Efficient Coverage Optimization with Formal-Guided Testcase Generation in UVM Verification

Yu-Shien Shen[1], Yean-Ru Chen[1*], Yu-Tung Chen[1], En-Hsiang Lin[2]
[1]Department of Electrical Engineering, National Cheng Kung University, Taiwan (R.O.C.)
[2]Realtek Semiconductor Corp., Taiwan (R.O.C.)
Email: chenyr@mail.ncku.edu.tw*

***Abstract*—Processor verification faces significant challenges in state-space explosion and test coverage limitations, particularly in complex micro-architectures. Formal verification provides precise correctness guarantees but is constrained by computational overhead and scalability issues. Conversely, simulation-based approaches, including constrained-random verification and fuzz testing, provide scalability but often lack systematic guidance to effectively cover critical design regions and rarely exercised state transitions. To overcome these challenges, we propose *Formal-Guided Test Sequence Optimization* (FGTSO), a framework that integrates formal verification with simulation to systematically target coverage holes and enhance verification efficiency. FGTSO mitigates false alarms by refining formal assumptions and resolving black-box (BBOX) limitations through abstraction modeling. By continuously aligning formal and simulation environments, FGTSO reduces test redundancy while enabling precise corner-case exploration. This approach enhances verification completeness, efficiently covering hard-to-reach design behaviors that traditional methodologies often overlook. Experimental results on the CVA6 RISC-V core show that FGTSO achieved 99.91% branch coverage within 8 days, which is 4.41% higher than HyPFuzz's 95.5%, effectively covering 98% of the previously uncovered regions. Furthermore, within 10 days, FGTSO achieved 100% coverage across all key metrics, including line, toggle, condition, and branch coverage. These results validate FGTSO's ability to identify complex corner-case behaviors that traditional methods fail to reach, significantly enhancing verification completeness and efficiency.**

***Index Terms*—Processor Verification, Formal Verification, Simulation, Test Sequence Optimization, Universal Verification Methodology**

I. Introduction

As hardware designs grow increasingly complex, the verification process faces significant challenges. Existing methods, such as formal verification and simulation, can detect certain vulnerabilities, but each has limitations in terms of efficiency and coverage. Formal verification provides comprehensive design exploration, but encounters the problem of state-space explosion when applied to large-scale designs. On the other hand, simulation offers high scalability and broad applicability, and its coverage can be improved using constrained-random verification (CRV). However, its effectiveness heavily depends on well-designed constraints; biased constraints may miss critical scenarios, while overly strict ones can increase debugging complexity.

To maximize the strengths of simulation and formal verification while mitigating their limitations, we propose Formal-Guided Test Sequence Optimization (FGTSO) to enhance coverage and verification efficiency. This methodology leverages simulation's scalability and formal verification's precision to efficiently target critical conditions, reducing simulation overhead while improving coverage. Key challenges include resolving false alarms from formal-simulation discrepancies and addressing BBOX constraints in high-storage or high-computation modules. Our framework mitigates these issues, ensuring a more comprehensive and efficient verification methodology. By integrating simulation and formal verification, our proposed framework improves coverage, reduces redundancy, and enhances verification accuracy.

Our contribution is as follows:

1) We propose a verification framework that integrates formal verification with simulation, utilizing the FGTSO technique. This framework reduces false alarms through environment alignment and addresses the black-box (BBOX) challenges that are often overlooked or inadequately handled in related works. We introduce abstract models to overcome the limitations of BBOX in formal verification, particularly for components with high storage requirements (e.g., memory) and complex computational units (e.g., multipliers, floating-point units), thereby significantly enhancing simulation coverage in areas related to BBOX modules.
2) Applied to the RISC-V CVA6 processor [9], our framework achieves 99.38% branch coverage in 3 days (4.6% higher than HyPFuzz [3]) and 99.91% in 8 days (4.41% higher than HyPFuzz). Ultimately, within 10 days, it reaches 100% coverage across all key metrics, demonstrating superior efficiency and completeness.

The remaining organization of this paper is as follows. Section II discusses related work; Section III introduces our proposed methodology, FGTSO; Section IV presents a case study; Section V compares the experimental results of our proposed methodology with those of the related work. Finally, Section VI concludes this work.

II. Related Work

In hardware verification, tools like RISC-V Torture [5] and RISCV-DV [4] have played a crucial role in verifying the basic functional correctness of RISC-V processors. RISC-V Torture generates random instruction sequences to achieve faster coverage closure, making it suitable for general functional validation. However, it lacks structured verification guidance, limiting its effectiveness in verifying specific functionalities or extreme corner cases. RISCV-DV, developed by Google, extends these capabilities by supporting CRV and privileged ISA features, offering greater flexibility. Nevertheless, its reliance on randomness makes it difficult to systematically cover boundary conditions, and the test results are often unstable, making it challenging to reproduce specific errors. Despite their effectiveness, both approaches lack systematic methodologies to ensure comprehensive coverage of critical design aspects, particularly in complex control flows where formal guidance becomes essential.

Fuzzing-based tools like TheHuzz [6] and HyPFuzz [3] have further advanced processor verification by leveraging golden-reference models and hybrid approaches, respectively. TheHuzz adopts fuzzing techniques to detect processor vulnerabilities and can effectively generate diverse execution paths. However, it primarily relies on a large number of random or semi-random inputs to trigger abnormal behaviors, which cannot guarantee coverage of all possible states or paths. HyPFuzz combines fuzzing with formal verification to accelerate coverage and uncover previously undetected vulnerabilities,

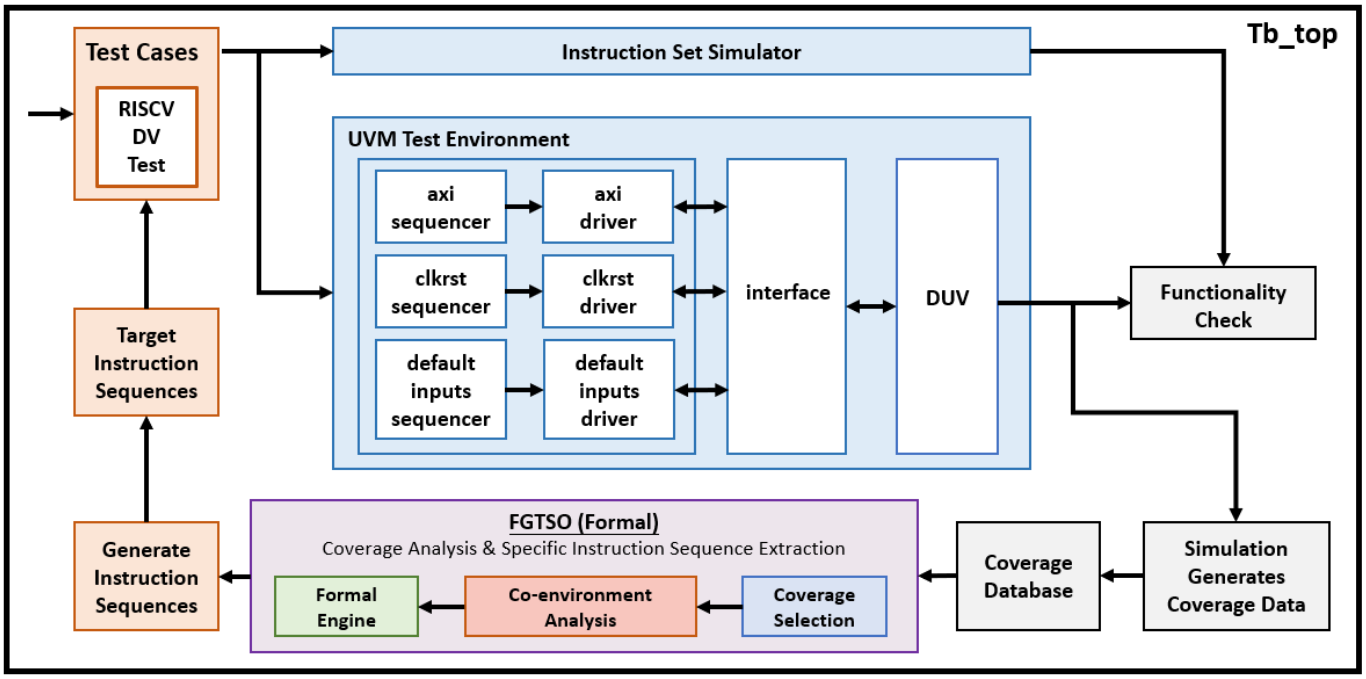

Fig. 1. Overview of formal-guided simulation workflow

including new Common Vulnerabilities and Exposures (CVEs). While this hybrid approach significantly improves verification efficiency, it does not fully address two major challenges in formal verification: false alarms due to environmental mismatches and BBOX issues caused by excessive computational complexity or large storage modules. These issues highlight the need for a more structured framework that not only accelerates coverage but also ensures formal verification consistency and completeness.

III. Methodology

A. Framework Overview

This paper presents a formal-guided simulation-based verification framework (Fig. 1) that integrates constrained-random test generation, UVM-based simulation, functional checks, and targeted instruction sequence optimization to enhance processor verification coverage and efficiency. The process begins with RISCV-DV generating randomized test cases, ensuring broad verification by covering diverse instruction combinations. These test cases are then simulated within a UVM environment and executed on the processor's Design Under Verification (DUV). The results are cross-verified with an Instruction Set Simulator (ISS) to ensure functional correctness. Among them, only the FGTSO part (highlighted in purple in Fig. 1) corresponds to formal verification, while the remaining processes belong to the simulation-based execution environment.

During simulation, the simulator generates coverage reports that capture the executed instructions, conditions, and data flow states within the design. These reports are systematically analyzed by our proposed FGTSO, which selects coverage holes for further refinement. Using formal verification, FGTSO can precisely analyze the identified coverage hole and generates targeted instruction sequences to address the coverage hole. The optimized sequences are iteratively fed back into the verification process, continuously refining the verification dataset and enhancing verification coverage.

This paper introduces a verification optimization process based on FGTSO, as illustrated in Fig. 2, designed to effectively generate instruction sequences that address coverage holes in processor designs. By leveraging formal verification to guide simulation, FGTSO optimizes the process to produce targeted instruction sequences, enabling efficient exploration and comprehensive coverage of the design space.

The FGTSO process consists of four key steps:

1) **Coverage Hole Selection:** The process begins by identifying and prioritizing coverage holes that have the most significant impact on verification efficiency. This ensures that the optimization efforts firstly focus on the most efficient hole in coverage.
2) **Environment Consistency Check:** Since false alarms often arise due to inconsistencies between formal and simulation environments, this step aligns formal and simulation by applying *assume* properties in SystemVerilog Assertion (SVA) [1] during formal verification and modifying the existing constraint in RISCV-DV accordingly. Additionally, this step addresses BBOX issues by constructing the abstraction model, ensuring that the generated instruction sequences are more effective in subsequent steps.
3) **Formal Cover Property Generation:** Once the environments are aligned, formal *cover* properties in SVA are defined to precisely characterize the behavior of the selected coverage hole, enabling the generation of targeted instruction sequences for that specific coverage hole.
4) **Targeted Instruction Sequence Extraction:** Using the defined formal *cover* property, FGTSO generates a cover trace (i.e., evidence waveform), which is then translated into target instruction sequences. These sequences are iteratively fed back into the simulation environment, continuously improving coverage and strengthening the overall verification process.

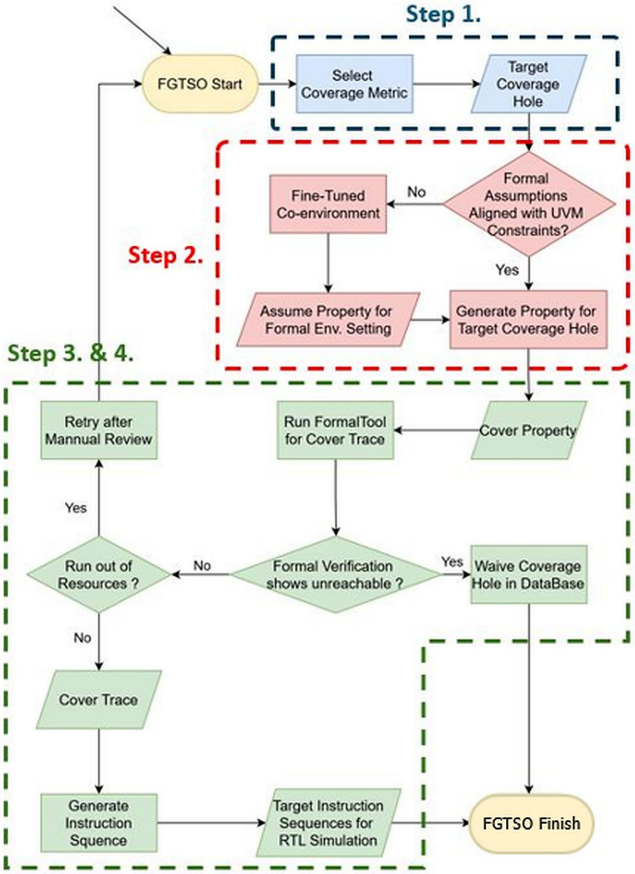

Fig. 2. Workflow of FGTSO

B. Formal-Guided Test Sequence Optimization

1) Step 1 of FGTSO (Blue Part)

To ensure efficient allocation of verification resources and maximize coverage improvement, we need a strategy that prioritizes addressing the most impactful coverage holes. By focusing on coverage holes that significantly affect verification efficiency, this approach optimizes resource utilization and accelerates the verification process.

Specifically, we employ the MaxUncovd strategy, a heuristic method derived from the work of Chen et al. [3]. This strategy prioritizes selecting modules with the lowest coverage, under the assumption that addressing the most uncovered areas yields the greatest verification impact. By systematically targeting critical coverage deficiencies, MaxUncovd enhances the overall verification efficiency and ensures comprehensive design verification. Moreover, this step does not introduce additional analysis overhead, as the tool can directly rank coverage data.

2) Step 2 of FGTSO (Red Part)

Aligning the formal verification environment with the simulation environment is a critical step. In our proposed methodology, misalignment detection is performed to determine whether the two environments are consistent. First, formal verification generates instruction sequences targeting specific coverage holes, which are then executed in the simulation environment. If the results from formal verification and simulation are inconsistent—that is, the instruction sequences generated by formal verification fail to effectively cover the intended coverage holes during simulation—it indicates a misalignment between the two environments. There are three main causes of inconsistency between the formal verification and simulation environments: primary inputs (PIs), BBOX, and insufficient constraint definitions in RISCV-DV. The following sections will describe the handling methods for each of these causes of environment inconsistency.

a. Primary Input:

In formal verification, PIs are treated as free nets, allowing them to assume arbitrary values without constraints. This can lead to incorrect coverage analysis, as generated cover traces may fail to address coverage holes in simulation, preventing effective coverage closure and compromising verification accuracy.

Furthermore, unconstrained PIs can result in misleading coverage estimation, especially when certain test scenarios correspond to unreachable states by design. For instance, if AXI [7] protocol behavior is not properly described, the rvalid signal may be asserted before arvalid and arready, violating realistic AXI protocol and memory access sequences. Consequently, formal verification may fail to generate meaningful instruction patterns, leaving specific coverage holes unresolved.

```
r_valid: assume property (
(req.ar_valid && req.ar_ready) |=>  (resp.r_valid)
);
```

Fig. 3. Formal assumption for AXI response

To address this, formal assumptions must be introduced to align formal verification with realistic system behavior, as shown in Fig. 3. This reduces false alarms, enhances verification efficiency, and ensures resources are allocated to valid coverage goals, ultimately improving overall verification quality.

b. BBOX:

In most commercial Electronic Design Automation (EDA) tools, to ensure the feasibility of formal verification, modules with high storage requirements or complex computational logic are often treated as BBOX. Since BBOX modules cannot correctly drive the outputs, they may become free nets, which can lead to false alarms. Additionally, this limitation prevents the internal logic of the module from driving its outputs, resulting in verification gaps and reduced simulation coverage. To address this challenge, we introduce abstraction models that preserve the critical behavior of BBOX modules while ensuring their interaction with the rest of the system remains correctly verifiable. This paper introduces how we handle modules with high storage requirements or complex computational logic when they are treated as black boxes.

- For high storage requirements module:
  When handling large storage components like caches, tracking all possible tags incurs excessive storage overhead, often leading to the module being treated as a BBOX. However, cache tags are crucial for determining hit/miss behavior. To address this, we introduce a functional abstraction model that simplifies verification complexity while preserving key behaviors.
  For example, as shown in Fig. 4, a predefined tag replaces full tag tracking, significantly reducing storage requirements.
  The abstract model is constructed by using an SVA array tracks cache index accesses, marking indices as valid upon first access to ensure efficient logging. A single predefined tag is enforced for all accesses, reducing overhead while preserving cache functionality. By adhering to this abstraction, verification complexity is reduced without compromising coverage accuracy.

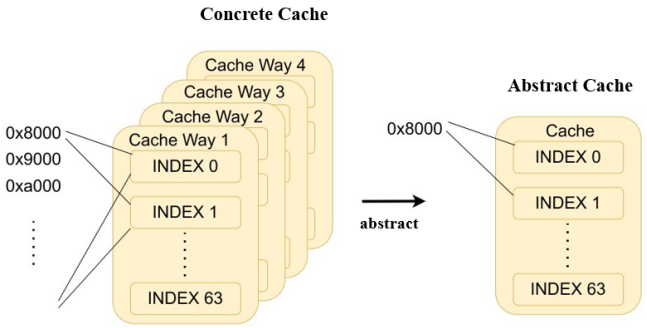

Fig. 4. Cache abstraction model concept

- For complex computational logic module:
  Computationally intensive components, such as multipliers, are also often treated as BBOX in most commercial formal verification tools due to overloading the verification complexity, resulting in arbitrary outputs and unverifiable conditions. To conquer this issue, we introduce a symbolic computation approach using SVA to approximate expected outputs and enhance verification accuracy.
  For instance, as shown in Fig. 5, in a multiplier, SVA captures input signals to trace the source of computation, performs a simplified symbolic multiplication, and uses the result as the output to ensure consistency between symbolic and actual computations.

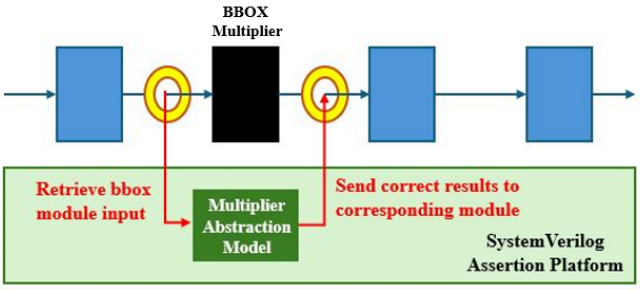

Fig. 5. Multiplier abstraction model concept

c. Insufficient RISCV-DV Constraint:
Although formal verification can generate valid coverage evidence, inconsistencies may still occur during simulation. This is because formal verification may produce specialized memory data sequences that the existing constraint definitions in RISCV-DV cannot properly handle. For example, in the case of specific page table data sequences, the SV39 multi-level page table mechanism includes 1 GB, 2 MB, and 4 KB page tables. However, due to the original constraint definitions in RISCV-DV, we are only able to access the 1GB page table. To resolve this issue, we modify the constraints in RISCV-DV related to page table entries. As shown in Fig.6, when accessing a 4 KB page table, we apply constraints to the addresses pointed to by the 1 GB and 2 MB page table entries and set the X, W, and R bits to 0, indicating that these entries should point to the next level of the page table. With these adjustments, the original coverage hole is able to be successfully filled.

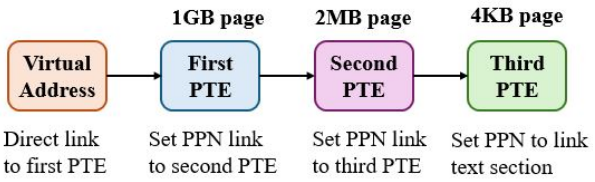

Fig. 6. Our proposed RISCV-DV's page constraint

3) Step 3 & 4 of FGTSO (Green Part)

After aligning the verification environments, generating instruction sequences that effectively address target coverage holes is crucial. Our solution uses formal cover properties to obtain cover traces, guiding instruction sequence generation for improved verification coverage.

First, we use formal cover properties to define the behavior of the selected coverage hole. This step automatically extracts the coverage hole expression and applies it to predefined cover templates. Next, the formal tool executes the cover property, derives the input patterns required to satisfy the condition, and generates the corresponding cover trace. Finally, based on the generated cover trace, we extract the instruction sequence to effectively cover the coverage hole, while improving test efficiency and verification completeness. However, during the instruction sequence generation process, the formal tool may encounter two key challenges as follows.

a. Unreachable Coverage Holes:
If the formal verification tool reports a coverage hole as unreachable, we first review whether the constraints in both the formal and simulation environments are overly restrictive. If the constraints are appropriate and the coverage hole is indeed unreachable, it is waived; however, the information is still provided to designers as a reference for debugging. Waiving is simply to avoid unnecessary resource waste in covering such holes.

b. Resource Exhaustion:
One of the primary limitations of formal tools is state space explosion, which is particularly problematic for large-scale designs. If the verification process exhausts computational resources, we decompose complex properties into smaller sub-properties, collect cover traces for each sub-property, and ultimately integrate them into a comprehensive instruction sequence. This approach ensures manageable complexity while maintaining verification workability.

By systematically addressing these challenges, our methodology enhances the effectiveness of coverage-driven test generation, ensuring that targeted instruction sequences can efficiently bridge coverage holes without unnecessary resource consumption.

Fig. 7. FGTSO step 1: MaxUncovd automatically selects coverage holes

```
615  asm_tag_dcache_idle_0: assume property
616  (
617      if (valid_tag_dcache[gen_cache_wt.i_cache_subsystem.i_wt_dcache.i_wt_dcache_mem.vld_addr])
618      ##1 gen_cache_wt.i_cache_subsystem.i_wt_dcache.i_wt_dcache_mem.gen_tag_srams[0].i_tag_sram.rdata_o == 'h100000080000
619      else
620      ##1 gen_cache_wt.i_cache_subsystem.i_wt_dcache.i_wt_dcache_mem.gen_tag_srams[0].i_tag_sram.rdata_o == 'b0
621  );
```

Fig. 8. FGTSO step 2: constructing cache abstraction model

```
623  test1: cover property
624  (
625      gen_cache_wt.i_cache_subsystem.i_wt_dcache.gen_rd_ports[1].genblk1.i_wt_dcache_ctrl.state_q == 'h2 &&
626      ~gen_cache_wt.i_cache_subsystem.i_wt_dcache.gen_rd_ports[1].genblk1.i_wt_dcache_ctrl.req_port_i.kill_req &&
627      gen_cache_wt.i_cache_subsystem.i_wt_dcache.gen_rd_ports[1].genblk1.i_wt_dcache_ctrl.miss_replay_i
628  );
```

Fig. 9. FGTSO step 3: formal cover property for simulation instruction sequence generation

```
main:
    addiw x10, x10, -27
    ld x24, 0(x2)
    slli x8, x8, 8
    addi x8, x2, 0
    csrrw x16, 0x7c1, x2
    lh x15, 4(x2)
    lw x9, 0(x8)
    lhu x3, 1192(x24)
    csrrsi x8, 0x7c1, 3
    lb x28, 1536(x9)
    beq x1, x2, next13632

next13632:
    slli x8, x8, 8
    lbu x0, -2047(x15)
```

```
179      MISS_REQ: begin
180        miss_req_o = 1'b1;
181
182        if (req_port_i.kill_req) begin
183          req_port_o.data_rvalid = 1'b1;
184          if (miss_ack_i) begin
185            state_d = KILL_MISS;
186          end else begin
187            state_d = KILL_MISS_ACK;
188          end
189        end else if (miss_replay_i) begin
190          state_d = REPLAY_REQ;
191        end else if (miss_ack_i) begin
192          state_d = MISS_WAIT;
193        end
```

Fig. 10. FGTSO step 4: generated simulation instruction sequence

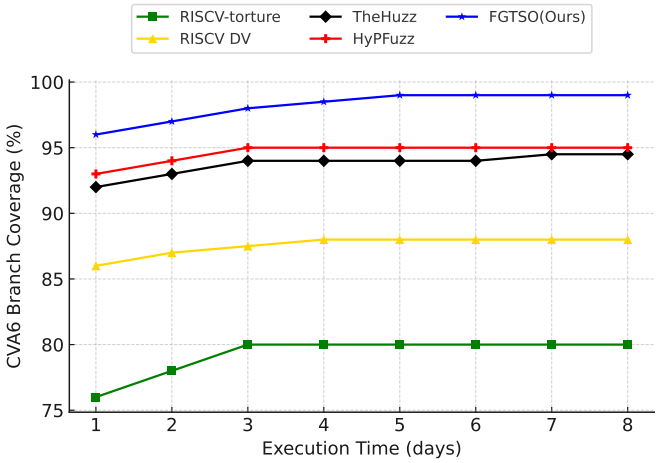

Fig. 11. Branch coverage achieved across execution times

IV. Case Study

This example illustrates how FGTSO systematically identifies, analyzes, and resolves a specific simulation coverage hole. After completing a simulation run, coverage data is collected directly from the simulator. Using the MaxUncovd strategy, the most critical coverage hole is automatically selected without additional manual effort, as the simulator is capable of sorting coverage metrics by their priority. The first step is shown in Fig. 7, the identified coverage hole corresponds to a branch condition in the cache control module.

This coverage hole involves the cache subsystem, which introduces challenges due to the presence of a BBOX cache model (Fig. 8) in formal verification. To resolve this, we construct an abstraction model for the cache, preserving only behaviors relevant to the target property. Once the environments are aligned, we propose a formal cover property that explicitly describes the target coverage condition (Fig. 9). With assumptions and abstraction in place, the formal tool successfully executes the cover property and generates a corresponding cover trace. From this trace, we extract the associated instruction sequence, capturing the precise conditions require to trigger the uncovered behavior. This instruction sequence is then embedded into a test case generated by RISCV-DV (Fig. 10). Upon simulation, the target coverage hole is successfully resolved, validating the correctness and effectiveness of the generated sequence. This case highlights the seamless integration of formal and simulation-based techniques within our framework, enabling efficient and scalable coverage closure.

V. Experiment

In this experiment, we evaluate the branch coverage achieved by different verification tools, including RISC-V Torture [5], RISCV-DV [4], TheHuzz [6], HyPFuzz [3], and our proposed FGTSO-based method. The DUV used for this study is the CVA6 RISC-V core [9]. To conduct a comprehensive evaluation, we utilize Cadence Jasper [2] as the formal verification tool and Synopsys VCS [8] as the simulation platform. The experiments are executed on a system equipped with an Intel(R) Xeon(R) Gold 5118 CPU @ 2.30 GHz 48-core Processor and 251 GB DDR4 RAM.

To assess both short-term and extended performance in achieving branch coverage, the experiments are conducted over two time spans: three days and eight days, as shown in Fig. 11. For TheHuzz and HyPFuzz, the experimental data are obtained from HyPFuzz's publications, while the data for RISC-V Torture and RISCV-DV are derived from their open-source implementations, with features configured exactly as provided on their official websites without any additional modifications. In the case of RISC-V Torture and RISCV-DV, the generated test cases are compiled into binary files and fed into the CVA6 UVM environment for simulation to measure the experimental outcomes. Notably, this CVA6 UVM environment is identical to the one employed in our proposed methodology, ensuring a consistent basis for comparison across all evaluated techniques.

Fig. 12. FGTSO's achievements of 100% simulation coverage in CVA6

To evaluate verification efficiency, we compare branch coverage achieved within a 3-day execution period. RISC-V Torture reaches 79.47%, relying on random testing, which struggles to cover complex design areas. RISCV-DV attains 94.19% by leveraging CRV for more diverse test sequences. TheHuzz achieves 88.68%, effective in bug detection but lacking systematic verification. HyPFuzz reaches 94.78%, benefiting from formal-guided simulation but limited by test sequence effectiveness. In contrast, our FGTSO can achieve the highest coverage at 99.38%, aligning formal verification with simulation to generate targeted test sequences.

To assess long-term performance, we extend the evaluation to 8 days. RISC-V Torture shows minimal improvement at 79.64%, confirming its limitations. RISCV-DV plateaus at 94.20%, restricted by predefined constraints. TheHuzz grows slightly to 90.00%, hindered by the absence of systematic guidance. HyPFuzz reaches 95.50%, though its improvement slows over time. FGTSO continues to refine coverage, achieving 99.91%, outperforming all methods in extended verification.

To assess the effectiveness of FGTSO, we analyze its final verification coverage results. FGTSO can achieve 100% coverage across all critical verification metrics, including Line, Toggle, Condition, and Branch Coverage, as shown in Fig. 12. Its success stems from a formal-guided test generation approach that aligns formal verification with simulation, refining test data iteratively while eliminating redundancies. Additionally, FGTSO addresses BBOX verification challenges by incorporating abstraction models, ensuring comprehensive validation of all design components. These results confirm FGTSO's ability to enhance coverage efficiency and guarantee verification completeness.

VI. Conclusion

This paper presents a verification framework that successfully integrates the precision of formal verification in covering simulation coverage holes with the scalability of simulation. By addressing key challenges in formal-guided simulation, including false alarms and BBOX handling, FGTSO enhances verification completeness while reducing redundancy and manual intervention, surpassing traditional approaches. In terms of performance, FGTSO demonstrates superior efficiency in both simulation coverage and verification effectiveness. Compared to HyPFuzz, our framework achieves a 98% improvement in the coverage of previously uncovered regions, highlighting its ability to more effectively identify hard-to-reach areas in the design. Furthermore, within 10 days, FGTSO ensures comprehensive validation by achieving 100% coverage across all critical verification metrics, including Line, Toggle, Condition, and Branch Coverage. This guarantees a thorough verification of the target processor design, significantly reducing blind spots and potential vulnerabilities.

VII. Acknowledgment

This research was supported by Realtek Semiconductor Corporation and the National Science and Technology Council (NSTC), Taiwan (Project Nos. NSTC 113-2640-E-006-002 and NSTC 113-2221-E-006-128), with license support provided by Cadence Design Systems and the Taiwan Semiconductor Research Institute (TSRI).

References

[1] IEEE Standards Association. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)*, pages 1–1354, 2024.

[2] Cadence Design Systems. Cadence Jasper™ Formal Verification Platform, 2024. Accessed: 2025-01-23.

[3] Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. HyPFuzz:Formal-Assisted processor fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1361–1378, 2023.

[4] Google. RISC-V DV: A Random Instruction Generator for RISC-V Verification. https://github.com/google/riscv-dv, 2019. Accessed: 2025-01-23.

[5] UC Berkeley Architecture Research Group. Risc-v torture: Random instruction set testing tool for risc-v. https://github.com/ucb-bar/riscv-torture, 2017. Accessed: 2025-01-23.

[6] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3219–3236, 2022.

[7] ARM Ltd. *AMBA AXI and ACE Protocol Specification*, 2022.

[8] Synopsys. Vcs functional verification solution, 2024. Accessed: 2025-01-23.

[9] Florian Zaruba and Luca Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, 2019.