ESSENT: A High-Performance RTL Simulator

Scott Beamer Thomas Nijssen Krishna Pandian Kyle Zhang
Computer Science & Engineering
University of California, Santa Cruz
Santa Cruz, CA, USA
{sbeamer, tnijssen, kpandian, kmzhang}@ucsc.edu

***Abstract*—RTL simulation is a critical tool for hardware development, debugging, design space exploration, and verification. The slow speed of hardware simulation can often be a bottleneck for the design process, so any speed improvements could either reduce design times or improve design quality. We overview ESSENT, a high-performance RTL simulator available as open-source. Not only does it contain novel optimizations that make it typically faster than other software RTL simulators, the codebase can serve as a foundation for simulation research.**

***Index Terms*—simulation, open source, agile hardware design**

I. Introduction

Simulation is an important tool in any hardware design flow. Although there are many types of simulation, cycle-accurate RTL simulation is the workhorse for hardware design, debugging, design space exploration, and verification. Many simulation methods are suitable for modest designs over small time periods. As the simulation is scaled up in both space (i.e. larger designs) and time (i.e. longer simulations), simulation efficiency becomes critical. In this work, we overview Essential Signal Simulation Enabled by Netlist Transformations (ESSENT), a high-performance RTL simulator [4]. It excels at simulation speed, and we are continuing to increase the scale at which its acceleration techniques are beneficial.

ESSENT pioneers novel optimizations to accelerate simulation and it is open source[1]. In this overview, we provide: a background on simulation, a survey of ESSENT's features, a brief performance demonstration of ESSENT, and discuss its applicability.

II. Simulation Background

In this work, we find it helpful to represent hardware designs as directed graphs. Each node represents a logic element or state element, and each edge represents a wire. Thus, a hardware design is a graph of elements (nodes) connected by wires (edges). Simulating a single clock cycle is thus an evaluation of the graph which consumes external inputs and the values of its internal state elements to produce external outputs and new values for its internal state elements. Longer simulations result in more graph evaluations.

Quickly, simulation times can become problematic, so there has been substantial prior work on accelerating simulation [6], [7], [10], [19], [20]. Accelerating software simulation boils down to reducing the work expended per simulated cycle. Some of the first simulators created use an *event-driven* approach, in which they dynamically schedule nodes to be evaluated as changes propagate through the graph (hardware design). If node evaluations are scheduled carelessly, it can result in a node being evaluated multiple times per simulated cycle, which is wasteful. A more efficient schedule is possible in which each node is evaluated at most once per simulated cycle, but it requires the design graph to be acyclic. With an acyclic graph, one can use an algorithm such as topological sort to produce an efficient schedule. To clarify, only combinational loops are problematic in terms of creating cycles in the design graph, as feedback paths through state elements are broken naturally by synchronizations via clock cycles. Combinational loops are typically small, and the strongly connected components that represents them can be collapsed into a supernodes, making the overall graph acyclic. *Levelization* is a practical method akin to breadth-first search (BFS) to order node evaluations to prevent unnecessary repeat evaluations [21], [23]. Even with a more efficient schedule, event-driven simulators expend a great deal of effort in overhead from scheduling. They are continuously creating events, prioritizing them, and choosing which event to evaluate next.

Full-cycle simulators obviate the scheduling overhead from

[1]https://github.com/ucsc-vama/essent

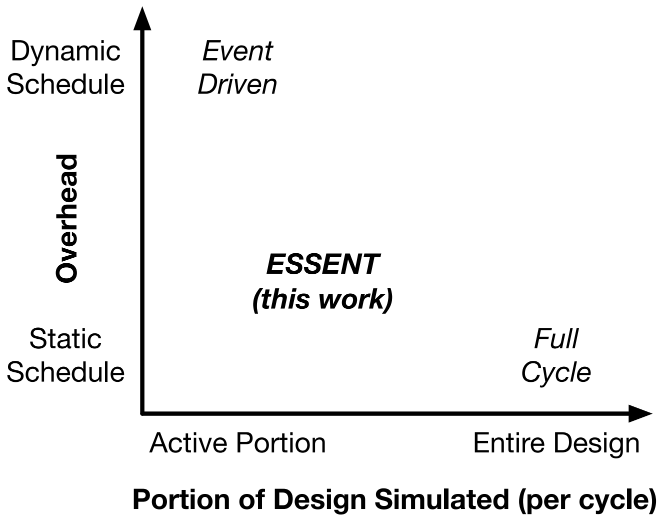

Fig. 1. ESSENT (this work) provides nearly the scheduling efficiency of full-cycle simulation while simulating close to only the active portion of the design (like event-driven simulation).

event-driven simulation by performing all scheduling statically in advance [5], [9], [20]. Precomputing the schedule once is possible since hardware design topologies do not change mid-simulation. A full-cycle simulator effectively inlines the entire design and turns it into straight-line code. This code can be compiled to make a simulator specialized for the target design. Executing that code will typically incur few bottlenecks in the host processor's backend, but it can often challenge the processor's frontend with poor temporal locality for instructions [3]. As designs get bigger, so does the simulator program binary and thus the reuse interval for a given instruction. The static schedule of a full-cycle simulator evaluates every node in the graph every simulated cycle as it is unaware of what has changed.

The prevailing simulation approaches (event-driven and full-cycle) present an interesting tradeoff (Figure 1). Event-driven simulators have tremendous scheduling overhead, but they only perform work on the active portion of the design. Full-cycle simulators eliminate the scheduling overhead, but they obliviously simulate the entire design. Even within "active" hardware components, most signals rarely change, so the inability to skip over inactive portions is a significant inefficiency. Many of the fastest simulators today use a full-cycle approach as typically removing the scheduling overhead is more beneficial than reducing the fraction of the design simulated [20]. Prior work attempts to create hybrids to reduce both scheduling overheads and the fraction of the design simulated. ESSENT is such a hybrid, and it improves on similar prior work in a number of ways.

III. ESSENT

ESSENT is an open-source RTL simulator generator [3], [4]. Given a design in the FIRRTL [14] format, it produces C++ code that can be compiled to produce a high-performance simulator for the design. It reduces scheduling overhead by performing the scheduling once statically at compile time. It reduces the fraction of the design simulated by dynamically skipping over inactive portions of the design. Its key research contributions are its low overhead techniques that enable it to skip over inactive portions of the design.

If the inputs to an inactive hardware component do not change, the outputs can be reused without re-evaluating the entire component. This ability to reuse is always true for acyclic combinational logic, but it can also be true for sequential logic with some caveats. Thus, instead of always evaluating a component, it is evaluated conditionally only if its inputs or internal state has changed. Performing activity detection and reuse on a node granularity is impractical as the overhead it introduces negates its benefit. Naturally, the solution is to coarsen the granularity at which the reuse is performed to amortize the overheads. Instead of checking the inputs to a single node (e.g. an adder), we check if the inputs to dozens or hundreds of components has changed. Coarsening the hardware design is equivalent to partitioning the design graph.

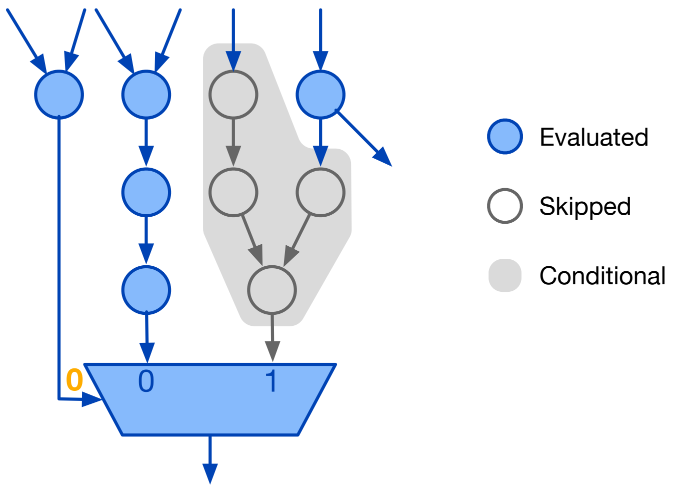

Fig. 2. Example use of *dynamic don't cares* for a multiplexer. ESSENT at compile-time recognizes the shaded portion feeding in way 1 as the maximum portion of the design that can be skipped if the other way (way 0) is selected.

To ensure an efficient schedule is possible in which each node is evaluated at most once per simulated cycle, the graph must be acyclic. Unfortunately, even if starting with an acyclic graph, most partitionings will be cyclic [11]. What is needed is an acyclic partitioning, and regrettably there has been far less research into acyclic graph partitioning than there has been into general graph partitioning [16]. The lack of a high-quality, easy-to-use acyclic partitioner hindered prior hybrid simulator attempts [7], [10], [17]. To deal with shortcomings of their partitioners, prior work had various coping strategies such as: replicating parts of the design to break cycles, re-evaluating components that are cyclic, and excluding state elements from partitioning. Additionally, sometimes the challenges of acyclic partitioning are exposed to the user in the form of: complicated design-specific parameters, forcing the user to write their own activity detection code, or forcing the user to partition the design themself.

ESSENT's key enabling technology is its novel acyclic partitioner. With it, it not only enjoys the performance benefit of conditionally evaluating coarsened partitions at most once per cycle, but it is also able to automate the partitioning. Since ESSENT acts as a code generator rather than a simple library, it is more able to automate and tailor the result to the target design. The ESSENT framework partitions the design as well as automatically generating the code necessary to detect activity and reuse outputs. ESSENT's partitioner only has a single parameter that corresponds to coarseness, and the fastest parameter value is largely insensitive to the input design [4]. Thus, a user can tune ESSENT once for their platform and reuse that parameter for a variety of designs. Since most users' platforms are relatively similar, they can probably use the default parameter value without even needing to worry about tuning it.

In addition to its primary optimization of dynamically skipping over inactive portions of the design, ESSENT has other optimizations. It is able to recognize *dynamic don't cares*

Design	Verilog Lines	FIRRTL Nodes	FIRRTL Edges
rocket16	112,167	26,554	47,290
rocket18	328,367	71,545	123,226
rocket20	246,589	70,349	120,236

TABLE I
OPEN-SOURCE PROCESSOR DESIGNS USED FOR EVALUATION

in which the value of a signal will not impact the simulation, and it dynamically skips over the unneeded portion. These dynamic don't cares frequently occur for unselected ways of multiplexers (Figure 2) as well as inputs to registers who currently have writes disabled. ESSENT also includes branch hints for the compiler in its generated code [3]. To support our generated code, we develop a C++ template library for arbitrary width digital signals. To represent signals internally, it uses appropriately sized storage and it is crafted to allow the compiler to optimize for multi-word arithmetic.

ESSENT is able to implement sophisticated optimizations with moderate developer effort by leveraging the productivity of the FIRRTL library as well as the Scala language [14]. The FIRRTL library not only provides a means to load designs in the FIRRTL intermediate representation, but it also provides many useful functions for processing and manipulating hardware designs. With those functionalities provided, we are able to focus our development efforts on aspects unique to simulation and our optimizations without needing to spend much effort on infrastructure to get started. The Scala language provides great productivity benefits from its object-oriented and functional language features as well as its safety. Even though ESSENT is written in Scala, it generates C++ that can be compiled to fast native code for a fast simulator.

ESSENT consumes designs in the FIRRTL format. FIRRTL is most commonly produced by designs written in Chisel [2], but other flows are possible. There are other languages that serve as frontends to produce FIRRTL, such as Spatial [13] and PyRTL [8]. For components written in Verilog, there are also tools to convert it to FIRRTL such as Yosys [24] and LiveHD [22].

IV. Performance Demonstration

We provide a brief performance comparison of ESSENT at various optimization levels compared to Verilator. We encourage the reader to consult prior publications for more thorough comparisons or performance analyses [3], [4], as the purpose of this comparison is to provide more recent data. We use an Intel 8-core 3.6 GHz i7-7820X (Skylake) which has 11 MB of L3 cache and 64 GB of DRAM to perform our experiments. For designs, we use Rocket Chip [1] from different years (Table I). Over time, the default configuration has changed and grown, so this provides some variety in scale. We animate the processors by running the dhrystone benchmark, and our prior work demonstrates the variance from multiple software workloads is moderate [4].

ESSENT accepts a command-line flag for optimization levels much like a compiler, with `-O0` performing no optimizations and `-O3` performing the most aggressive optimizations.

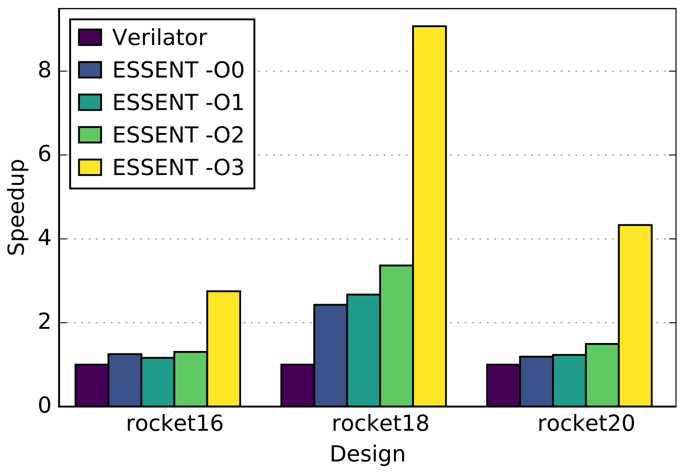

Fig. 3. Speedup of various ESSENT optimization levels relative to Verilator.

Without optimizations enabled, ESSENT is a straightforward full-cycle simulator, and its performance is similar to Verilator. The first level of optimization (`-O1`) removes long chains of wires without intervening gates and it streamlines the register update process. The next level of optimization (`-O2`) uses conditional execution to avoid dynamic don't cares. The highest level of optimization (`-O3`) uses our optimization to coarsely skip over inactive portions of the design. We expose the optimization levels to the user for two reasons. First, higher optimization reduce simulation time at the expense of increased compile time, but users may want to weigh that tradeoff for their usage scenario. Second, for users keen to examine and even modify the code generated by ESSENT, they may appreciate the simplicity of less optimized code.

In general, the benefit of ESSENT's optimizations becomes more apparent for larger designs. On smaller designs (e.g. rocket16), the host processor's frontend can learn the branching behavior and easily cache the instructions of the simulator program. As the design grows larger, so does its corresponding simulator program, and it will place more strain on the frontend. Eventually it will overflow the frontend, and the host processor will be executing at a significantly reduced rate. The speedup for rocket18 stands out as the design happens to straddle a qualitative size boundary. On Verilator, it overflows the frontend, but with ESSENT's optimizations, it can mostly fit, which causes the stark performance difference. For larger designs, the simulators produced by ESSENT will overflow the host processor's frontend, but the optimizations will still reduce the load and thus provide a speedup.

V. Future Work

ESSENT is available open source with a BSD license. Along with the code, we provide usage examples, and are in the process of creating and releasing more examples. ESSENT benefits from regression tests from past designs and they are executed frequently via continuous integration (CI). We are in the process of releasing our CI infrastructure to ease others testing efforts.

We are currently working on modest improvements to our partitioning process and the structure of our emitted code. We believe further refinements to our approach will allow us to continue to reduce the effective fraction of the design simulated and thus produce additional speedups. Furthermore, more analysis of how our hardware signals are handled by the compiler may allow us to improve code generation and memory layouts. Since this work initially began, we are delighted by the recent release of an open-source acyclic partitioner [11]. We are keen to test it out within our framework and consider its adoption or ways to improve our partitioner based on insights from it.

Due to the ubiquity of multicore processors, multicore parallelization is a clear opportunity to increase simulation speed. We are pursuing this path, but we find multiple caveats worth noting. First, although parallelization can reduce the time of a single simulation, it may hinder overall throughput. In many situations, there are many simulation tasks to be run, such as extensive testing or design space exploration. Running many simulations in parallel instead of a few parallelized simulations may yield better throughput at a system level. Second, parallelization is likely to be most beneficial for large designs. Cycle boundaries are a natural synchronization point between threads, as prior work using optimistic concurrency for hardware simulation found rollbacks to be frequent [15]. Due to the high simulation rates ESSENT is capable of, the designs presented in this work are likely to be synchronization-limited if parallelized. Thus, target designs will need to be significantly larger (perhaps greater than one million FIRRTL nodes) in order in order to enjoy reasonable parallel speedups.

VI. Discussion

In our work, we focus on software simulation, and we view hardware-accelerated simulation as a complementary alternative [12], [18]. Hardware-accelerated simulation can provide astronomical simulation rates, but it comes at the cost of higher equipments costs and start up times. For many situations, software simulation is more practical and this is often true within open-source projects which may have less resources. We expect many projects will continue to use both software and hardware-accelerated simulation depending on their needs and resources. We hope to be a common choice for software simulation.

As an open-source tool, we envision multiple communities that can benefit from ESSENT. First, it is an open-source simulator faster than other software simulators, so designers can plug it in and spend less time waiting for simulation results. Alternatively, with the efficiency improvement, they could perform more exhaustive simulation or verification. Second, ESSENT could serve as a great platform for researchers. Due to the productivity advantages of Scala as well as the FIRRTL library, ESSENT is an order of magnitude fewer lines of code than comparable simulators it outperforms. With that concise expressiveness, it is easier to use ESSENT as a foundation for pioneering new simulation techniques or augment a simulation in a desired way.

VII. Acknowledgments

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

References

[1] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, et al. Chisel: constructing hardware in a scala embedded language. *DAC*, pages 1216–1225, 2012.

[3] Scott Beamer. A case for accelerating software RTL simulation. *IEEE Micro*, 2020.

[4] Scott Beamer and David Donofrio. Efficiently exploiting low activity factors to accelerate RTL simulation. *DAC*, 2020.

[5] Richard Buchmann and Alain Greiner. A fully static scheduling approach for fast cycle accurate SystemC simulation of MPSoCs. *Internatonal Conference on Microelectronics*, pages 101–104, 2007.

[6] Colin C Charlton, D Jackson, and Paul H Leng. Lazy simulation of digital logic. *Computer-Aided Design*, 23(7):506–513, 1991.

[7] Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco. Event-driven gate-level simulation with GP-GPUs. *DAC*, page 557, 2009.

[8] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. A pythonic approach for rapid hardware prototyping and instrumentation. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[9] Robert S French, Monica S Lam, Jeremy R Levitt, and Kunle Olukotun. A general method for compiling event-driven simulations. In *DAC*, pages 151–156. ACM, 1995.

[10] JP Grossman, Brian Towles, Joseph A Bank, and David E Shaw. The role of Cascade, a cycle-based simulation infrastructure, in designing the Anton special-purpose supercomputers. *DAC*, 2013.

[11] Julien Herrmann, M Yusuf Ozkaya, Bora Uçar, Kamer Kaya, and Umit V Catalyurek. Multilevel algorithms for acyclic partitioning of directed acyclic graphs. *SIAM Journal on Scientific Computing*, 41(4):A2117–A2145, 2019.

[12] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, et al. Firesim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *International Symposium on Computer Architecture (ISCA)*, pages 29–42. IEEE Press, 2018.

[13] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, et al. Spatial: A language and compiler for application accelerators. *Conference on Programming Language Design and Implementation (PLDI)*, 53(4):296–311, 2018.

[14] Patrick S Li, Adam M Izraelevitz, and Jonathan Bachrach. Specification for the FIRRTL language. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-9*, 2016.

[15] Yukinori Matsumoto and Kazuo Taki. Parallel Logic Simulation on a Distributed Memory Machine. *European Conference on Design Automation*, 1992.

[16] Orlando Moreira, Merten Popp, and Christian Schulz. Graph partitioning with acyclicity constraints. *arXiv.org*, April 2017.

[17] Daniel Gracia Pérez, Gilles Mouchard, and Olivier Temam. A new optimized implementation of the SystemC engine using acyclic scheduling. *DATE*, 2004.

[18] Eric Schkufza, Michael Wei, and Christopher J Rossbach. Just-in-time compilation for verilog: A new technique for improving the fpga programming experience. In *ASPLOS*, pages 271–286, 2019.

[19] Steven P Smith, M Ray Mercer, and Bishop Brock. Demand driven simulation: BACKSIM. *DAC*, 1987.

[20] Wilson Snyder. Verilator: Speedy reference models, direct from RTL. *Presentation to University of Massachusetts Amherst*, 2017.

[21] Laung-Terng Wang, Nathan E Hoover, Edwin H Porter, and John J Zasio. SSIM: A software levelized compiled-code simulator. *DAC*, 1987.

[22] Sheng-Hong Wang, Rafael Trapani Possignolo, Haven Blake Skinner, and Jose Renau. LiveHD: A productive live hardware development flow. *IEEE Micro*, 40(4):67–75, 2020.

[23] Zhicheng Wang and Peter M Maurer. LECSIM: a levelized event driven compiled logic simulation. *DAC*, 1990.

[24] Claire Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/, 2016.