GSIM: Accelerating RTL Simulation for Large-Scale Designs

Lu Chen*†, Dingyi Zhao‡, Zihao Yu*, Ninghui Sun*† and Yungang Bao*†
*State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, China
†University of Chinese Academy of Sciences, China
‡Beijing Institute of Open Source Chip, China

***Abstract*—Register Transfer Level (RTL) simulation is widely used in design space exploration, verification, debugging, and preliminary performance evaluation for hardware design. Among various RTL simulation approaches, software simulation is the most commonly used due to its flexibility, low cost, and ease of debugging. However, the slow simulation of complex designs has become the bottleneck in design flow. In this work, we explore the sources of computation overhead of RTL simulation and conclude them into four factors. To optimize these factors, we propose several techniques at the supernode level, node level, and bit level. Finally, we implement these techniques in a novel RTL simulator GSIM. GSIM succeeds in simulating XiangShan, the state-of-the-art open-source RISC-V processor. Besides, compared to Verilator, GSIM can achieve speedup of 7.34x for booting Linux on XiangShan, and 19.94x for running CoreMark on Rocket.**

***Index Terms*—RTL simulation, optimization, large-scale**

I. Introduction

Register Transfer Level (RTL) simulation is essential in hardware design. It is widely used in various aspects, including design space exploration, verification, debugging, and preliminary performance evaluation [1] [2] [3]. It has become an indispensable step in the chip development workflow.

There are currently three main approaches to RTL simulation: software simulation, FPGA prototyping, and hardware emulation. Among them, FPGA prototyping is the fastest, but debugging tools are limited. Hardware emulation offers high speed along with good debugging ability, but it is extremely expensive. Software simulation provides 100% signal visibility, low cost and ease of debugging, but is the least efficient. Despite this drawback, software simulation remains the most commonly used method for RTL simulation [2].

With the increasing scale of modern digital circuits, the simulation speed of the software approach decreases dramatically, and becomes the bottleneck in verification [4]. Table I presents the simulation speed of processors at different scales using Verilator [5], a widely used open-source simulator. In this table, "IR node" refers to nodes in the RTL graph (including registers and logic units), and "IR edge" represents the number of connections between nodes, together indicating the complexity of each processor. As shown, the simulation speed for large-scale designs is typically limited to only a few kHz. Therefore, accelerating RTL simulation would significantly benefit the workflow for large-scale designs.

However, it is challenging to accelerate RTL simulation for large-scale designs. A software RTL simulator must simulate the signal propagation across all nodes in the circuit by computation. The heavy computation is the primary cause of the slow speed of RTL simulation. ESSENT [3] proposes an essential signal simulation approach that significantly accelerates RTL simulation. We aim to further explore opportunities to reduce computation demands.

TABLE I: Verilator (single thread) simulation speed of booting Linux on different processors. The host CPU is Intel i9-9900K. stuCore is designed by undergraduate student.

Name	Configuration	IR node	IR edge	Speed
stuCore	In-Order Single-Issue	9933	17369	∼ 900KHz
Rocket [6]	In-Order Single-Issue	234807	293164	∼ 30KHz
BOOM [7]	Out-of-Order Triple-Issue	571038	827619	∼ 9KHz
XiangShan [8]	Out-of-Order Six-Issue	6218427	9007005	∼ 0.9KHz

To identify the sources of computation, we further analyze the essential signal simulation approach. Listing 2 shows the behavior of simulating a single cycle. Each node is associated with an `active` bit. During each cycle, the simulator traverses all nodes in the graph. If the `active` bit of a node is not set (Line 7), its value can be reused and the evaluation is skipped to reduce computation. Otherwise, the value of this node needs to be evaluated (Line 3). If the value changes after evaluation, its successors will be activated (Line 4-5).

```
eval1():
  update new value
cycle():
  eval1()
  eval2()
```

Listing 1: Simulation framework in Verilator.

```
1  eval1():
2    save old value
3    update new value
4    if (old != new)
5      active[succ] = true
6  cycle():
7    if (active[1]) eval1()
8    if (active[2]) eval2()
```

Listing 2: Simulation framework in ESSENT.

From the analysis above, the sources of computation can be concluded to four factors: ❶ the accessing of `active` bits, including checking and updating, ❷ the value evaluation in each node, ❸ the total number of nodes, ❹ the activity factor, which is the ratio of active nodes to the total number of nodes.

This paper proposes a novel RTL simulator, GSIM [1], which aims to reduce the overhead of the four factors mentioned above through three levels of granularity, including supernode level, node level and bit level.

❶ **At the supernode level**, one challenge is to determine which nodes should be grouped into a supernode. A supernode is a set of nodes that are activated simultaneously, with the aim of reducing the activation overhead, but it may increase the activity factor. To balance this trade-off, we propose a novel partitioning algorithm to group nodes with strong correlations. This approach reduces the activation overhead while maintaining a low activity factor.

❷ **At the node level**, GSIM employs several techniques. First, GSIM eliminates redundant nodes and optimizes expressions during value evaluation by data flow analysis. Second, we observe that although inlining the value evaluation of one node into another helps to reduce the number of nodes, it may introduce additional computation. To determine whether inlining is beneficial, GSIM constructs a cost model and selects the choice with lower cost. Finally, GSIM moves reset handling to the slow path, which reduces the number of checking reset signals by orders of magnitude.

❸ **At the bit level**, we find that bits within each node may not be accessed simultaneously. By applying data flow analysis to each bit and splitting nodes according to their accessing patterns, GSIM can further reduce the activity factor.

GSIM successfully simulates XiangShan [8], the state-of-the-art open-source RISC-V processor. When booting Linux on XiangShan, GSIM achieves a 7.34x speedup compared to Verilator. When running CoreMark [9] on Rocket [6], GSIM reaches 19.94x of Verilator, outperforming the best between ESSENT [3] and Arcillator [10] by 2.52x.

II. Background and Analysis

A. RTL Simulation

To perform RTL simulation, the input design is represented as a directed graph. In this graph, each node corresponds to a register or logic unit, and each edge represents the propagation of signals between nodes. During simulation, the circuit's behavior is modeled as evaluation of signal values that propagate between nodes along the edges. The work of determining the evaluation order of nodes is called scheduling.

RTL simulation can be categorized into two types based on the scheduling policy [11]. In event-driven simulation, nodes are dynamically scheduled at runtime, with signal changes propagated as events. When the value of a node changes, an event is generated and sent to all its successors. This approach can flexibly model arbitrary delays, and is widely used in post-synthesis gate-level simulation. However, frequent scheduling in complex designs incurs high overhead. Commercial simulators like VCS [12] employ the event-driven simulation model.

In contrast, for full-cycle simulation, nodes are statically scheduled at compile time and evaluated in a fixed order at runtime. This eliminates the overhead of runtime scheduling. Although the minimum granularity of simulation is one clock cycle, it still suffices to perform functional verification for large-scale designs like processors. The open-source simulator Verilator [5] employs full-cycle simulation. Generally, full-cycle simulation is faster than event-driven simulation [13].

In a basic implementation of full-cycle simulation, to build a directed acyclic graph (DAG), registers are often split into two nodes (one for reading and one for writing) to break cycles in the graph. Then a topological sort is performed on this graph. In each cycle, all nodes are evaluated in the topological order, as shown in Listing 1. Verilator employs this approach.

To further improve the efficiency of full-cycle simulation, ESSENT [3] introduces an essential signal simulation approach. This approach is effective, because only a few nodes have their value changed in a cycle. Listing 2 shows the implementation. Each node is associated with an `active` bit. During a cycle, all `active` bits are examined. A node is evaluated only when its `active` bit is set. If the value of the node changes after evaluation, the active bits of its successors will be set to indicate the need of re-evaluation.

B. Modeling and Analysis

Inspired by ESSENT, we further explore optimization potential by identifying the sources of computation. According to the simulation framework in Listing 2, the simulation overhead per cycle T can be modeled as $T = ((E + A_{succ}) * af + A_{exam}) * N$, where E is the computation overhead of value evaluation in a node, A_{succ} is the overhead of activating successors, af is the activity factor, A_{exam} is the overhead of examining the `active` bit, and N is the number of nodes.

E represents the computation required to evaluate the value of a node based on its predecessors. It contains various operations, including arithmetic, shifting, bitwise, and conditional operations. A_{succ} and A_{exam} are introduced by the essential signal simulation approach. Each node requires an additional examination before evaluation and an activation step after evaluation. The former adds a branch instruction, while the latter involves a branch and several memory access instructions. N and af are determined by the circuit design. They together indicate the number of nodes with the `active` bit set in a cycle, which require evaluation. As the complexity of the circuit increases, N may grow a lot, but af remains low in most designs. According to our evaluation, af is about 4.61% when running CoreMark on XiangShan.

Although E, A_{succ}, A_{exam} and af seem negligible, their impact on T is amplified by a large N. Therefore, reducing N can significantly improve simulation performance. Additionally, small improvements on the other factors can still yield substantial performance gains. For example, during the simulation of Xiangshan, 82.26% of all executed branch instructions are dedicated to examining the `active` bits. Therefore, it is important to further optimize these factors.

III. GSIM Design

To accelerate RTL simulation, GSIM employs a series of optimization techniques targeting the factors above at three

[1] http://github.com/OpenXiangShan/gsim

levels: super-node level, node level, and bit level. Note that some techniques may improve one factor while degrading another. In such cases, GSIM balances the trade-offs among these factors with corresponding strategies.

A. Supernode Level Optimization

Checking the `active` bit of each node separately will incur high overhead for A_{exam} (Listing 2). To reduce A_{exam}, nodes are typically grouped into supernodes [3]. Each supernode is associated with a single `active` bit. Nodes inside a supernode share an `active` bit and are evaluated together (Listing 3). As a result, activating any node within a supernode leads to the evaluation of all nodes in it. We define the size of a supernode to be the number of nodes inside it. A supernode with larger size helps to reduce A_{exam}, but it may increase af.

```
eval_super1():
  eval1()
  eval2()
cycle():
  if (active[1])
    eval_super1()
```

Listing 3: Supernode level activation.

```
if (active[7:0] != 0)
  if (active[1])
    eval_super1()
  if (active[2])
    eval_super2()
  ...
```

Listing 4: Check multiple bits with a single condition.

To keep af low, GSIM tries to group nodes that are likely to be activated simultaneously into the same supernode. We observe that nodes in close proximity in the RTL graph often exhibit strong correlations and tend to be activated together. To find such nodes, a traditional graph partitioning algorithm may be a solution, such as Kernighan's Algorithm [14]. However, the goal of the traditional algorithm usually conflicts with building supernodes. This is because it tries to minimize the number of cut edges between partitions. It tends to divide nodes connected with a small number of edges into different partitions (the left of Figure 1). However, such nodes may be activated together during simulation, and they should be grouped into the same supernodes (the right of Figure 1).

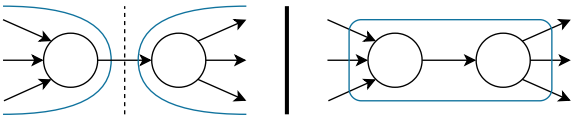

Fig. 1: Traditional partitioning algorithms fail to group nodes connected with few edges into the same partitions.

To address this challenge, GSIM employs an enhanced version of Kernighan's Algorithm. Specifically, the enhanced algorithm initially groups nodes with strong correlations based on certain rules, protecting them from being divided into different supernodes in the subsequent steps. These rules are derived from the following observations: ❶ A node with an out-degree of 1 is typically activated along with its successor. ❷ A node with an in-degree of 1 is usually activated when its predecessor is activated. ❸ Siblings with the same predecessors are always activated simultaneously. Subsequently, GSIM adopts the original Kernighan's Algorithm to construct supernodes. The maximum size of a supernode can be controlled by a parameter in the algorithm.

Additionally, to take advantage of the low af, GSIM speculatively assumes multiple `active` bits are clear, and uses a single condition to examine them on the fast path (Line 1 in Listing 4). If the condition holds, GSIM can skip the individual examination of each `active` bit to reduce A_{exam}.

B. Node Level Optimization

Redundant node elimination. To reduce N, we eliminate the following types of redundant nodes (Figure 2) by applying data flow analysis. ❶ **Alias Nodes** are duplicate nodes representing the same signal. ❷ **Dead Nodes** are nodes not used by other nodes. ❸ **Shorted Nodes** are nodes not selected due to another signal. ❹ **Unused Registers** are registers not used by other nodes, but they may be self-updated.

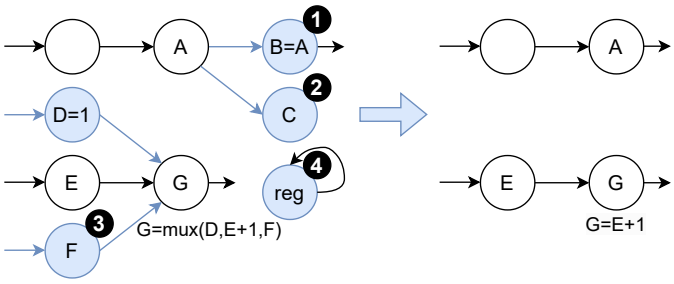

Fig. 2: Examples of eliminating redundant nodes.

Node inline and extraction. For non-redundant nodes, GSIM considers whether to inline the evaluation of one node into its successors. This is a trade-off between N and E. Extracting common parts of evaluation can reduce E by reusing the value stored in an extra intermediate node. For example, in the left part of Figure 3, node B stores the result of f(A), and its successors (C and D) can directly use the value in B without re-evaluating f(A). Conversely, reducing N may increase E. As shown in the right part of Figure 3, nodes C and D must re-evaluate f(A) without the intermediate node.

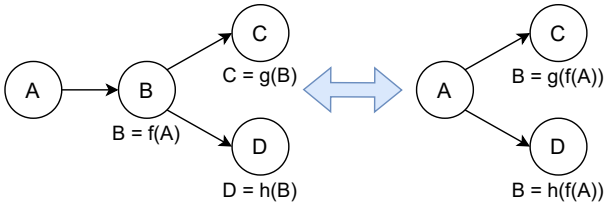

Fig. 3: Transform between node inline and extraction.

To address the trade-off above, GSIM models the cost of each decision and chooses the best one. Specifically, GSIM estimates the evaluation overhead of f(A), $cost(\text{f(A)})$, and the overhead of introducing a new node, $cost_{node}$, in terms of the number of operators involved, and the reference count of f(A), $\#\text{f(A)}$. If $cost(\text{f(A)}) \times \#\text{f(A)} > cost(\text{f(A)}) + cost_{node}$, f(A) is extracted as an extra node to reduce E. Otherwise, f(A) is inlined to reduce N.

Activation overhead optimization. To optimize A_{succ}, ESSENT replaces the condition checking (Line 4 and 5 in Listing 2) with logical operations to relieve the pressure of the host branch predictor [3]. However, we find that it may cause performance degradation when there are too many successors to activate, since this technique introduces extra operations for each successor to activate. To handle this trade-off, GSIM models the cost of each decision and chooses the optimal one, which is similar to the approach mentioned above.

Reset handling optimization. During the evaluation of each register with reset port, the reset signal must be checked every cycle to determine the value to be written (Listing 5). However, the number of reset signals is usually small, and they are hardly valid during simulation. Based on this observation, GSIM first speculatively assumes reset does not happen, and updates the register with the value evaluated based on its predecessors (Listing 6). This can remove the checking of reset signals from the fast path during evaluation, which helps to reduce E. After that, the reset signals are checked at the end of every cycle. In this way, GSIM can reduce the number of checking reset signals by orders of magnitude, from the number of registers with reset port to the number of reset signals in the design.

```
eval1():
  reg1 = reset ? 0 : new_val
  ...
cycle():
  if (active[1]) ...
  ...
```

Listing 5: Before reset optimization.

```
eval1():
  reg1 = new_val
  ...
check_reset():
  if (reset)
    reg1 = 0
    reg2 = init_val2
    ...
    activate_all_succ()
```

Listing 6: After reset optimization.

Expression simplification. GSIM employs various techniques to optimize expressions during evaluation, such as constant propagation and recognition of complex patterns. For example, it is common to generate a one-hot signal and check each bit of it in RTL design. This may result in two nodes: one to evaluate `B = 1 << A`, and the other to evaluate `C = bits(B, k, k)`, where `k` is a constant. GSIM detects this pattern and optimizes it to `C = (A == k)`.

C. Bit Level Optimization

An activation is unnecessary if the value of the node remains the same after evaluation. We find that in many long signals, only certain bits change each cycle. If a successor only uses the unchanged bits, activating it will be unnecessary. As shown in the left part of Figure 4, if node A changes while B and C are inactive, both D and E will change, leading to the activation of F and G. However, the value of G remains unchanged, since it does not depend on A. Therefore, the activation of G is unnecessary. According to our statistics, among nodes with multiple bits in XiangShan, 23.7% of them are composed of concatenations from other nodes, and 23.2% of the references to these nodes only access a subset of their bits.

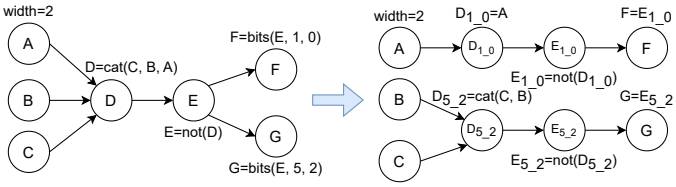

Fig. 4: Example of splitting nodes at the bit level.

To address this issue, GSIM adopts a bit-level node splitting policy. Specifically, GSIM applies data flow analysis to each bit. For a node P_0, if there exists a bit set S in P_0 and a path $P_0P_1 \dots P_n$, where P_n does not depend on $\overline{S}$ (the complementary bit set of S in P_0), GSIM can split all nodes on the path according to S and $\overline{S}$. In this way, we can remove the unnecessary activation of node P_n when only bits in $\overline{S}$ change, thereby reducing af. As shown in the right part of Figure 4, node A will no longer trigger the activation of G.

Note that although splitting nodes may increase N, we can apply node level optimization techniques mentioned above to the generated nodes.

D. Implementation Details

GSIM accepts circuits in Firrtl [15], an intermediate representation for hardware that various HDL can be converted into. GSIM includes a Firrtl parser that converts the input design into an abstract syntax tree (AST) and further transforms it into a graph. Most optimizations are performed on this graph. After optimization, GSIM emits C++ simulation code. Additionally, GSIM allows users to adjust the maximum size of a supernode from the command line.

IV. Evaluation

A. Evaluation Setup

We compare GSIM with the following simulators. ❶ **Verilator** [5] is the most widely used open-source Verilog simulator. We use version v5.026 with `-O3` flag to generate C++ files. It also supports multi-threaded simulation. ❷ **Arcilator** [10] is an MLIR [16] simulator developed as part of the CIRCT [17] project. ❸ **ESSENT** is a single-threaded Firrtl [15] simulator, with `-O3` flag to generate C++ files.

We evaluate the simulators on processors in Table I. Table II shows their versions. All processors are mainly developed in Chisel [18]. For a few Verilog modules, we replace them with Chisel implementations. As shown in Figure 5, each design is first converted into Chirrtl IR, and then translated into Firrtl, Verilog, and MLIR files using the FIRRTL [15] compiler or CIRCT [17] compiler. ESSENT and GSIM accept Firrtl as input, Verilator accepts Verilog, and Arcilator accepts MLIR.

TABLE II: The version of the processors.

	commit hash (truncated)	commit date
Rocket	e3773366a5c473b6b45107f037e3130f4	Sep 26, 2023
BOOM	9459af0c1f6847f8411622dac770ac78f	Oct 17, 2023
XiangShan	a42a7ffe5e4be903c75d53cf243761ce7	Mar 1, 2024

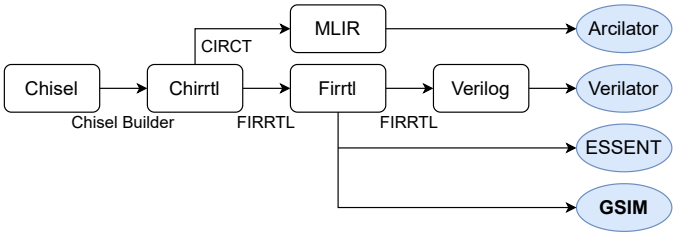

Fig. 5: The input of different simulators.

We choose CoreMark [9] and Linux as the software workloads. The former exhibits hot spots, while the latter does not. The host configuration is Intel 8-core i9-9900K CPU at 3.60GHz with 64GB of RAM. The host OS is Debian 12.

B. Overall Performance

Figure 6 shows the performance of different simulators on various processors and software workloads. Since Verilator is the most widely used open-source RTL simulator and successfully simulates all selected processors, we use Verilator as the baseline and normalize the other results to it.

In terms of correctness, aside from Verilator, GSIM is the only simulator that can successfully simulate XiangShan. ESSENT and Arcilator fail to simulate some of the designs due to errors during emitting C++ files or wrong simulation results. For performance, GSIM achieves speedup of 7.34x for booting Linux on XiangShan, and 19.94x for running CoreMark on Rocket. We also evaluate Verilator with multi-threading. GSIM significantly outperforms multi-threading Verilator on most of the designs.

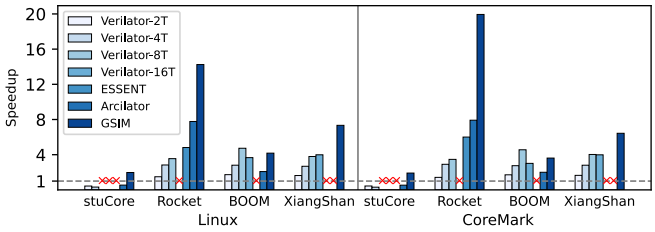

Fig. 6: Overall performance. The speedup is normalized to the single thread version of Verilator. '2T' means simulating with 2 threads. '×' means failing to perform the simulation.

C. Running SPEC CPU2006

In CPU simulation, checkpoints have long been used to accelerate performance evaluation [8] [19]. We sample SPEC CPU2006 [20] using SimPoint [21] and extract 40M-instruction segments to create checkpoints. The selected benchmarks cover various types of workloads, including streaming and irregular memory access, integer and floating-point computation, branch-intensive operations, and instruction-cache-intensive [22]. We run these checkpoints on XiangShan using both Verilator and GSIM. As shown in Figure 7, GSIM is 3.72x faster than single-threaded Verilator across all checkpoints on average, and achieves an average 1.18x of 8-threaded Verilator.

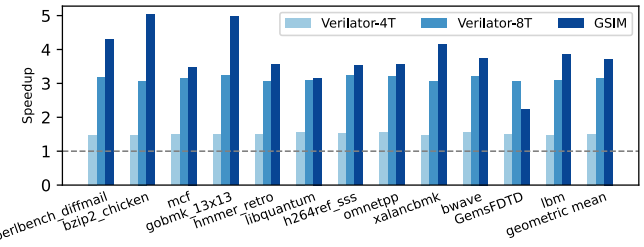

Fig. 7: SPEC CPU2006 performance on XiangShan. The speedup is normalized to the single thread version of Verilator.

D. Performance Breakdown

To evaluate the contribution of each optimization technique adopted in GSIM, we apply all techniques incrementally, starting with a baseline with no optimizations. In this way, we get a series of performance data $P_0, P_1, \ldots, P_n$. In Figure 8, the height of the bar for the ith technique is calculated as $\log_{10}(P_i/P_{i-1})$. As shown in the figure, introducing supernode significantly improves performance for all designs. Splitting nodes at bit level delivers substantial improvement for BOOM and notable gains for XiangShan, but has little impact on stuCore and Rocket.

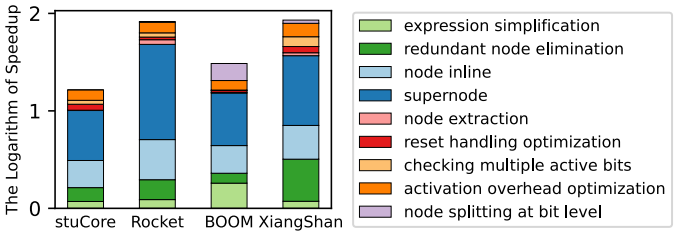

Fig. 8: Performance breakdown for each technique.

E. Selecting the Maximum Size of Supernode

We also investigated the relationship between the maximum size of supernode and the simulation performance. As mentioned above, a larger size of supernode reduces A_{exam} but increases af. The optimal size depends on the circuit design. To find the optimal size, we try different sizes with all other optimization techniques enabled. For the designs we select, the optimal size ranges from 20 to 50, as shown in Figure 9.

F. Different Algorithms to Build Supernode

We compare the enhanced partitioning algorithm used in GSIM against the original Kernighan's Algorithm and ESSENT's partitioning algorithm (noted as MFFC-based). These algorithms are implemented on GSIM with all other optimization techniques disabled to show their effectiveness. They are evaluated by running CoreMark on BOOM under their own optimal parameters. We measure the number of supernodes, the number of times to active a successor, and the number

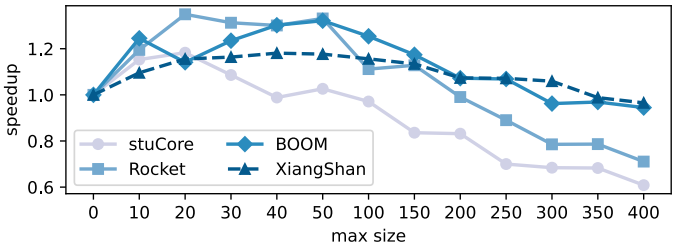

Fig. 9: Performance with respect to maximum supernode size.

of active node. These metrics can reflect A_{exam}, A_{succ} and E respectively. As shown in Table III, our algorithm performs the best in terms of the simulation speed, by achieving a better balance among the three factors above.

TABLE III: Comparison of different partitioning algorithms by running CoreMark on BOOM.

	partition time (s)	supernode	activation times	active node	speed (Hz)
None	–	536721	64044	42398	913
Kernighan	1.3	21769	13104	93327	4522
MFFC-based	88.9	36478	7310	124202	6802
GSIM	1.6	14261	8169	114255	8184

G. Resource Usage

For the emission time, we measure the time required to emit simulation files. As shown in Table IV, although GSIM outperforms Verilator, GSIM still delivers comparable emission time to Verilator. ESSENT requires more time to generate C++ files. One reason may be the slow execution of Scala programs. Arcilator requires more than 100GB memory to emit code for BOOM, which leads to accessing swap with a very long emission time. It even fails to emit code for XiangShan due to out-of-memory.

For the code size, we measure the size of the `.text` section of the binary reported by `readelf`. As shown in Table IV, GSIM emits the least amount of code among all simulators, as it employs more optimization techniques.

For the data size, we use `sizeof` to obtain the size of the top-level design class in C++ files, whose members contain all variables used for simulation, and we exclude the array used to simulate the main memory (128MB). As shown in Table IV, GSIM generates slightly larger data than Verilator.

V. Related Work

Verilator [5] is a widely used open-source, full-cycle RTL simulator. It supports multi-threading by partitioning the RTL design. Arcilator [10], an RTL simulator based on CIRCT, applies optimizations at various IR levels. Both of them evaluate all signals in the circuit every cycle and improve simulation speed through various expression optimization techniques. In contrast, GSIM skips the evaluation of a signal when it is inactive. Khronos [2] employs cross-cycle optimizations to reduce memory access overhead during simulation, which performs well in circuits with many pipeline registers. However, GSIM targets more general designs. ESSENT [3] leverages the low activity factor in circuits, partitioning the design and filtering out inactive regions to efficiently reduce the computation overhead. GSIM further explores the potential of accelerating RTL simulation. It reduces the computation overhead from four factors with optimization techniques across three levels.

RepCut [1] is a multi-threaded RTL simulator based on ESSENT. It reduces inter-thread synchronization overhead by introducing redundant computation. [13] proposes a coarse-grained circuit deduplication method to improve throughput for a batch of simulation tasks. GSIM focuses on improving the performance of single-threaded simulation under a single task. These works are orthogonal to GSIM and provide directions for future work.

TABLE IV: Comparison of resources across various designs. Instances labeled with '*' mean failing to run.

Design	Simulator	Emission Time(s)	Code Size(B)	Data Size(B)
stuCore	Verilator	1.1	394K	15K
	*ESSENT	7.7	659K	21K
	Arcilator	0.4	131K	13K
	GSIM	0.4	133K	16K
Rocket	Verilator	5.8	2.9M	62K
	ESSENT	37.8	1.6M	107K
	Arcilator	2.9	305K	49K
	GSIM	9.8	1.4M	72K
BOOM	Verilator	22.9	7.7M	954K
	*ESSENT	-	-	-
	Arcilator	4194.6	2.8M	799K
	GSIM	31.3	4.4M	976K
XiangShan	Verilator	374.6	40.7M	22.2M
	*ESSENT	-	-	-
	*Arcilator	-	-	-
	GSIM	389.1	25.4M	22.3M

VI. Conclusion

This paper proposes GSIM, a novel RTL simulator based on the essential signal simulation approach. GSIM employs a series of optimization techniques at the supernode level, node level, and bit level. GSIM successfully simulates XiangShan, the state-of-the-art open-source RISC-V processor. Compared to Verilator, GSIM achieves a 7.34x speedup when booting Linux on XiangShan, and a 19.94x speedup when running CoreMark on Rocket.

Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable feedback and comments. This work is supported by the National Key R&D Program of China (2022YFB4500403).

References

[1] H. Wang and S. Beamer, "Repcut: Superlinear parallel rtl simulation with replication-aided partitioning," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 572–585. [Online]. Available: https://doi.org/10.1145/3582016.3582034

[2] K. Zhou, Y. Liang, Y. Lin, R. Wang, and R. Huang, "Khronos: Fusing memory access for improved hardware rtl simulation," in *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023, pp. 180–193.

[3] S. Beamer and D. Donofrio, "Efficiently exploiting low activity factors to accelerate rtl simulation," in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, ser. DAC '20. IEEE Press, 2020.

[4] L. Lavagno, I. Markov, G. Martin, and L. Scheffer, *Electronic Design Automation for IC System Design, Verification, and Testing*. CRC Press, 2017. [Online]. Available: https://books.google.com/books?id=UeobDAAAQBAJ

[5] W. Snyder, "Verilator," https://veripool.org/verilator/.

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

[7] C. Celio, D. A. Patterson, and K. Asanović, "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor," Tech. Rep. UCB/EECS-2015-167, Jun 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[8] Y. Xu, Z. Yu, D. Tang, G. Chen, L. Chen, L. Gou, Y. Jin, Q. Li, X. Li, Z. Li, J. Lin, T. Liu, Z. Liu, J. Tan, H. Wang, H. Wang, K. Wang, C. Zhang, F. Zhang, L. Zhang, Z. Zhang, Y. Zhao, Y. Zhou, Y. Zhou, J. Zou, Y. Cai, D. Huan, Z. Li, J. Zhao, Z. Chen, W. He, Q. Quan, X. Liu, S. Wang, K. Shi, N. Sun, and Y. Bao, "Towards developing high performance risc-v processors using agile methodology," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1178–1199.

[9] "Coremark," https://www.eembc.org/coremark/.

[10] Z. Y. B. H. T. G. Martin Erhart, Fabian Schuiki, "Arcilator: Fast and cycle-accurate hardware simulation in circt," https://llvm.org/devmtg/2023-10/slides/techtalks/Erhart-Arcilator-FastAndCycleAccurateHardwareSimulationInCIRCT.pdf.

[11] L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009.

[12] "Vcs: Functional verification solution," https://www.synopsys.com/verification/simulation/vcs.html.

[13] H. Wang, T. Nijssen, and S. Beamer, "Don't repeat yourself! coarse-grained circuit deduplication to accelerate rtl simulation," 2024.

[14] B. W. Kernighan, "Optimal sequential partitions of graphs," *Journal of the ACM (JACM)*, vol. 18, no. 1, pp. 34–40, 1971.

[15] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson *et al.*, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 209–216.

[16] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[17] "Circt: Circuit ir compilers and tools," https://circt.llvm.org/.

[18] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 1216–1225.

[19] "Gem5: Checkpoints," https://www.gem5.org/documentation/general_docs/checkpoints/.

[20] "Spec cpu® 2006," https://www.spec.org/cpu2006/.

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGPLAN Notices*, vol. 37, no. 10, pp. 45–57, 2002.

[22] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the spec cpu2006 benchmark suite," in *Proceedings of the 34th annual international symposium on Computer architecture*, 2007, pp. 412–423.