ERASER: Efficient RTL FAult Simulation Framework with Trimmed Execution Redundancy

Jiaping Tang*†‡, Jianan Mu*†✉, Silin Liu*†‡, Zizhen Liu*†, Feng Gu*†‡, Xinyu Zhang*†‡, Leyan Wang*†‡, Shengwen Liang*†, Jing Ye*†‡, Huawei Li*†‡✉, Xiaowei Li*†

*State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

†University of Chinese Academy of Sciences, Beijing, China

‡CASTEST Co., Ltd., Beijing, China

{tangjiaping22s, mujianan, liusilin23s, liuzizhen, gufeng22s, zhangxinyu23s, wangleyan24s, liangshenwen, yejing, lihuawei, lxw}@ict.ac.cn

***Abstract*—As intelligent computing devices increasingly integrate into human life, ensuring the functional safety of the corresponding electronic chips becomes more critical. A key metric for functional safety is achieving a sufficient fault coverage. To meet this requirement, extensive time-consuming fault simulation of the RTL code is necessary during the chip design phase. The main overhead in RTL fault simulation comes from simulating behavioral nodes (always blocks). Due to the limited fault propagation capacity, fault simulation results often match the good simulation results for many behavioral nodes. A key strategy for accelerating RTL fault simulation is the identification and elimination of redundant simulations. Existing methods detect redundant executions by examining whether the fault inputs to each RTL node are consistent with the good inputs. However, we observe that this input comparison mechanism overlooks a significant amount of implicit redundant execution: although the fault inputs differ from the good inputs, the node's execution results remain unchanged. Our experiments reveal that this overlooked redundant execution constitutes nearly half of the total execution overhead of behavioral nodes, becoming a significant bottleneck in current RTL fault simulation. The underlying reason for this overlooked redundancy is that, in these cases, the true execution paths within the behavioral nodes are not affected by the changes in input values. In this work, we propose a behavior-level redundancy detection algorithm that focuses on the true execution paths. Building on the elimination of redundant executions, we further developed an efficient RTL fault simulation framework, Eraser. Experimental results show that compared to commercial tools, under the same fault coverage, our framework achieves a 3.9 × improvement in simulation performance on average.**

***Index Terms*—RTL fault simulation, functional safety verification**

I. Introduction

With the growing integration of intelligent computing systems like autonomous vehicles and smart robotics into human life, the functional safety of electronic chips is receiving heightened attention. The ISO 26262 standard [1] is introduced to define specific functional safety requirements for automotive-grade chips, such as microcontroller unit (MCU), application-specific integrated circuit (ASIC), and system-on-chip (SoC). According to ISO 26262, ensuring functional safety requires high fault coverage in chip design, necessitating extensive fault simulations and significant time costs [10]. Compared to the high computational cost of gate-level simulation and the insufficient accuracy of instruction-level simulation, RTL-level fault simulation has become the mainstream approach for functional safety verification. However, RTL fault simulations are also computationally intensive, often becoming a bottleneck in real-world development cycles [30], [31]. Therefore, accelerating RTL fault simulation is urgently needed.

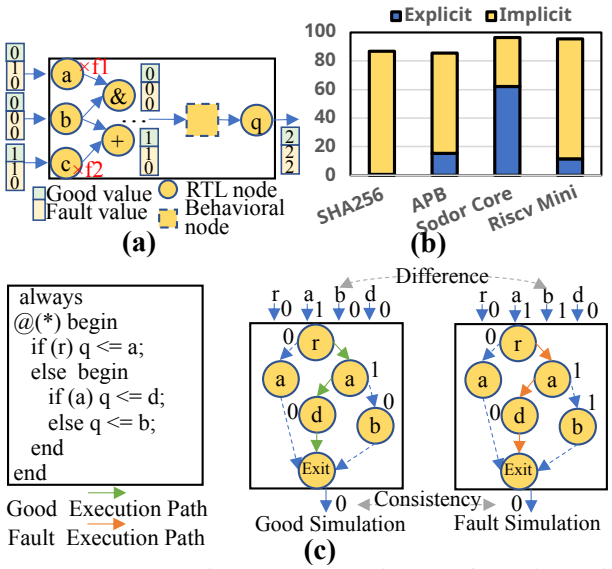

Fig. 1: (a) The execution redundancy in RTL fault simulation. (b) The ratio of explicit and implicit redundancy. Explicit redundancy occurs when the good and fault inputs of a node are identical, while implicit redundancy occurs when the fault inputs differ from the good, yet retain the same outputs. (c) Example of implicit redundancy.

The RTL design consists of behavioral nodes (codes of always blocks) and RTL nodes (logical connections linking these behavioral nodes) [8], which shows in Fig .1(a). Fault simulation for RTL code involves stimulating behavioral and RTL nodes with both good and faulty inputs. Based on our measurements, shown in Section V-C, the simulation time cost of behavioral nodes accounts for 60% of the total cost. However, due to the limited fault propagation capabilities in

978-3-9826741-0-0/DATE25/© 2025 EDAA

most cases, the propagation of both faulty and good values tends to be consistent, leading to identical simulation results across a large number of nodes. Therefore, fault simulations that produce the same results as good simulations at a given node are redundant. We illustrate these redundancies in Fig. 1(a). Our experiments indicate that such redundant executions account for approximately 90% of the behavioral node simulation time. Consequently, the key to improving fault simulation performance is to eliminate these redundant executions. Existing RTL fault simulation frameworks [14], [15] detect the inputs to each node, and for a behavioral node, if its faulty inputs match the good inputs, the corresponding fault simulation is skipped.

However, this approach of directly checking node inputs fails to fully eliminate the redundant execution of behavioral nodes. We find that beyond the explicit redundant executions where the inputs between good and fault simulations are identical, a significant amount of implicit redundancies for behavioral nodes remains unsettled. We illustrate such a case in Fig. 1(c): although the fault inputs differ from the good inputs, the final output remains the same. Since the fault inputs are different from the good inputs, this redundant execution is overlooked by the existing method. We test the proportion of these implicit redundant executions across several circuits. The results shown in Fig. 1(b) indicate that these implicit redundancies account for almost half of the total behavioral node executions in these cases. Therefore, the challenge in improving the performance of RTL fault simulation is addressing implicit redundancies of behavioral nodes.

To this end, we propose Eraser, an efficient batched RTL fault simulation framework to eliminate redundant executions. First, we develop a redundancy detection algorithm based on execution paths at run-time. This algorithm identifies implicit redundancy by checking the consistency of execution paths and data dependencies. Next, we integrate this algorithm with explicit redundancy detection, which is based on input comparison at behavioral nodes, to create a simulation framework that thoroughly eliminates behavioral node redundancies. By removing the redundancies more effectively, our framework achieves significant performance improvement compared to existing RTL fault simulation methods. The main contributions are summarized as follows.

- We identify significant implicit redundancies of behavioral nodes in RTL fault simulation and propose a redundancy detection algorithm based on execution paths at runtime to eliminate them.
- We propose the Eraser framework, which aims to exploit implicit redundancy detection while effectively managing both good and faulty behavioral executions as well as RTL node operations.
- The experimental results show that compared to the state-of-the-art commercial simulator and an open-source fault simulator, our simulator achieves an average performance acceleration of 3.9× and 5.9×, respectively.

II. Background

A. RTL code and fault simulation

The RTL code can be represented as a directed graph [8], as shown in Fig. 2. The directed graph termed the RTL Graph, represents the signal connections between variables, arithmetic and logic operations, and behavioral code in the RTL design. We refer to the nodes composed of behavioral codes as behavioral nodes, and other nodes as RTL nodes.

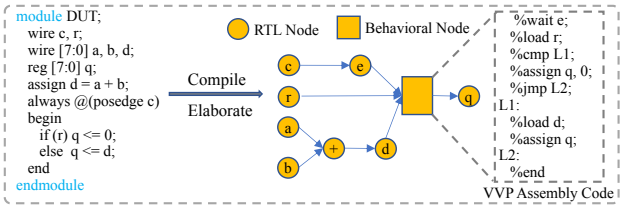

Fig. 2: A RTL code and the internal representation [8].

Concurrent is a batch fault simulation algorithm widely used in gate-level fault simulation, which we have extended to the RTL level. To facilitate understanding in the following sections, several key terms need to be clarified.

- **Good gate and bad gate.** A good gate exists in the fault-free network and maintains a list of bad gates. A bad gate exists in the faulty network and is used to store any differences arising from faults.
- **Visible and invisible bad gates.** If the output value of a bad gate is identical to the good gate, then the bad gate is considered invisible. Conversely, the bad gate is visible.
- **Good and faulty behavioral code.** A good behavioral code is activated by a good gate, with the execution path referred to as the good execution path. Conversely, a faulty behavioral code is activated by a bad gate, with the execution path referred to as the faulty execution path.

B. Related work

Despite considerable progress, RTL fault simulation methods still face key challenges. Most approaches only support single fault simulation [4]–[6], [9]–[11], [13], [18], and fail to address path convergence effectively, leading to redundant computations and suboptimal performance. While some efforts have explored concurrent fault simulation at the RTL level [14], [15], these solutions remain incomplete, especially in dealing with implicit redundancies in behavioral code, which contribute significantly to computational overhead.

Techniques such as RTL code modification [4]–[6] and using built-in commands in simulators [7], [8] have been proposed, but these methods either require extensive manual intervention or rely heavily on specific tools. Higher-level fault simulation [9]–[11] can improve efficiency but often sacrifices accuracy. Recent efforts to enhance Verilator [13], [18] still lack support for batch fault simulation, limiting scalability. Multilevel batch fault simulators [14], [15] offer better performance but do not fully address redundancies in behavioral nodes. Therefore, eliminating these redundancies remains a crucial opportunity for improving RTL fault simulation efficiency.

III. Motivation of Eraser

To analyze the core mechanisms of implicit redundancy, we compare the implicit redundancy and explicit redundancy in Fig. 3, where the box illustrates the internal execution path under different input values of a behavior code. The nodes are categorized into two types: path decision nodes (green) and data dependency nodes (gray). Path decision nodes represent branch statements in RTL, while data dependency nodes indicate data dependencies along the path. Given a set of inputs, a specific execution path is selected, and data propagation along this path produces a result.

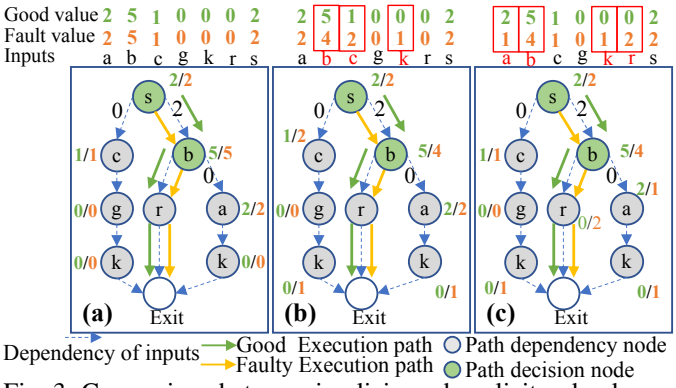

Fig. 3: Comparison between impliciy and explicit redundancy.

Fig. 3(a) illustrates an explicit redundancy node, where the inputs remain unchanged, leading to the same output. Fig. 3(b) represents an implicit redundancy. In this case, the fault inputs for signals *b*, *c*, and *k* differ from the good ones. Specifically, signal *b* is a path decision node and the value is changed, but the execution path remains unchanged. Meanwhile, these input changes do not appear in the data dependencies of the actual execution path, resulting in identical simulation outcomes for both good and faulty behavioral codes. This indicates execution redundancy at the behavioral node. In Fig. 3(c), the changes in the fault do not affect the execution path, but the value of signal *r* influences the data dependencies on the actual path, causing a different result. Therefore, this cannot be identified as a redundant execution. From the above comparison, it can be concluded that detecting implicit redundancy is to determine whether the differences between the fault and good values affect the choice of execution path and the data dependencies on this path.

IV. Eraser Framework with Redundant Node Detection

The framework takes the RTL code as input and ultimately outputs the fault coverage of the circuit. The framework consists of eight steps. Step 1 is to compile and elaborate the RTL code to generate an RTL graph, which includes RTL nodes and behavioral nodes. The concurrent fault simulation is used to reduce execution redundancy on the RTL nodes in step 2 and 3. Some good and faulty behavioral codes will be activated by RTL nodes after RTL nodes simulation in step 4. Faulty behavioral codes will be skipped if redundancy detection

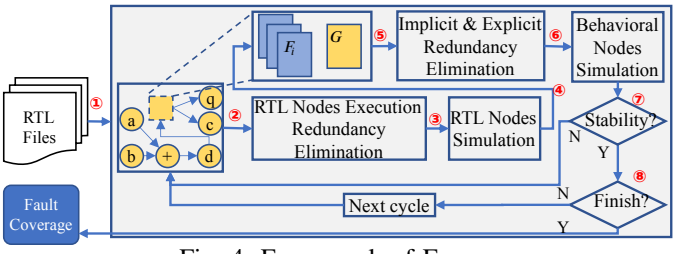

Fig. 4: Framework of Eraser.

methods determine them to be redundant in step 5. Otherwise, they will be executed in step 6. It is worth noting that ERASER introduces redundancy detection based on the actual execution flow to identify implicit redundant that previous methods have overlooked. A detailed explanation of this detection method is provided in Section IV-A. Additionally, we have reproduced the explicit redundancy detection methods for behavior and RTL nodes from existing approaches, as explained in Section IV-B and IV-C. Steps 2 to 6 are iterated based on whether the RTL nodes or behavioral nodes have reached a stable state, or if the entire simulation has been completed, otherwise, output the fault coverage.

A. Implicit Redundancy Elimination and Fault Simulation on Behavioral Nodes

To efficiently eliminate redundant execution in behavioral nodes, we propose Algorithm 1, an execution flow redundancy elimination approach. Algorithm 1 takes behavior nodes, and fault id as input. By using the good execution path as a reference and monitoring the path decision nodes and path dependency nodes, it detects the redundant execution of faulty behavioral codes where the fault input differs from the good input but the output remains consistent with the good behavioral codes. We describe the Algorithm 1 in the context of a circuit case, as shown in Fig. 5.

Preprocess (Line 1 in Algorithm 1): To trace the execution path of behavioral code, we first partition the code, with each partition representing a potential execution segment where no branching occurs. We utilize control flow analysis techniques to achieve this partitioning, converting the behavioral code into a control flow graph (CFG). As shown in Fig. 5(b), this represents the control flow graph for Fig. 5(a).

Visibility dependency graph construction (Line 2 in Algorithm 1): Determining the dominance relationships of various input signals under different values is important to identify which input signals truly impact the execution results of the behavioral code and which do not. To obtain the dependency relationships between different input signals across various execution paths, we extend the CFG to construct a visibility dependency graph of input signals. Fig. 5(c) illustrates the visibility dependency graph derived from Fig. 5(b). As shown, the structure of the dependency graph mirrors that of the CFG, with each node storing the input signals that are read by the corresponding execution segment. Besides, we can simplify the visibility dependency graph by removing empty nodes.

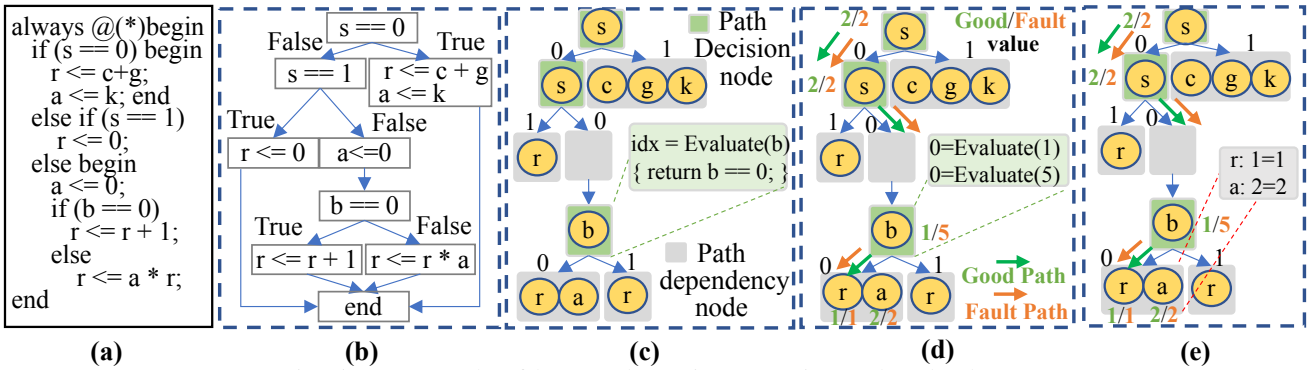

Fig. 5: An example of how to determine execution path redundancy.

Algorithm 1: Implicit Redundancy Elimination

Input : Behavior_code: A fault-free behavioral code;
fault_id: Specified the faulty behavioral code
Output: The faulty behavioral code is redundant or not

```
1  CFG ← build_control_flow_graph(Behavior_code);
2  VDG ← build_visibility_dependency_graph(CFG);
3  cur ← the entry node of VDG;
4  while cur not null do
5      if cur is path decision node then
6          next_good_node ← Evaluate(good gates);
7          next_fault_node ← Evaluate(bad gates);
8          if next_good_node ≠ next_fault_node then
9              return false;
10         end
11     end
12     if cur is path dependency node then
13         foreach signal ∈ cur do
14             if IsVisible(signal, fault_id) then
15                 return false;
16             end
17         end
18     end
19     cur ← next_node;
20 end
21 return true;
```

This graph effectively captures the varying execution paths of the behavioral code under different inputs. Moreover, by focusing on a specific execution path, we can identify which input data at any given time influences the execution results of behavioral codes. For instance, as shown in Fig. 5(c), when the input signal s assumes the value of 1, the signals c, g and k are dominated, while a and r are not. Thus, only the values of c, g, and k will affect the execution result, while the signals a and r will not. In the visibility dependency graph, the nodes are categorized into two types: path decision nodes and path dependency nodes. Path decision nodes determine the actual execution path at a given time. Each decision node contains an *Evaluate* function extracted from CFG, which assesses the current signal values to determine the appropriate sub-path. In contrast, path dependency nodes store the input signals on which a sub-path relies during execution. These signals affect the final result of the execution path.

Check on path decision point (Line 5-11 in Algorithm 1): A faulty behavioral code is considered redundant only if its execution path exactly matches that of the good behavioral code, and the input signals along the path are identical. To determine whether a faulty behavioral code is redundant, the first step is to check if its execution path aligns with that of the good behavioral code. Therefore, we need to analysis the path decision nodes. The next sub-path for the behavioral code can be determined by the output of the *Evaluate* function at the path decision node. For both the good behavioral code and the faulty behavioral code, we first retrieve their respective input signals and then invoke the *Evaluate* function to determine the subsequent execution paths (Lines 6-7 of the algorithm), as shown in Fig. 5(d) determining the sub-path for node b. If the paths at the path decision node are not consistent between the two, we conclude that the execution of the faulty behavioral code is non-redundant. Otherwise, further analysis of the subsequent paths is required.

Check on path dependency point (Line 12-18 in Algorithm 1): Under the condition that the execution paths are consistent, the agreement of the path dependency nodes affects whether the execution results of the good behavioral code and the faulty behavioral code are aligned. Therefore, we need to assess the consistency of the path dependency nodes. Since each path dependency node stores all the input signals that the behavioral code relies on for that sub-path, we must traverse these signals and check each for equality, as shown in Fig. 5(e) checking the visibility for both input signal a and r. If any input signals on this sub-path differ, we conclude that the execution results of the faulty behavioral code and the good behavioral code are inconsistent, indicating that the faulty behavioral code is non-redundant. Otherwise, further analysis of the subsequent paths is necessary.

B. Explicit execution redundancy detection

Concurrent fault simulation only simulates activated events in the fault-free network plus any differences that arise due to faults [15], which has been used to reduce explicit redundancy. So, we extend the concurrent fault simulation algorithm from

gate level to RTL to eliminate the explicit redundancy existing in behavioral nodes.

C. Simulation on the RTL Nodes

We enhance the concurrent fault simulation capabilities for all RTL nodes (logic nodes, arithmetic nodes, and others) in Iverilog. Due to Iverilog's inherent event scheduling strategy, where certain nodes are evaluated immediately upon receiving new values while others are deferred [8], issues, such as fake events, arise during the extension. Those issues may undermine the accuracy of fault simulation results. In our framework, we resolve this problem by adjusting the evaluation and scheduling relationships of the RTL nodes.

Fake events refer to the premature evaluation of bad gates on event nodes in Iverilog, due to the impact of good events, resulting in erroneous activation of waiting behavioral codes. Because of good events are sent to their successors before bad events, which causes the bad gate of event nodes to mistakenly detect an event when receiving good values, resulting in incorrect activation of waiting behavioral codes. Specifically, because good events are sent to successors before bad events, the bad gate of event nodes mistakenly detects an event when receiving good values, while the actual arrival of bad events may generate different events, resulting in incorrect activation of waiting behavioral codes. To avoid abnormal activation caused by fake events, we postpone the evaluation of all event nodes until all blocking events [16] have been processed.

V. Evaluation

A. Experimental Settings

We implement Eraser in C++ and the evaluation environment is shown in Table I. For comparison, we chose three simulators that support RTL fault simulation. The first one is a state-of-the-art commercial simulator, Z01X [17]. The second one is an open-source fault simulator [18] which is based on Verilator [19], we called VFsim for convenience. Besides, we implemented fault simulation in Icarus Verilog [8] using the *force* command, which we refer to as IFsim.

TABLE I: Evaluation Environment

Filed	Value
CPU	Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40GHz
OS	Red Hat Enterprise Linux Server 7.9(Maipo)
Compiler	gcc 11.1.0, -O3
Simulator	Z01X T-2022.06-SP2(Z01X) [18] 2021 based on Verilator(VFsim) Iverilog 12(IFsim)

Benchmark and testbench. Table II presents the benchmarks used in this paper, covering arithmetic cores [21], [22], encryption cores [23], communication controllers [20], RISCV CPUs [24], [25], MIPS CPUs [28], and accelerators [26], [27]. In our experiments, the stimuli used are either sourced from test benches provided by the design developers or written by us based on the function of benchmarks if test benches are not provided. The simulation cycles for each design are also shown in Table II.

TABLE II: Benchmark Information

Benchmark	#Stimulus	#Cells	#Faults	Fault coverage(%)	
				Eraser	Z01X
ALU (64)	1.5k	19996	1182	95.69	95.69
FPU (32)	9k	8875	1256	99.04	99.04
SHA256_HV*	2.6k	8677	660	99.85	99.85
APB	1.2k	7051	98	91.84	91.84
Sodor Core	3k	16943	1252	81.07	81.07
RISCV Mini	6k	9087	526	27.97	27.97
PicoRV32	4k	17488	1040	32.79	32.79
Conv_acc	4k	39812	1032	79.75	79.75
SHA256_C2V⋆	4k	9716	2174	99.31	99.31
MIPS CPU	1.2k	15000	1346	44.40	44.40

#Cells: Number of cells reported by the Yosys tool [29].
∗ **SHA256_HV**: the handwritten Verilog of SHA256.
⋆ **SHA256_C2V**: the Verilog generated by Chisel of SHA256.

Fault list and fault coverage. We generate stuck-at faults for wires and regs in designs and set observation points at all output ports. An observation is triggered to determine whether any faults have been detected when good events occur at observation points. As shown in Table II, the Eraser fault coverage is consistent with that of the commercial tool Z01X across all benchmarks, demonstrating that the accuracy of the fault simulation framework proposed in this paper.

B. Evaluation on Benchmarks and Comparisons

We conducted performance tests on all the aforementioned testbenches and compared them with mainstream commercial tools and the latest open-source simulators. Notably, Eraser achieved the same fault coverage as commercial tools, as shown in TABLE II, demonstrating the correctness of our algorithm. The experimental results are shown in Fig. 6, which includes the algorithm's execution time and corresponding speedup ratios. As observed, Eraser achieved the best performance across all benchmark circuits except for SHA256_C2V. For example, in the arithmetic unit ALU, the execution times of IFsim, VFsim, Z01X, and Eraser are 5.9s, 1.2s, 2s, and 0.3s, respectively. Compared to the baseline IFsim, they achieved speedup ratios of 4.9×, 3.0×, and 19.7×, respectively. For the floating-point unit FPU, VFsim, Z01X, and Eraser achieved speedup ratios of 7.8×, 27.7×, and 41.9× compared to IFsim.

Eraser outperforms the commercial tool Z01X on most benchmarks, especially on the APB benchmark for a speedup of 10.0× and on the ALU benchmark for a speedup of 6.7×. However, in the case of the SHA256_C2V benchmark, the performance of Eraser is inferior to that of Z01X. The reason can be attributed to two aspects. First, the execution time of the behavioral codes in SHA256_C2V accounts for only 1% of the total execution time, which significantly limits the effectiveness of optimizations for redundant execution of behavioral nodes. Second, the commercial Z01X employs various engineering optimizations for node execution that Eraser has not yet utilized. We plan to incorporate these optimizations in future work to further enhance Eraser's performance. On average, Eraser achieves a 3.9× speedup compared to commercial tool and a 5.9× speedup compared to the latest open-source tools.

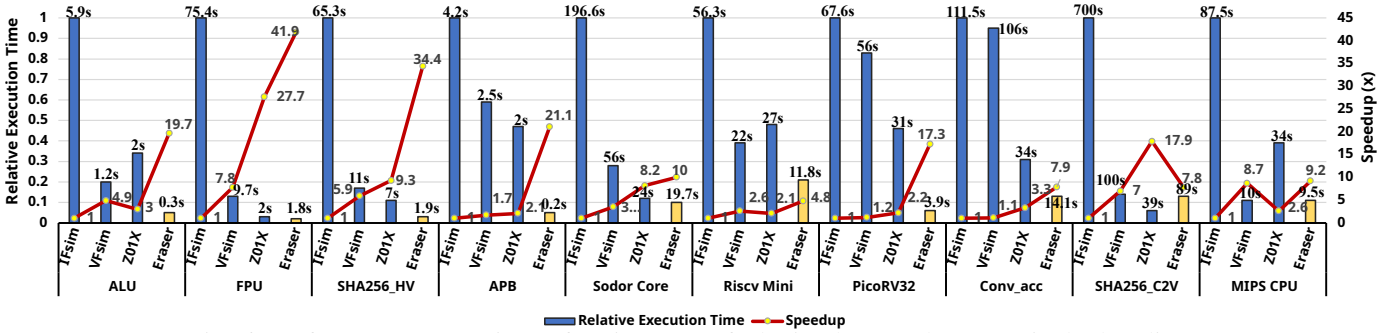

Fig. 6: Performance comparison of various RTL fault simulators. The *IFsim* is the baseline.

C. *Ablation study*

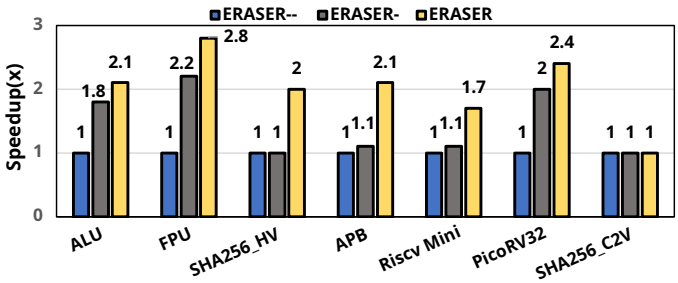

Fig. 7: Ablation Study on Redundancy Elimination: Eraser-- (No Redundancy Elimination), Eraser- (Eliminates Explicit Redundancy), Eraser (Eliminates Both Explicit and Implicit Redundancy)

We conducted an ablation study to demonstrate the acceleration effect of implicit redundancy elimination on RTL fault simulation. We implement another two versions of Eraser: Eraser--, which does not detect any redundancy; Eraser-, which detects only explicit redundancy. Aside from redundancy detection, they have the same implementation as Eraser. In Fig. 7, we present the speedup across various circuits. The comparison shows that Eraser achieves significant speedup over both Eraser-- and Eraser- for most circuits. For example, for the FPU circuit, Eraser achieves a 2.8× speedup over Eraser-- and a 1.3× speedup over Eraser-. Similarly, for the SHA256_HV implementation, Eraser achieves a 2.0× speedup over Eraser-- and a 2.0× speedup over Eraser-. However, for SHA256_HV, Eraser shows similar performance with Eraser- and Eraser–. This is because, compared to the Verilog version, the Chisel implementation contains many RTL nodes, resulting in very few implicit and explicit redundancies in behavioral nodes.

Table III further illustrates the proportion of redundant executions in these circuits. We list the proportion of runtime for behavioral nodes, the total number of behavioral node executions (without eliminating any redundant executions), the total number of redundant executions, and portions of explicit and implicit redundancies. As shown in Table III, in circuits such as SHA256_HV, APB, and RISCV Mini, implicit redundancies account for more than 70% of the redundancy behavioral node executions. Correspondingly, Eraser demonstrates significant performance improvements over Eraser- and Eraser-- in these circuits. Meanwhile, implicit redundancies in the PicoRV32 only account for 13%, which explains the limited performance gain of Eraser over Eraser- in Fig. 7. Additionally, in SHA256_C2V, the overhead of behavioral node simulation accounts for only 1% of the total, making the optimization of behavioral node execution less impactful. As a result, Eraser- and Eraser show performance similar to Eraser--. On average, across these circuits, the proportions of implicit and explicit redundancies are both around 45%.

TABLE III: Proportion of Redundant Behavioral Node Executions

Benchmark	Time for BN(%)	#Total BN Execution	#Elimination	Explicit(%)	Implicit(%)
ALU	57	339592	324714	82	14
FPU	70	1891740	1793457	81	14
SHA256_HV	70	992540	862612	1	86
APB	74	211000	180650	15	70
RISCV Mini	53	2779987	2650970	11	84
PicoRV32	61	5701568	5650319	86	13
SHA256_C2V	1	834539	634533	49	27
Average		-		**46**	**44**

The above analysis demonstrates that implicit redundancy constitutes a significant part of the execution in RTL fault simulations. By eliminating it, Eraser achieves noticeable performance improvements.

VI. Conclusion

In this work, we identify implicit redundancy in the simulation of behavioral nodes in RTL fault simulation and propose a redundancy detection algorithm based on execution paths at runtime and data dependencies to eliminate it. We further implement an efficient RTL fault simulation framework that trimmed the redundancy. The experimental results show that compared to the commercial simulator and an open-source fault simulator, our simulator achieves an average acceleration of 3.9× and 5.9×, respectively.

Acknowledgment

This paper is supported in part by the National Natural Science Foundation of China (NSFC) under grant No. (92473203, 92373206), and in part by the Chinese Academy of Sciences under grant No. XDB0660100. The corresponding authors are Jianan Mu and Huawei Li.

REFERENCES

[1] ISO. ISO 26262. Road vehicles - Functional Safety - Part 4: Product development at the system level

[2] N. Saxena and A. Lotfi, "Error Model (EM)—A New Way of Doing Fault Simulation," 2022 IEEE International Test Conference (ITC), Anaheim, CA, USA, 2022, pp. 324-333, doi: 10.1109/ITC50671.2022.00040.

[3] J. E. R. Condia, J. -D. Guerrero-Balaguera, F. F. Dos Santos, M. S. Reorda and P. Rech, "A Multi-level Approach to Evaluate the Impact of GPU Permanent Faults on CNN's Reliability," 2022 IEEE International Test Conference (ITC), Anaheim, CA, USA, 2022, pp. 278-287, doi: 10.1109/ITC50671.2022.00036.

[4] J. -C. Baraza, J. Gracia, S. Blanc, D. Gil and P. -J. Gil, "Enhancement of Fault Injection Techniques Based on the Modification of VHDL Code," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 16, no. 6, pp. 693-706, June 2008, doi: 10.1109/TVLSI.2008.2000254.

[5] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson and J. Karlsson, "Fault injection into VHDL models: the MEFISTO tool," Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing, Austin, TX, USA, 1994, pp. 66-75, doi: 10.1109/FTCS.1994.315656.

[6] V. Sieh, O. Tschache and F. Balbach, "VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions," Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing, Seattle, WA, USA, 1997, pp. 32-36, doi: 10.1109/FTCS.1997.614074.

[7] Synopsys inc. [n. d.]. VCS. https://www.synopsys.com/verification/ simulation/vcs.html

[8] Stephen Williams. Iverilog. https://github.com/steveicarus/iverilog

[9] N. Bombieri, F. Fummi and V. Guarnieri, "Accelerating RTL Fault Simulation through RTL-to-TLM Abstraction," 2011 Sixteenth IEEE European Test Symposium, Trondheim, Norway, 2011, pp. 117-122, doi: 10.1109/ETS.2011.58.

[10] N. Bombieri, F. Fummi and V. Guarnieri, "Accelerating RTL Fault Simulation through RTL-to-TLM Abstraction," 2011 Sixteenth IEEE European Test Symposium, Trondheim, Norway, 2011, pp. 117-122, doi: 10.1109/ETS.2011.58.

[11] D. Mueller-Gritschneder, U. Sharif and U. Schlichtmann, "Performance and Accuracy in Soft-Error Resilience Evaluation using the Multi-Level Processor Simulator ETISS-ML," 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Diego, CA, USA, 2018, pp. 1-8, doi: 10.1145/3240765.3243490.

[12] E. Kaja, N. O. Leon, M. Werner, B. Andrei-Tabacaru, K. Devarajegowda and W. Ecker, "Extending Verilator to Enable Fault Simulation," MBMV 2021; 24th Workshop, online, 2021, pp. 1-6.

[13] Johannes Geier and Daniel Mueller-Gritschneder. 2023. VRTLmod: An LLVM based Open-source Tool to Enable Fault Injection in Verilator RTL Simulations. In Proceedings of the 20th ACM International Conference on Computing Frontiers (CF '23). Association for Computing Machinery, New York, NY, USA, 387–388. https://doi.org/10.1145/3587135.3591435

[14] S. Gai, P. L. Montessoro and F. Somenzi, "MOZART: a concurrent multilevel simulator," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 7, no. 9, pp. 1005-1016, Sept. 1988, doi: 10.1109/43.7799.

[15] K. P. Lentz and J. B. Homer, "Handling behavioral components in multi-level concurrent fault simulation," Proceedings 33rd Annual Simulation Symposium (SS 2000), Washington, DC, USA, 2000, pp. 149-156, doi: 10.1109/SIMSYM.2000.844911.

[16] "IEEE Standard for Verilog Hardware Description Language," in IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001) , vol., no., pp.1-590, 7 April 2006, doi: 10.1109/IEEESTD.2006.99495.

[17] Synopsys inc. [n. d.]. Z01X. https://www.synopsys.com/verification/ simulation/vc-z01x.html.

[18] E. Kaja, N. O. Leon, M. Werner, B. Andrei-Tabacaru, K. Devarajegowda and W. Ecker, "Extending Verilator to Enable Fault Simulation," MBMV 2021; 24th Workshop, online, 2021, pp. 1-6.

[19] Veripool. [n. d.]. Veripool. https://www.veripool.org/verilator/

[20] OpenCores. [n. d.]. OpenCores. https://opencores.org/

[21] FreeCores. [n. d.]. FreeCores. https://github.com/freecores/verilog_fixed _point_math_library

[22] Kimia Noorbakhsh. [n. d.]. FPU. https://github.com/kimianoorbakhsh/Verilog-Matrix-Multiplier

[23] secworks. [n. d.]. sha256. https://github.com/secworks/sha256

[24] UC Berkeley Architecture Research. ucb-bar. https://github.com/ucb-bar

[25] YosysHQ. [n. d.]. picorv32. https://github.com/YosysHQ/picorv32

[26] Anish Singhani. crypto-accelerator. https://github.com/asinghani/crypto-accelerator

[27] liyu-ao. [n. d.]. LeNet-accelerator. https://github.com/liyu-ao/PE-array-for-LeNet-accelerator-based-on-FPGA/tree/main

[28] Jeremiah Mahler. [n. d.]. mips-cpu. https://github.com/jmahler/mips-cpu

[29] Yosys Headquarters. [n. d.].Yosys: A Framework for Verilog RTL Synthesis. http://www.clifford.at/yosys/

[30] N. Saxena and A. Lotfi, "Error Model (EM)—A New Way of Doing Fault Simulation," 2022 IEEE International Test Conference (ITC), Anaheim, CA, USA, 2022, pp. 324-333, doi: 10.1109/ITC50671.2022.00040.

[31] J. E. R. Condia, J. -D. Guerrero-Balaguera, F. F. Dos Santos, M. S. Reorda and P. Rech, "A Multi-level Approach to Evaluate the Impact of GPU Permanent Faults on CNN's Reliability," 2022 IEEE International Test Conference (ITC), Anaheim, CA, USA, 2022, pp. 278-287, doi: 10.1109/ITC50671.2022.00036.

[32] Mahyar Emami, Sahand Kashani, Keisuke Kamahori, Mohammad Sepehr Pourghannad, Ritik Raj, and James R. Larus. 2024. Manticore: Hardware-Accelerated RTL Simulation with Static Bulk-Synchronous Parallelism. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23). Association for Computing Machinery, New York, NY, USA, 219–237. https://doi.org/10.1145/3623278.3624750