energy estimates with confidence intervals.

- **Simmani** (Chapter 6): This thesis lastly presented Simmani, an activity-based runtime power modeling by identifying key signals for power dissipation. Simmani is developed from the observation that signals showing similar toggle patterns have similar effect on power dissipation. Simmani selects key signals from signal clustering using the toggle pattern matrix constructed from VCD dumps of the training set. With these signals, Simmani constructs module-level power models with regression against power traces from CAD tools. Simmani also automatically instruments activity counters for the selected signals to collect statistics for runtime power analysis with FPGA-based simulation. Simmani is demonstrated with Hwacha running SqueezeNet as well as Rocket and BOOM running SPEC 2006/2017 integer benchmark suite. Power models trained by Simmani will be useful for thermal modeling and studies on dynamic power/thermal optimizations for custom accelerators.

7.2 Future Work

We believe there are lots of opportunities on future work based on the progress of this thesis. Therefore, this section alludes some ideas on promising future work.

First of all, there are a list of improvements that need to be done on top of MIDAS v1.0:

- **Multi-FPGA Simulation.** Today's high-end processors are too big to fit into a single FPGA for simulation, and thus, it is necessary to partition these

designs across multiple FPGAs. This partitioning should be done very carefully to minimize simulation performance degradation. Schelle et al. [114] and Assad et al. [7] demonstrate multi-FPGA emulation of industry processors. Likewise, we may also want to extend MIDAS v1.0 for multi-FPGA simulation, preserving all useful features supported by the current framework.

- **Multi-Clock Support.** MIDAS v1.0 only supports designs with a single-clock domain. However, real-world hardware designs are likely to have multiple clock domains having different voltages and clock frequencies. We may want to extend MIDAS v1.0 to support multi-clock domains. This new version needs to automatically detect different clock domains, transform each domain into a FAME1 model, and connect them with specialized channels. The operating frequency of each domain can be changed by writing control registers in these channels. This extension will also be useful for studies on fine-grained DVFS.

- **Auto FAME5 Transform.** FAME5 is a host multithreaded model on top of FAME1 [132]. This technique improves FPGA resource utilization by simulating a multi-core processor through multithreading with shared resources. We believe FAME5 models can be automatically generated from RTL by identifying shared resources across different modules and duplicating state elements to keep the context of each module. We believe this technique will enable more scalable simulation using a single FPGA.

Even though DESSERT helps debugging of single-core systems, we still need the following improvements:

- **Debugging Multi-Core Systems.** The difficulty of debugging multi-core systems is that memory operations across different cores can be arbitrary interleaved, and thus, values returned by loads are non-deterministic. To make it worse, real-world processors allow more relaxed memory models than sequential consistency, which makes it much more difficult to decide the correct value of a load.

  When we use DESSERT for multi-core systems, we should assume all values returned by loads are correct. In addition, we should check whether or not the memory consistency model is violated for a given instruction/memory trace. Unfortunately, verifying memory consistency is NP-complete unless the order of all memory operations to the same address is known [50]. We believe that the future version of DESSERT can support not only low-overhead but incomplete memory model checking with commit logs but also high-overhead but complete memory model checking with traces from memory bus monitors.

- **Bug Localization.** DESSERT can catch and replay errors that violate high-level properties. Even though we can obtain waveforms for these errors, it is

mentally tough to find their locations in the design by manually examining the waveforms. For example, pipeline hung is very easy to catch during simulation, but there are many possible reasons causing this error.

To alleviate this pain, we can develop a tool that pinpoints which part of the design is buggy from the error trace replayed by DESSERT. One promising approach is spec-mining for bug localization. First, fine-grained assertions are mined from correct traces generated by small tests and periodic replays from long simulation. Next, these fine-grained assertions are checked against the error trace to locate bug candidates that are signals exhibiting erroneous behaviors at the first time.

Finally, based on Strober and Simmani, we can do more exiting research with custom accelerators as follows:

- **Thermal Modeling and Analysis.** As we can obtain module-by-module power traces from FPGA-based simulation, as well as floorplans from CAD tools, we can conduct pre-silicon thermal analysis for novel hardware designs running real-world workloads. HotSpot [122] is one example framework for thermal analysis.

  HotSpot can be integrated into Simmani for runtime thermal analysis with FPGA-based simulation. Specifically, CAD tools dump the design's floorplan in the DEF format, which can be automatically transformed into grids compatible with HotSpot's grid model. As we can obtain each module's runtime power density from the power model and the floorplan, HotSpot can use this information to compute fine-grained thermal distribution in runtime, which can be also useful for studies on dynamic thermal optimizations.

- **Power Model Composition.** In this thesis, Simmani is demonstrated for a relatively smaller hardware design with a single tile compared to contemporary heterogeneous multi-core SoCs. In heterogeneous multi-core systems, cores(tiles) and uncore are fairly independent blocks, and thus, we can improve Simmani's scalability with *power model composition*: we can train individual power models for each core(tile) and uncore, and then compose the total power with statistical methods. Lee et al. [80] also present such a methodology.

- **Dynamic Power/Thermal Optimizations.** Runtime techniques for power and thermal management have been widely studied in the context of CPUs (e.g. [122, 61, 40, 41, 125, 46]). As custom accelerators are prevalent in computer systems, it is also important to do research on these techniques in the context of a variety of accelerators. As Simmani is generic for any hardware designs, we believe Simmani will be a useful tool for activity-based runtime power/thermal techniques for custom accelerators.

- **Auto Training Set Generation.** For Simmani, it is crucial to have a good training set for both signal selection and power model regression. We used ISA tests, microbenchmarks, and random samples from long-running applications using Strober as a training set. However, for some other novel hardware designs, it is even more challenging to find a good training set that is fully representative for their real-world applications.

  A good training set should have *good coverage of valid signal activities*. In fact, this challenge is also shared with input generation for simulation-based verification. Our future work will tackle this problem in a general setting for both hardware verification and power modeling. We believe workload generation with *coverage-based fuzzing* such as RFUZZ [76] is one promising approach.

Bibliography

[1] Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1.

[2] Cint2006. https://www.spec.org/cpu2006/CINT2006/.

[3] CoreMark EEMBC Benchmark. https://www.eembc.org/coremark/.

[4] Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[5] B. Alpern et al. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

[6] David Arthur and Sergei Vassilvitskii. K-Means++: the Advantages of Careful Seeding. In *ACM-SIAM Symposium on Discrete Algorithms*, 2007.

[7] Sameh Asaad, José Tierno, Ralph Bellofatto, Bernard Brezzo, Chuck Haymes, Mohit Kapur, Benjamin Parker, Thomas Roewer, Proshanta Saha, and Todd Takken. A cycle-accurate, cycle-reproducible multi-FPGA system for accelerating multi-core processor simulation. In *FPGA*, page 153. ACM Press, 2012.

[8] Krste Asanović, Rimas Avizienis, 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. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[9] David Atienza, Pablo G. Del Valle, Giacomo Paci, Francesco Poletti, Luca Benini, Giovanni De Micheli, and Jose M. Mendias. A fast HW/SW FPGA-based thermal emulation framework for multi-processor system-on-chip. In *DAC*, 2006.

[10] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Design Automation Conference (DAC)*, 2012.

[11] Somnath Banerjee and Tushar Gupta. Efficient online RTL debugging methodology for logic emulation systems. In *VLSI*, 2012.

[12] Somnath Banerjee and Tushar Gupta. Fast and scalable hybrid functional verification and debug with dynamically reconfigurable co-simulation. In *ICCAD*, 2012.

[13] Nathan Beckmann and Daniel Sanchez. Cache Calculus: Modeling Caches through Differential Equations. *IEEE Computer Architecture Letters*, 16(1):1–5, 2017.

[14] Frank Bellosa. The benefits of event: driven energy accounting in power-sensitive systems. In *The 9th ACM SIGOPS European Workshop*, 2000.

[15] R. Bertran, M. Gonzelez, X. Martorell, N. Navarro, and E. Ayguade. A Systematic Methodology to Generate Decomposable and Responsive Power Models for CMPs. *IEEE Transactions on Computers*, 62(7):1289–1302, Jul 2013.

[16] Kristof Beyls and Erik H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the International Conference on Parallel and Distributed Computing and Systems*, 2001.

[17] Abhishek Bhattacharjee, Gilberto Contreras, and Margaret Martonosi. Full-system chip multiprocessor power evaluations using FPGA-based emulation. In *ISLPED*, 2008.

[18] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, and Krste Asanović. FASED: FPGA-Accelerated Simulation and Evaluation of DRAM. In *Proceedings of the 27th ACM/SIGDA International Symposium on Field-Programmable Architectures*, FPGA '19, 2019.

[19] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[20] W. Lloyd Bircher and Lizy K. John. Complete System Power Estimation: A Trickle-Down Approach Based on Performance Events. In *ISPASS*, 2007.

[21] Stehpen M. Blackburn et al. The DaCapo benchmarks. In *OOPSLA*, 2006.

[22] Bluespec. RISC-V Verification Factory, 2017.

[23] Avrim Blum, John Hopcroft, and Ravindran Kannan. Foundations of data science. *Vorabversion eines Lehrbuchs*, 2016.

[24] Alessandro Bogliolo, Luca Benini, and Giovanni De Micheli. Regression-based RTL power modeling. *ACM Transactions on Design Automation of Electronic Systems*, 5, 2000.

[25] David Brier and Raj S. Mitra. Use of C/C++ models for architecture exploration and verification of DSPs. In *DAC*, 2006.

[26] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, 2000.

[27] Kevin Camera and Robert W. Brodersen. An integrated debugging environment for FPGA computing platforms. In *FPL*, 2008.

[28] Trevor E. Carlson, Wim Heirmant, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *SC*, 2011.

[29] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, 2016.

[30] Christopher Celio. The ccbench micro-benchmark collection. https://github.com/ucb-bar/ccbench/wiki.

[31] Christopher Celio. Speckle: A wrapper for the SPEC CPU2006 benchmark suite. https://github.com/ccelio/Speckle.git, 2014.

[32] Christopher Celio. *A Highly Productive Implementation of an Out-of-Order Processor Generator*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2018.

[33] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A. Patterson, and Krste Asanović. BOOMv2: an open-source out-of-order RISC-V core. In *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[34] Christopher Celio, David A. Patterson, and Krste Asanović. 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, Jun 2015.

[35] Debapriya Chatterjee, Anatoly Koyfman, Ronny Morad, Avi Ziv, and Valeria Bertacco. Checking Architectural Outputs Instruction-By-Instruction on Acceleration Platforms. In *DAC*, 2012.

[36] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007.

[37] Chin-Lung Chuang et al. Hybrid Approach to Faster Functional Verification with Full Visibility. *IEEE Design & Test of Computers*, 24(2):154–162, 2007.

[38] Eric S. Chung, Michael K. Papamichael, Eriko Nurvitadhi, James C. Hoe, Ken Mai, and Babak Falsafi. ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, 2(2):1–32, Jun 2009.

[39] Joel Coburn, Srivaths Ravi, and Anand Raghunathan. Power emulation: A new paradigm for power estimation. In *DAC*, 2005.

[40] Ryan Cochran, Can Hankendi, Ayse Coskun, and Sherief Reda. Pack & Cap: Adaptive DVFS and thread packing under power caps. In *MICRO*, number 1, pages 175–185. ACM, 2011.

[41] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch, and Ricardo Bianchini. CoScale: Coordinating CPU and memory system DVFS in server systems. *MICRO*, 2012.

[42] Christophe Dubach, Timothy M. Jones, and Michael F.P. O'Boyle. Microarchitectural design space exploration using an architecture-centric approach. In *MICRO*, 2007.

[43] Pradeep K. Dubey, George B. Adams, and Michael J. Flynn. Instruction Window Size Trade-Offs and Characterization of Program Parallelism. *IEEE Transactions on Computers*, 43(4):431–442, 1994.

[44] Brandon H. Dwiel, Niket K. Choudhary, and Eric Rotenberg. FPGA modeling of diverse superscalar processors. In *ISPASS*, 2012.

[45] Philip G. Emma and Edward S. Davidson. Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance. *IEEE Transactions on Computers*, C-36(7):859–875, 1987.

[46] Stijn Eyerman and Lieven Eeckhout. A counter architecture for online DVFS profitability estimation. *IEEE Transactions on Computers*, 59(11):1576–1583, 2010.

[47] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A performance counter architecture for computing accurate CPI components. In *ASPLOS*, 2006.

[48] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of Statistical Software*, 33(1):1–22, 2010.

[49] Mohammad Ali M.A. Ghodrat, Kanishka Lahiri, and Anand Raghunathan. Accelerating system-on-chip power analysis using hybrid power estimation. In *DAC*, 2007.

[50] Phillip B. Gibbons and Ephraim Korach. Testing Shared Memories. *SIAM Journal on Computing*, 26(4):1208–1244, 1997.

[51] Joseph L. Greathouse and Gabriel H. Loh. Machine learning for performance and power modeling of heterogeneous systems. In *ICCAD*, 2018.

[52] Subodh Gupta and Farid N. Najm. Power modeling for high-level power estimation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(1):18–29, 2000.

[53] Anthony Gutierrez, Joseph Pusdesris, Ronald G. Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D. Emmons, Mitchell Hayenga, and Nigel Paver. Sources of error in full-system simulation. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[54] A. Hartstein and Thomas R. Puzak. The optimum pipeline depth for a microprocessor. In *ISCA*, 2002.

[55] Mark D. Hill and Alan Jay Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[56] Ted Hong, Yanjing Li, Sung-Boem Park, Diana Mui, David Lin, Ziyad Abdel Kaleq, Nagib Hakim, Helia Naeimi, Donald S. Gardner, and Subhasish Mitra. QED: Quick Error Detection tests for effective post-silicon validation. In *IEEE International Test Conference*, pages 1–10, nov 2010.

[57] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. Technical report, 2016.

[58] Intel. SignalTap II Logic Analyzer: Introduction & Getting Started (ODSW1164), 2017.

[59] Engin Ïpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS*, 2006.

[60] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *MICRO*, 2003.

[61] Canturk Isci, Alper Buyuktosunoglu, Chen Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *MICRO*, pages 347–358, 2006.

[62] Canturk Isci and Margaret Martonpsi. Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques. In *HPCA*, pages 122–133, 2006.

[63] Yousef S. Iskander et al. Improved abstractions and turnaround time for FPGA design validation and debug. In *FPL*, 2011.

[64] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations. In *Proceedings of the 36th International Conference on Computer-Aided Design*, ICCAD '17, 2017.

[65] Hans Jacobson, Alper Buyuktosunoglu, Pradip Bose, Emrah Acar, and Richard Eickemeyer. Abstraction and microarchitecture scaling in early-stage power modeling. In *HPCA*, 2011.

[66] P. J. Joseph, Kapil Vaswani, and Matthew J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *MICRO*, 2006.

[67] P.J. Joseph, Kapil Vaswani, and Matthew J. Thazhuthaveetil. Construction and Use of Linear Regression Models for Processor Performance Analysis. In *HPCA*, 2006.

[68] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden,

Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*, 2017.

[69] Norman P. N.P. Jouppi. The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance. *IEEE Transactions on Computers*, 38(12):1645–1658, 1989.

[70] 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. FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, 2018.

[71] Tejas S. Karkhanis and James E. Smith. A First-Order Superscalar Processor Model. In *ISCA*, 2004.

[72] E Kass, R and E Raftery, A. Bayes factors. *Journal of the American Statistical Association*, 90(1995):773–795, 1995.

[73] Salman Khan, Polychronis Xekalakis, John Cavazos, and Marcelo Cintra. Using Predictive Modeling for Cross-Program Design Space Exploration in Multicore Systems. In *PACT*, number Pact, 2007.

[74] Hokeun Kim, David Broman, Edward A. Lee, Michael Zimmer, Aviral Shrivastava, and Junkwang Oh. A predictable and command-level priority-based DRAM controller for mixed-criticality systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[75] Dirk Koch et al. Efficient hardware checkpointing: concepts, overhead analysis, and implementation. In *FPGA*, 2007.

[76] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. Rfuzz: coverage-directed fuzz testing of rtl on fpgas. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018.

[77] Michael LeBeane, Jee Ho Ryoo, Reena Panda, and Lizy Kurian John. WattWatcher: Fine-Grained Power Estimation for Emerging Workloads. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2015.

[78] Benjamin C Lee and David M Brooks. Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction. In *ASPLOS*, 2006.

[79] Benjamin C Lee and David M Brooks. Illustrative Design Space Studies with Microarchitectural Regression Models. In *HPCA*, 2007.

[80] Benjamin C. Lee, Jamison Collins, Hong Wang, and David Brooks. CPR: Composable performance regression for scalable multiprocessor models. In *MICRO*, 2008.

[81] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[82] Jae W. Lee, Myron King, and Krste Asanović. Continual hashing for efficient fine-grain state inconsistency detection. In *ICCD*, 2007.

[83] Wooseok Lee, Youngchun Kim, Jee Ho Ryoo, Dam Sunwoo, Andreas Gerstlauer, and Lizy K John. PowerTrain: A learning-based calibration of McPAT power models. In *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2015.

[84] Yunsup Lee. *Decoupled Vector-Fetch Architecture with a Scalarizing Compiler*. PhD thesis, University of California, Berkeley, May 2016.

[85] Yunsup Lee, Albert Ou, Colin Schmidt, Sagar Karandikar, Howard Mao, and Krste Asanović. The hwacha microarchitecture manual, version 3.8.1. Technical Report UCB/EECS-2015-263, EECS Department, University of California, Berkeley, Dec 2015.

[86] Yunsup Lee, Colin Schmidt, Sagar Karandikar, Daniel Dabbelt, Albert Ou, and Krste Asanović. Hwacha preliminary evaluation results, version 3.8.1. Technical Report UCB/EECS-2015-264, EECS Department, University of California, Berkeley, Dec 2015.

[87] Yunsup Lee, Colin Schmidt, Albert Ou, Andrew Waterman, and Krste Asanović. The hwacha vector-fetch architecture manual, version 3.8.1. Technical Report UCB/EECS-2015-262, EECS Department, University of California, Berkeley, Dec 2015.

[88] Yunsup Lee, Andrew Waterman, Rimas Avizienis, Henry Cook, Chen Sun, Vladimir Stojanovic, and Krste Asanović. A 45nm 1.3 GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators. In *The 40th European Solid State Circuits Conference (ESSCIRC)*, 2014.

[89] Charles E. Leiserson and James B. Saxe. Optimizing synchronous systems. In *22nd Annual Symposium on Foundations of Computer Science*, 1981.

[90] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. GPUWattch: enabling energy optimizations in GPGPUs. In *ISCA*, 2013.

[91] Patrick S. Li, Adam M. Izraelevitz, and Jonathan Bachrach. Specification for the firrtl language. Technical Report UCB/EECS-2016-9, EECS Department, University of California, Berkeley, Feb 2016.

[92] Sheng Li, Jung Ho Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[93] Tao Li and Lizy Kurian John. Run-time modeling and estimation of operating system power consumption. In *SIGMETRICS*, 2003.

[94] Derek Lockhart, Gary Zibrat, and Christopher Batten. PyMTL : A Unified Framework for Vertically Integrated Computer Architecture Research. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[95] J. Marantz. Enhanced visibility and performance in functional verification by reconstruction. In *DAC*, 1998.

[96] Michael McKeown, Alexey Lavrov, Mohammad Shahrad, Paul J. Jackson, Yaosheng Fu, Jonathan Balkind, Tri M. Nguyen, Katie Lim, Yanqi Zhou, and David Wentzlaff. Power and Energy Characterization of an Open Source 25-Core Manycore Processor. In *HPCA*, 2018.

[97] Huzefa Mehta, Robert Michael Owens, and Mary Jane Irwin. Energy characterization based on clustering. In *DAC*, 1996.

[98] Francisco Javier Mesa-Martinez, Joseph Nayfach-Battilana, and Jose Renau. Power model validation through thermal measurements. 2007.

[99] Pierre Michaud, Andre Seznec, and Stephan Jourdan. Exploring Instruction-Fetch Bandwidth Requirement in Wide-Issue Superscalar Processors. In *PACT*, 1999.

[100] Micron Technology. Mobile LPDDR2 system-power calculator. https://www.micron.com/support/tools-and-utilities/power-calc.

[101] Micron Technology. Micron mobile LPDDR2 SDRAM s4. Datasheet, Micron Technology, Mar 2012.

[102] Jason E Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA*, 2010.

[103] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. CACTI 6.0 : A Tool to Model Large Caches. Technical Report HPL-2009-85, 2009.

[104] F.N. Najm. A survey of power estimation techniques in VLSI circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):446–455, dec 1994.

[105] R. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings of the Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'04)*, 2004.

[106] Derek B. Noonburg and John P. Shen. Theoretical modeling of superscalar processor performance. In *MICRO*, 1994.

[107] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical report, Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[108] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *ISCA*, 1997.

[109] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSS: a full system simulator for multicore x86 CPUs. In *48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011.

[110] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[111] Michael Pellauer, Michael Adler, Michel Kinsy, Angshuman Parashar, and Joel Emer. HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing. In *HPCA*, 2011.

[112] D. Pelleg and A.W. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000.

[113] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *ISCA*, 2013.

[114] Graham Schelle, Jamison Collins, Ethan Schuchman, Perrry Wang, Xiang Zou, Gautham Chinya, Ralf Plate, Thorsten Mattner, Franz Olbrich, Per Hammarlund, Ronak Singhal, Jim Brayton, Sebastian Steibl, and Hong Wang. Intel nehalem processor core made FPGA synthesizable. In *FPGA*, 2010.

[115] Andrew G. Schmidt, Bin Huang, Ron Sass, and Matthew French. Checkpoint/restart and beyond: Resilient high performance computing with FPGAs. In *FCCM*, 2011.

[116] Gideon Schwarz. Estimating the Dimension of a Model. *The Annals of Statistics*, 6(2):461–464, mar 1978.

[117] Rathijit Sen and David A. Wood. Reuse-based online models for caches. In *SIGMETRICS*, 2013.

[118] Ofer Shacham, Sameh Galal, Sabarish Sankaranarayanan, Megan Wachs, John Brunhaver, Artem Vassiliev, Mark Horowitz, Andrew Danowitz, Wajahat Qadeer, and Stephen Richardson. Avoiding game over: Bringing design to the next level. In *Design Automation Conference (DAC)*, 2012.

[119] H Shafi, P J Bohrer, J Phelan, C A Rusu, and J L Peterson. Design and validation of a performance and power simulator for PowerPC systems. *IBM Journal of Research and Development*, 47(5.6):641–651, 2003.

[120] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *ISCA*, 2014.

[121] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, 2002.

[122] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-aware microarchitecture. In *ISCA*, 2003.

[123] Daniel J. Sorin, Vijay S. Pai, Sarita V. Adve, Mary K. Vernon, and David A. Wood. Analytic evaluation of shared-memory systems with ILP processors. In *ISCA*, 1998.

[124] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *ISCA*, 2002.

[125] Bo Su, Junli Gu, Li Shen, Wei Huang, Joseph L. Greathouse, and Zhiying Wang. PPEP: Online Performance, Power, and Energy Prediction Framework and DVFS Space Exploration. In *MICRO*, 2014.

[126] Chen Sun, Mark T Wade, Yunsup Lee, Jason S Orcutt, Luca Alloatti, Michael S Georgas, Andrew S Waterman, Jeffrey M Shainline, Rimas R Avizienis, Sen Lin, Benjamin R Moss, Rajesh Kumar, Fabio Pavanello, Amir H Atabaki, Henry M Cook, Albert J Ou, Jonathan C Leu, Yu-Hsin Chen, Krste Asanović, Rajeev J Ram, Miloš A Popović, and Vladimir M Stojanović. Single-chip microprocessor that communicates directly using light. *Nature*, 528:534, dec 2015.

[127] Dam Sunwoo, Gene Y. Wu, Nikhil a. Patil, and Derek Chiou. PrEsto: An FPGA-accelerated Power Estimation Methodology for Complex Systems. In *FPL*, 2010.

[128] Stuart Sutherland. *The Verilog PLI Handbook*. The International Series in Engineering and Computer Science. Kluwer Academic Publishers, 2nd edition, 2002.

[129] Tarek M. Taha and D. Scott Wills. An instruction throughput model of superscalar processors. *IEEE Transactions on Computers*, 57(3):389–403, 2008.

[130] Zhangxi Tan, Zhenghao Qian, Xi Chen, Krste Asanović, and David Patterson. DIABLO: A Warehouse-Scale Computer Network Simulator using FPGAs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, 2015.

[131] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors. In *Proceedings of the 47th Design Automation Conference (DAC '10)*, 2010.

[132] Zhangxi Tan, Andrew Waterman, Henry Cook, Sarah Bird, Krste Asanović, and David Patterson. A Case for FAME: FPGA Architecture Model Execution. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*, 2010.

[133] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[134] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *ISCA*, 2000.

[135] J.I. Villar, J. Juan, M.J. Bellido, J. Viejo, D. Guerrero, and J. Decaluwe. Python as a hardware description language: A case study. In *Southern Conference on Programmable Logic (SPL)*, 2011.

[136] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, Mar 1985.

[137] Andrew Waterman and Yunsup Lee. Spike, a RISC-V ISA Simulator, 2011.

[138] Andrew Waterman, Yunsup Lee, Krste Asanović, and David Patterson. The RISC-V Instruction Set Manual: User-Level ISA Version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley.

[139] Andrew Waterman, Yunsup Lee, Krste Asanović, and David Patterson. The RISC-V Instruction Set Manual: Privileged Architecture Version 1.9.1. Technical Report UCB/EECS-2016-161, 2016.

[140] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David Patterson, and Krste Asanović. The RISC-V Instruction Set Manual: Privileged Architecture Version 1.7. Technical Report EECS-2015-157, EECS Department, University of California, Berkeley.

[141] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanović. The RISC-V Instruction Set Manual: User-level ISA Version 2.1. Technical Report UCB/EECS-2016-118, 2016.

[142] J. Wawrzynek, D. Patterson, M. Oskin, S. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic. Ramp: Research accelerator for multiple processors. *IEEE Micro*, 27(2):46–57, March 2007.

[143] Thomas F T.F. Wenisch, R.E. Roland E R.E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and James C J.C. Hoe. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro*, 26(4):18–31, Jul 2006.

[144] T. Wheeler, P. Graham, B. Nelson, and B. Hutchings. Using Design-Level Scan to Improve FPGA Design Observability and Controllability for Functional Verification. In *FPL*, 2001.

[145] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65, apr 2009.

[146] Darryl L. Willick and D. L. Eager. An analytic model of multistage interconnection networks. In *SIGMETRICS*, 1990.

[147] Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. GPGPU performance and power estimation using machine learning. In *HPCA*, 2015.

[148] R.E. Wunderlich, T.F. Wenisch, B. Falsafi, and J.C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA '03)*, 2003.

[149] Sam Likun Xi, Hans Jacobson, Pradip Bose, Gu-Yeon Wei, and David Brooks. Quantifying sources of error in McPAT and potential impacts on architectural studies. In *HPCA*, 2015.

[150] Xilinx. ChipScope Pro and the Serial I/O Toolkit, 2017.

[151] Zan Yang et al. Si-emulation: system verification using simulation and emulation. In *International Test Conference*, 2000.

[152] Xinnian Zheng, Lizy K John, and Andreas Gerstlauer. Accurate phase-level cross-platform power and performance estimation. In *53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016.

[153] Davide Zoni, Luca Cremona, William Fornaciari, and Milano Dipartimento. PowerProbe : Run-time Power Modeling Through Automatic RTL Instrumentation. In *DATE*, 2018.

[154] Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society. Series B: Statistical Methodology*, 67(2):301–320, 2005.

[155] Hui Zou, Trevor Hastie, and Robert Tibshirani. On the "degrees of freedom" of the lasso. *Annals of Statistics*, 35(5):2173–2192, 2007.