Title

Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation

ISBN

9781450367257

Authors

Beamer, Scott
Donofrio, David

Publication Date

2020-07-24

DOI

10.1109/dac18072.2020.9218632

Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation

Scott Beamer*
Computer Science & Engineering
University of California, Santa Cruz
Santa Cruz, CA, USA
sbeamer@ucsc.edu

David Donofrio*
Hardware Architecture
Tactical Computing Laboratories
San Francisco, CA, USA
ddonofrio@tactcomplabs.com

***Abstract*—Hardware simulation is a critical tool for design, but its slow speed often bottlenecks the entire design process. Although most signals in a digital design rarely change, most leading simulators still simulate the entirety of the design every cycle. Tracking which signals are unchanged and can thus be reused typically introduces too much overhead to deliver a practical speedup.**

In this work, we explore the challenge of efficiently detecting opportunities for reuse, and we demonstrate practical techniques to profitably exploit them. Thanks to our novel acyclic partitioning algorithm and other optimizations, our generated simulators outperform open-source and industrial state-of-the-art simulators.

I. Introduction

Due to the high financial and temporal cost of fabricating a chip, simulation is an invaluable tool for hardware design. Simulation is used in a variety of settings, whether it be development, design space exploration, debugging, verification, or validation. To improve simulation speed, common techniques include using reduced fidelity models (e.g. transaction-accurate simulation) or hardware acceleration (e.g. FPGA emulation). As such, cycle-accurate RTL simulation performed by software is still the most commonly used tool and remains a persistent bottleneck for hardware design. A substantial improvement in software cycle-accurate simulation performance could lead to multiple qualitative improvements. Faster simulation increases the number of designs that can be simulated in a day, which in turn could be used to: explore a larger design space, fix bugs quicker, increase coverage for verification, or even reduce computing costs.

Within many digital systems, many signals have low activity factors as they rarely change [6], [20]. A low activity factor suggests that most of the simulation results can be reused to save computation. Unfortunately, the overhead of tracking which signals have changed and triggering the necessary computation can often be much more onerous than the computation saved. For this reason, most state-of-the-art simulators use a full-cycle approach in which they evaluate the entirety of the circuit every cycle instead of a classic event-driven approach that is more activity proportional.

To profitably exploit low activity factors for a net speedup, two primary sources of overhead must be reduced: detecting which signals have changed and scheduling the needed signal computations. To amortize the overhead of detecting activity, we coarsen the granularity at which we track activity from single signals to modest partitions of 10-100s of elements. To reduce the overhead of dynamically scheduling the needed work, we generate a static schedule at compile time. The principal obstacle for such an activity-driven simulation approach is how to coarsen the design while still allowing for an efficient execution. Our solution to this coarsening problem is our key contribution: a novel acyclic graph partitioning algorithm.

* Early work performed at Lawrence Berkeley National Laboratory

In this work, we introduce the *essential signal simulation* approach, a method to efficiently exploit low activity factors in order to accelerate simulation. The result is an efficient simulation that skips over regions of inactivity while being unencumbered by scheduling overheads, repeat evaluations, or any substantial computation other than simulating the target design. Crucial to our approach is our acyclic partitioner which coarsens the design to reduce overheads. Improving significantly over prior related work, our approach is highly automated and requires near-minimal user intervention to operate. To demonstrate the utility of our insights, we implement ESSENT, a high-performance cycle-accurate simulator generator. Over multiple designs, our generated simulators outperform both a leading open-source simulator as well as a leading industrial simulator by $1.5 - 29.3\times$.

II. Simulation Background

Early RTL simulators computed the values of hardware designs by propagating signal updates as events, which also matched the semantics of the languages they modeled (e.g. Verilog). With these *event-driven* simulators, each time a signal is evaluated, it creates events to evaluate its children. To reduce unnecessary repeat evaluations, the events should be processed in a breadth-first manner, often referred to as *levelization* [22], [23].

Although levelized simulators eliminate unnecessary repeat activations, tracking which signals to activate and then dynamically scheduling them adds considerable overhead. *Full-cycle* simulators eliminate the scheduling overhead by performing the scheduling once at compile time and reusing it each cycle (static schedule) [12]. For a single static schedule to be sufficient, it needs to simulate the entirety of the design every cycle.

We define an efficient execution in which each signal is evaluated at most once per cycle to be a *singular* execution. A singular execution is possible with both full-cycle and event-driven approaches. In order for a schedule for a singular execution to exist, the graph must be acyclic [4]. Most hardware designs are acyclic or can be transformed to become acyclic. To break cycles from feedback paths through state elements (register or memory), each state element can be split into two nodes in the design graph (one for input & one for output). For combinational loops, the entire strongly connected component can be merged into a supernode. The supernode may need to be evaluated repeatedly until convergence, but the surrounding graph context is made acyclic by the transformation. In this work, we assume the designs are acyclic after initial transformations.

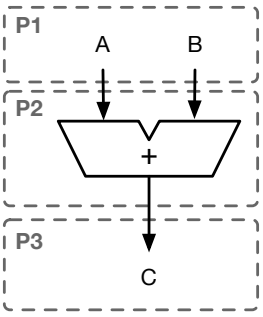
Low activity factors are common in digital designs. A low activity factor does not necessarily indicate wasteful design, as it is difficult to toggle every signal every cycle. Even the commonly used wave-

form file format Value Change Dump (VCD) exploits inactivity for compression by only recording signals when they change values.

Since in practice most signals rarely change, the simulation effort should ideally be proportional to the amount of activity in the design instead of the design size. Fortunately, event-driven simulation propagates activity and its simulation effort is thus activity proportional. Unfortunately, full-cycle simulation simulates the entire design every cycle and is thus activity oblivious. However, the current fastest simulators are typically full cycle, as the reduction in overhead outweighs the efficiency loss from not exploiting low activity factors.

For a full-cycle simulator to exploit low activity factors, it must detect which signals are unchanged to determine which outputs can be reused without computing them. The overhead of detecting and acting on these opportunities for reuse on a signal by signal granularity typically exceeds the benefit of the skipped computation. To amortize these overheads, the decision of which signals to conditionally evaluate should be performed on a coarser granularity.

How the design is *coarsened* can greatly hinder the efficiency or the usability of a simulator attempting to exploit inactivity. The design's module hierarchy does partition the design, but it is almost certainly cyclic. In the general case, these frequent module cycles almost entirely preclude singular execution. Prior work that uses the module hierarchy to exploit low activity factors has been forced to reevaluate modules multiple times per cycle [19] or unreasonably require the user to ensure the module graph is acyclic [11]. Other prior work groups together nearby signals into clusters [8]. However, to break cycles to make the cluster graph acyclic, they replicate portions of the design to reduce inter-cluster dependences.




```
…
evalP2()
  active[2] = false
  C$old = C
  C = A + B
  active[3] |= C != C$old
…
eval()
  …
  if (active[2]) evalP2()
  …

main()
  for (max_cycles)
    eval()
```

Fig. 1. Example of how a partition (P2) gets mapped to a function and fits into the rest of the simulator. When P2 is activated, it first deactivates itself for the next cycle, saves the old values of its outputs, computes itself using full-cycle simulation, and triggers its consumers if its output changed. P2 will not be activated again unless P1 activates it because A or B changes.

III. Essential Signal Simulation

We present our essential signal simulation technique whose goal is to expend simulation effort only on what is necessary and to minimize all other unnecessary computations. In particular, our method exploits low activity factors with low overhead by using a *conditional, coarsened, singular, static (CCSS)* execution schedule. Conditional execution allows for skipping the computation of unchanged signals. Coarsening the design reduces the overhead of detecting signal changes and triggering work. Singular execution ensures each component is evaluated at most once per cycle. A static schedule further reduces the scheduling overhead.

Efficiently achieving CCSS execution can be challenging, as the coarsening process is often at odds with singular execution (Section II). Our approach overcomes the shortcomings of prior work through the introduction of an acyclic partitioner. A partitioning, as opposed to a clustering, requires each signal to be in exactly one partition (not replicated). Requiring the partitioning to be acyclic guarantees the existence of a singular schedule.

Although CCSS execution provides outstanding simulation efficiency, to be usable, it must not place unreasonable constraints on the target design or overly burden the user. Our approach solves these usability concerns through automation. Our novel acyclic partitioner (Section IV) coarsens the design in a manner allowing for singular execution without user assistance. Furthermore, we also automatically generate all of the infrastructure to detect activity and trigger necessary executions.

A. ESSENT Structure

We implement *Essential Signal Simulation Enabled by Netlist Transformations (ESSENT)* to demonstrate the efficiency of our proposed technique. ESSENT is a simulator generator that given a hardware design, produces C++ code that can be compiled to produce a high-performance cycle-accurate simulator. We describe the structure of the simulators it produces, the optimizations they use, as well as the ESSENT tool itself.

The simulators generated by ESSENT use our proposed CCSS execution method. ESSENT partitions the design (coarsened), determines activation conditions for each partition (conditional execution), and schedules the partitions at compile time (static schedule). In the generated simulator, ESSENT generates code for each partition and joins them together with a single eval function. Thanks to the partitioning being acyclic, the static schedule only needs to consider each partition once (singular).

Partitions are only evaluated if they are active, so in addition to evaluating themselves, they must also detect which other partitions to activate (Figure 1). Once activated, each partition detects if its outputs change and activates the consumers of those outputs. Since a partition potentially has multiple outputs, we find it profitable to trigger activations at the fine granularity of individual outputs since it prevents unnecessary activations. Each partition will sleep the following cycle unless it is activated again. In the main eval function, the simulator also detects changes to external inputs to appropriately trigger dependent activations.

We perform the triggering for activity in the push direction, that is, each producer is responsible for awakening its consumers. Alternatively, we could perform the triggering in the pull direction (each partition checks if its inputs have changed), but we expect most partitions to be inactive most of the time, so the push direction results in less overhead. For the actual triggers themselves, we find using an OR-reduction as shown in Figure 1 delivers the best performance since it is branchless.

B. Simulation Optimizations

ESSENT utilizes numerous performance optimizations to accelerate its generated simulators including conditionally evaluating multiplexor ways or classic compiler optimizations such as dead code elimination, common subexpression elimination, and constant propagation. In this section, we outline our most novel optimizations.

1) Optimizing State Element Evaluation: We initially represent each state element as two nodes to eliminate potential cycles (Section II). With the following optimization, we are usually able to eliminate the second node from the split, saving both computation and memory space.

We are able to update a register immediately (eliminate the need for a second variable and a copy to it), if all of the register's consumers read it before it is updated [11], [15]. In terms of a directed graph representation of the design, this optimization is safe if and only if there is not a directed path from the register input node to any node that reads the register output node. Once the optimization has been determined to be safe, we add special ordering edges to the graph from all of the register's readers to the register input node to ensure the reads will be scheduled before the write.

Although prior work performs the register update elision optimization [11], [17], [21], we significantly improve it by making it compatible with our CCSS approach. Some prior coarsened simulators even exclude registers from conditional execution [8], thus limiting potential speedups.

We perform the state element update optimization pass after partitioning. Like the simpler unpartitioned case, when the optimization is possible, we ensure that every partition that reads the state element is scheduled before the partition that writes the state element. When state elements are incorporated into partitions, they must awaken their consumers in the following cycle. Our key insight is that the writing partition can immediately awaken its consuming partitions. If a state element is able to be updated in a single phase, that means all of its consumers have already executed that cycle. Thus, it can safely awaken them and those consuming partitions will not be evaluated until the following cycle. A state element can also trigger a wakeup to its own partition for the following cycle if that partition also reads that state element (feedback loop).

The benefit of performing state element updates within conditionally evaluated partitions is that it reduces scheduling overhead. State elements must wake up their consumers if they change, but these equality tests and partition activations are unnecessary if the state element inputs did not change. By incorporating a state element update into its partition, these scheduling activities can also be included in the partition, and thus performed only if the partition is active. We perform this operation not only for registers, but also for more complicated memories as well.

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

C. Implementation Details

ESSENT accepts hardware designs in FIRRTL, an intermediate language for hardware [16]. Compared to classic netlist formats, FIRRTL retains substantially more semantic information about the design which can better guide optimizing transformations. Since FIRRTL is an intermediate language, other projects can generate new frontends or backends for FIRRTL and reuse the rest of the tool flow. Chisel [2] is a hardware construction language that is the most mature FIRRTL frontend with the largest designs. Spatial [14] and PyRTL [9] have added support to emit Chisel or FIRRTL. Additionally, using Yosys, one can translate most synthesizable Verilog to FIRRTL [24]. Our tool, ESSENT, is a backend that consumes FIRRTL as its input, so it can take designs from any language that produces FIRRTL.

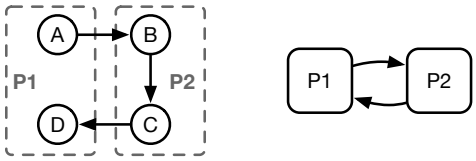


Fig. 2. Example of an acyclic graph (left) creating a cyclic partitioning (right). If the partitions are evaluated atomically, there is no way to get the correct result without reevaluating at least one partition. An alternate partitioning of $\{A, B\}$ and $\{C, D\}$ is acyclic.

ESSENT itself is implemented in ∼3,400 lines of Scala, and it reuses much of the infrastructure from FIRRTL's supporting library.

IV. Novel Acyclic Partitioning Algorithm

Coarsening the design is the biggest challenge for a CCSS simulation approach (Section II). In particular, the partitioning must be acyclic in order to enable an efficient schedule in which each partition is evaluated at most once per simulated cycle (singular execution). In an *acyclic partitioning*, the graph of partitions is acyclic. Even if the graph representing the hardware design is acyclic, partitioning the graph can induce cycles between partitions (Figure 2). In this section, we describe our novel acyclic graph partitioning algorithm which powers ESSENT.

Graph partitioning is a demanding optimization problem, whose most practical algorithms are largely driven by heuristics. Due to the complexity of these algorithms, it is typically preferable to use existing optimized libraries. Unfortunately, there is far less research on acyclic partitioning than general graph partitioning, and to the best of our knowledge, there are no open-source acyclic partitioners, despite claims to the contrary [18].

Our approach starts with an acyclic partitioning that has too many partitions, and it greedily merges partitions until no more merges are possible or the desired amount of coarsening is achieved. It quickly produces a reasonable acyclic partitioning by leveraging insights into common topological properties of hardware design graphs.

When merging partitions, in addition to using heuristics to consider the most profitable merges, we must also ensure each merge will not induce a cycle in the graph. We extend the work of Herrmann et al. [13], and find a simple test to see if partitions can be merged:

> Partitions A and B can be merged if and only if there is no external path in either direction between them.

An *external path* traverses nodes not in either partition. If there is an external path, when those partitions are merged, that path will be become a cycle. Our approach typically merges partitions that are adjacent to each other, so the edges that directly connect the two partitions are safely consumed within the new partition.

For our merging process, our primary goal is to eliminate small partitions, but we let the graph topology guide our approach and we do not strongly enforce balance constraints or limit the number of partitions. Small partitions are problematic, because they contain too few components to fully amortize the cut edges. To identify small partitions, we use a simple threshold parameter C_p, and any partition with fewer than C_p nodes is considered "small." In practice, C_p is mostly insensitive to the target design, and this is a significant improvement over prior work which exposed design-sensitive parameters to the user [8]. With a design-sensitive parameter, the user must retune the parameter for every design or even design change, while our design-insensitive parameter only needs to be tuned once for the host platform.

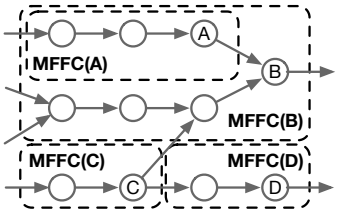


Fig. 3. Example maximum fanout free cones (MFFC) of nodes A, B, C, and D. Note if a node is in a MFFC, that node's MFFC is contained within that MFFC (e.g. MFFC(A) $\subset$ MFFC(B)).

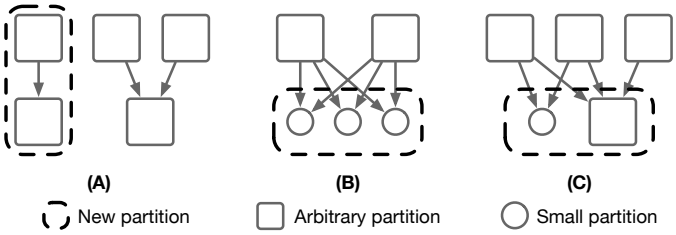


Fig. 4. Merging phases used by partitioner after initial MFFC decomposition

To bootstrap our algorithm, we first decompose the graph into *maximum fanout free cones (MFFC)*. The MFFC of a node v is the largest set of ancestors of v such that all of their descendants are either in the MFFC(v) or v itself (Figure 3). Sometimes MFFCs are trivially small (single node), and this happens when the target node v has siblings (shares a parent). MFFCs are useful building blocks, because any result from an MFFC(v) is only visible within the MFFC and at its target output node v. This beneficial property guaranties a MFFC-decomposition is acyclic [10].

Our merge-based acyclic partitioner performs the following:

Generate initial acyclic partitioning by decomposing the graph into MFFCs. To find a better MFFC decomposition, we start from the sink nodes (typically writes to state elements or external outputs) and crawl upwards identifying MFFCs.

Merge single-parent partitions into their parents. If all of the signals for a partition come from a single parent partition, the partitions can be safely merged (Figure 4A).

Merge small partitions with small siblings. Repeated structures such as operations on a bit-vector often result in partitions with high fanout to many small child partitions. To capture these structures in a single partition, we merge all small partitions (size $< C_p$) with their small siblings (Figure 4B). When choosing which merge to perform, we prioritize the absolute number of cut edges eliminated by a merge, which simultaneously maximizes the number of partitions in a merge as well as the number of common ancestors.

Merge small partitions with any siblings. The remaining small partitions do not have small siblings with which they share input signals. We repeatedly attempt to merge these small partitions with their potentially larger siblings, with a heuristic of maximizing the fraction of input signals in common (Figure 4C).

V. Evaluation

We establish the utility of our essential simulation technique by demonstrating ESSENT's performance advantages over prior work and by dissecting how it reduces overhead while exploiting low activity factors. In our evaluation, we use the following simulators:

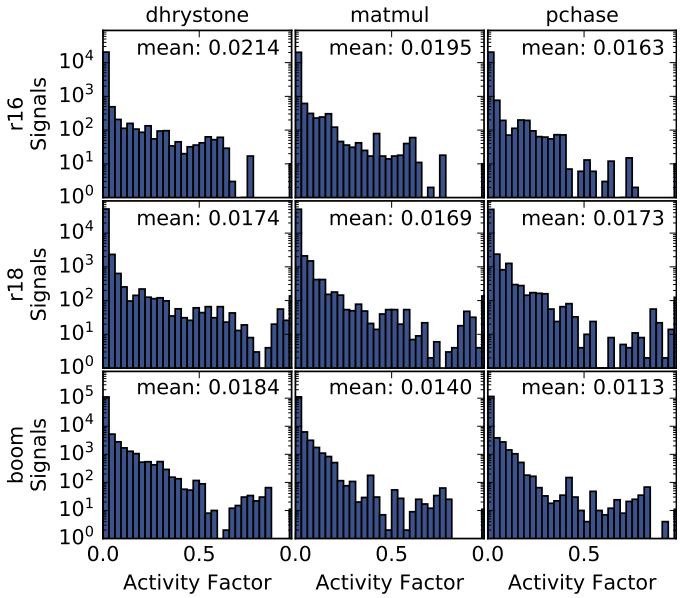


Fig. 5. Distribution of activity factors for all software workloads (horizontally) executing on all processors (vertically). Across all configurations, activities are typically low (note logarithmic y-axes).

Design	Verilog Lines	FIRRTL Nodes	FIRRTL Edges
r16	112,167	33,426	51,356
r18	328,367	67,803	123,151
boom	425,241	128,712	291,010

TABLE I
OPEN-SOURCE PROCESSOR DESIGNS USED FOR EVALUATION

Benchmark	Cycles (K)	Description
dhrystone	489.1	Dhrystone microbenchmark
matmul	715.8	Matrix multiplication benchmark
pchase	8,428.1	Pointer-chasing synthetic microbenchmark

TABLE II
SOFTWARE WORKLOADS FOR EVALUATION (CYCLE COUNTS FOR R16)

CommVer is a state-of-the-art commercial Verilog simulator (anonymized due to license). We appropriately finesse its options to maximize performance, including using 2-state simulation.

Verilator is an open-source Verilog simulator [21] with performance matching or exceeding commercial tools [3].

Baseline is a pure full-cycle simulator produced by the ESSENT tool flow with all optimizations disabled.

ESSENT is a CCSS simulator with all optimizations enabled, including the conditional execution of acyclic partitions.

To evaluate the simulators, we use open-source processor designs (Table I). Rocket Chip is a RISC-V SoC generator written in Chisel that is used in research and industry [1]. To create more designs, we use versions from both 2016 and 2018, and the increase in size shows the increasing sophistication of the default SoC configuration. We also use BOOM, an out-of-order processor generator [5]. We select three different software workloads to animate our target processor designs, and they expose a range of target CPU behaviors (Table II). 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.

We first measure the activity factors of all of our designs executing all of our workloads (Figure 5). We observe that typically only a few percent of signals change on a given cycle, and this is consistent

Design	Workload	CommVer	Verilator	Baseline	ESSENT	Speedup
r16	dhrystone	37.13	3.68	4.63	1.40	3.31
	matmul	54.21	5.17	7.12	1.85	3.84
	pchase	457.87	52.90	78.75	20.60	3.82
r18	dhrystone	46.21	40.97	26.71	4.01	6.65
	matmul	71.71	65.77	43.96	5.70	7.71
	pchase	831.26	743.03	485.51	69.87	6.95
boom	dhrystone	381.32	76.29	111.04	50.44	2.20
	matmul	431.67	109.70	161.17	59.85	2.69
	pchase	5529.25	1650.41	2534.32	746.69	3.39

TABLE III
EXECUTION TIMES (SEC.) & ESSENT'S SPEEDUP (×) OVER BASELINE

with prior work [6], [8], [20]. Interestingly, the IPC of the software workload the simulated processor executes has a significant relative impact on the activity factor, but only a modest change in absolute terms. These low activity factors provide an encouraging opportunity to accelerate simulation.

We next compare the simulators on overall performance (Table III). Our (unoptimized) baseline is comparable in performance to Verilator, which is reasonable since they are both full-cycle simulators. Verilator routinely outperforms CommVer, and raw performance is one of Verilator's most compelling features. ESSENT significantly outperforms the other simulators, besting Verilator by $1.5-11.5\times$ and CommVer by $7.2-29.3\times$. Despite being the newest and least mature simulator, ESSENT's performance advantage demonstrates the promise of the essential signal simulation technique.

Digging deeper into the performance results, ESSENT enjoys a substantially larger speedup on r18 than the other designs. The additional speedup ($2.86-2.99\times$) is primarily due to the branch hints, as they shrink the effective instruction working set below the cache size, greatly reducing the number of cache misses (Section III-B2). The effective instruction working sets for r16 and boom are not near the cache size, so they only experience a modest $1.02-1.13\times$ speedup from the branch hints.

Figure 6 shows the impact of our partitioning parameter C_p on the generated simulator's execution time. Across designs and workloads, the strong convergence on a good value for C_p demonstrates that our parameter is design-insensitive, so we select $C_p = 8$. Eliminating design-sensitive parameters from the process eliminates the need to perform extensive tuning before each new simulation [8].

The partitioning granularity is a tradeoff between overhead and the fraction of the design simulated (Figure 7). We define the *effective activity factor* to be the average fraction of the design simulated, which is greater than or equal to the actual activity factor of the design. ESSENT reduces the effective activity factor but it is counterweighted by overheads. We classify the overheads as either activity-agnostic *static overheads* or activity-dependent *dynamic overheads*. Static overheads are actions executed every cycle such as testing to see if a partition is active and should thus be simulated. Dynamic overheads are actions executed only if necessitated by activity such as a partition testing which of its outputs have changed to determine which subsequent partitions to activate.

Increasing C_p encourages the partitioner to merge more aggressively, which results in fewer partitions. With larger partitions, the effective activity increases since the execution is more coarse-grained. To first order, the static overheads are proportional to the number of partitions, while the dynamic overheads are proportional to the number of edges cut by active partitions. Thus, decreasing the number of partitions (increasing C_p) decreases static overheads. Interestingly, the dynamic overheads are roughly constant, as larger (and fewer) partitions typically have fewer edges cut. Overall, the highest performance is achieved with a moderately aggressive partitioning, which is a pragmatic balance between enjoying the benefit of a low effective activity factor while mostly reducing the static overheads.

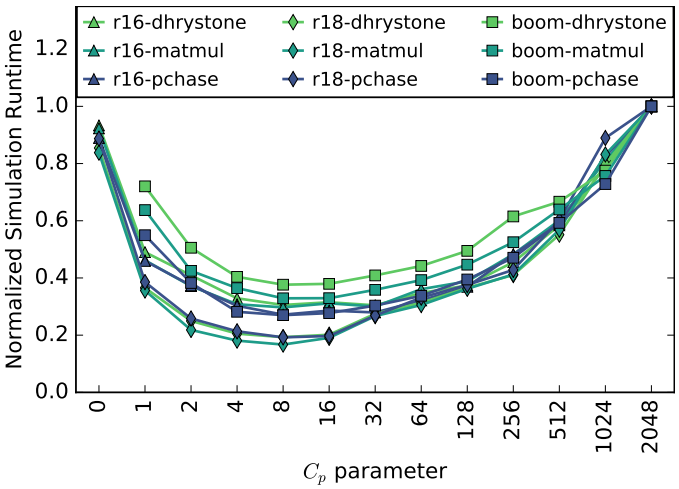


Fig. 6. The best partitioning parameter C_p for performance is mostly insensitive to the design and its software workload.

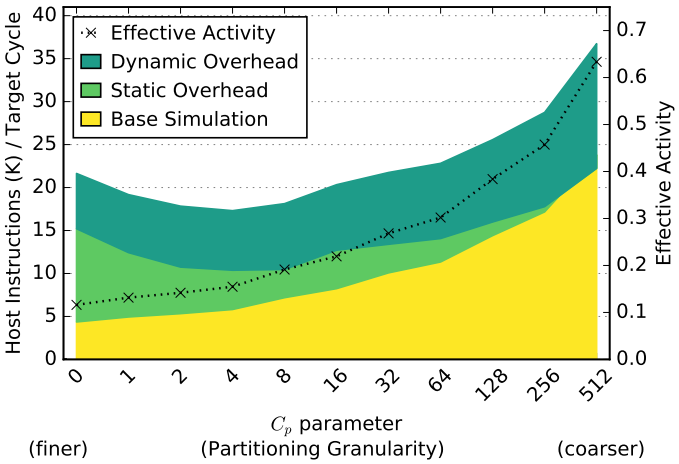


Fig. 7. Impact of partitioning parameter C_p on overhead and effective activity for r16 executing dhrystone. The selection of $C_p = 8$ (Figure 6) balances the two types of overhead with the base simulation work. For this figure, we calculate the overheads by executing different configurations and tracking host instructions executed.

VI. Related Work

Early prior work recognized low activity in designs as an opportunity for acceleration by only simulating a fraction of the design each cycle. Charlton et al. propose using lazy evaluation to make the simulation effort more activity proportional [6]. Although they succeed in reducing the number of signal activations, they do not report a practical speedup. Smith et al. propose BACKSIM, an approach that works backward from the desired outputs to determine the minimum number of signals to evaluate [20].

Our work leverages the strengths and improves upon many of the weaknesses of prior simulation work (Table IV). In particular, ESSENT greatly increases the automation for using a CCSS approach while simultaneously reducing overheads to deliver significant performance improvements in practice.

Approach	Conditional Execution	Coarsened Schedule	Static Schedule	Singular Execution	Coarsening Method	Coarsening Automated	Triggering Automated
Full-cycle (e.g. Verilator)			✓	✓	N/A	N/A	N/A
Event-driven (e.g. Icarus Verilog)	✓			✓	N/A	N/A	N/A
Pérez [19]	✓	✓	✓		user (via modules)		✓
Cascade [11]	✓	✓	✓	✓	user (via modules)		
Chatterjee [8]	✓	✓			clustering	✓	✓
ESSENT (this work)	✓	✓	✓	✓	acyclic partitioner	✓	✓

TABLE IV
COMPARISON OF SIMULATION APPROACHES WITH ATTRIBUTES DEFINED IN SECTION II

Pérez et al. implement an optimized SystemC simulator [19] that uses the user-provided module decomposition to coarsen the design and evaluates modules only if they are active. They use repeat evaluation to handle any cycles from the module graph.

Cascade is a C++ simulation library whose motivation to replace SystemC is driven by a focus on performance and memory usage efficiency [11]. Cascade uses a CCSS approach, but it lacks automation so much of the complexity of CCSS is passed onto the user. They use the user-defined modules to coarsen the design, so the user is responsible for breaking any cycles between modules. Additionally, for each module the user wants to potentially sleep, the user must provide a function to determine if the module has quiesced. By contrast, ESSENT not only automatically acyclically partitions the design, it also generates the necessary activity detection and triggering infrastructure.

Chatterjee et al. implement a high-performance gate-level simulator with optimizations for exposing parallelism and increasing regularity to execute well on GPUs [7], [8]. They coarsen the design with an acyclic clustering based on levelization. In order to increase the depth of the clusters while maintaining the acyclic constraint, they replicate gates to remove interdependences. In their evaluation, they report the amount of replication is $14 - 76\%$. Although the clustering is acyclic, we argue their execution is not singular due to replication since multiple copies of the same gate can be evaluated in a given cycle. The clustering process also requires design-specific parameters for which they recommend frequent autotuning. Their system only considers clusters of logic gates for conditional execution, and incurs overhead from unconditionally evaluating state elements. ESSENT has greater efficiency (static schedule, no replication, and conditionally evaluates state elements) and greater automation (design-insensitive partitioning parameter).

VII. DISCUSSION

In a quest to increase energy efficiency, we expect future hardware designers to continue to reduce activity in their designs, whether it be by writing RTL in a way to allow EDA tools to perform clock gating or by implementing power gating for whole blocks. To harness the simulation performance improvements possible with decreasing activity, simulation effort should be made activity proportional.

In this work, we demonstrate that low activity factors can be profitably exploited to accelerate simulation significantly. Our essential signal simulation technique combines CCSS execution with an acyclic partitioner. CCSS execution reduces both the simulation effort with conditional singular execution and the overheads with a static schedule on a coarsened granularity. Our acyclic partitioner provides the coarsening used to gracefully balance the benefit of skipping inactive components with the cost of detecting inactivity. Furthermore, our approach is highly automated, making its use much more practical. We are continuing to improve ESSENT, and it is available open source: `https://github.com/ucsc-vama/essent`

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] Pete Bannon, Ganesh Venkataramanan, Debjit Das Sarma, and Emil Talpes. Computer and redundancy solution for the full self-driving computer. In *IEEE Hot Chips 31 Symposium (HCS)*, 2019.

[4] 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.

[5] Christopher Celio, Krste Asanovic, and David Patterson. The berkeley out-of-order machine (BOOM): An industry- competitive, synthesizable, parameterized RISC-V processor. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167*, 2015.

[6] 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] Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco. Gate-level simulation with GPU computing. *Transactions on Design Automation Electronic Systems*, 16(3):1–26, 2011.

[9] 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.

[10] Jason Cong, Zheng Li, and Rajive Bagrodia. Acyclic multi-way partitioning of boolean networks. *DAC*, pages 670–675, 1994.

[11] 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.

[12] Craig Hansen. Hardware logic simulation by compilation. *DAC*, 1988.

[13] Julien Herrmann, Jonathan Kho, Bora Uçar, Kamer Kaya, and Ümit V Çatalyürek. Acyclic partitioning of large directed acyclic graphs. *International Symposium on Cluster, Cloud and Grid Computing*, pages 371–380, 2017.

[14] David Koeplinger, Matthew Feldman, et al. Spatial: A language and compiler for application accelerators. *PLDI*, 53(4):296–311, 2018.

[15] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. *PLDI*, 23(7):318–328, 1988.

[16] 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.

[17] Derek Lockhart, Gary Zibrat, and Christopher Batten. PyMTL: A unified framework for vertically integrated computer architecture research. *International Symposium on Microarchitecture (MICRO)*, 2014.

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

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

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

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

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

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

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