Optimizing Functional Fault Grading Flow for Memory Designs

Euisang Yoon[*], Arun Gogineni[*], Saurabh Srivastava[*], Eunjong Oh[†], Seongwook Lee[†], Sungyun Yoo[*], Geonbeom Kwon[*], Yoseop Lee[†], Hyojin Choi[†]

[†]Samsung Electronics Co., Ltd., Korea
{eunjong.oh, seonguki.lee, yoseop.lee, hyojin9.choi} @samsung.com

[*]Siemens EDA, {Korea, USA}
{euisang.yoon, arun.gogineni, saurabh.s, sungyun.yoo, geonbeom.kwon} @siemens.com

***Abstract*- As semiconductor process technologies continue to scale, the frequency of random defects has increased, making post-silicon defect screening increasingly critical. While structural test techniques are widely adopted in logic devices, their use in memory devices is constrained by area overhead. Accordingly, memory post-silicon testing relies heavily on functional patterns, emphasizing the need for effective evaluation of their defect detection capability. Fault injection simulation is a practical approach for evaluating test coverage; however, increasing design complexity and the growing number of faults and test patterns make exhaustive simulation infeasible. This paper presents a suite of optimization techniques for efficient fault injection simulation. In particular, we detail static fault filtering, fault clustering, design pruning, stimulus grading, dynamic fault optimization, and parallel simulation management. These methods were implemented in a commercial fault injection simulator toolset and applied to a production-grade NAND Flash memory design, achieving 3.7x reduction in total simulation time.**

I. Introduction

Continued scaling of semiconductor processes heightens the occurrence of random defects, posing serious yield and reliability concerns. As these defects dominate post-silicon failures, effective screening strategies have become essential. While logic devices widely adopt structural test approaches such as scan-based Design for Testability (DFT) [1], these methods impose significant area and routing overheads that make them less suitable for memory devices. Replacing standard D flop-flops with scan flip-flops increases the active area, while scan chain stitching introduces additional routing overhead that is difficult to accommodate without enlarging layout area. Adding metal layers to ease this routing congestion would further increase manufacturing cost, which is impractical for cost-sensitive memory products. Moreover, converting inherently asynchronous control and timing structures into fully scan-compatible forms often leads to power and timing penalties, further discouraging the application of scan DFT to memory design.

Given these limitations of scan-based testing, post-silicon testing of memory devices relies primarily on functional test patterns. These patterns are designed to validate the functional behavior of memory devices under normal and stressed operating conditions, and it is essential that they provide sufficient coverage to ensure comprehensive verification of device functionality. To ensure completeness and coverage closure, the defect detection capability of these functional tests must be quantitatively evaluated. Fault simulation provides a practical means to perform such evaluations by injecting modeled faults and analyzing the resulting output discrepancies. However, the computational complexity of fault simulation grows roughly with the product of the number of faults and test patterns, making exhaustive simulation infeasible for modern large-scale designs.

Fault simulation has benefited from advances in computing hardware and parallel execution frameworks, yet its applicability to large and complex designs remains limited due to the excessive number of simulations runs required. Although modern fault simulation frameworks are generally applicable to a wide range of designs, they have primarily evolved in the context of scan-based DFT, where they are used to evaluate the effectiveness of Automatic Test Pattern Generation (ATPG)-generated test vectors in logic circuits. When applied to designs without scan structures-such as memory peripheral circuits-the required number of simulations becomes prohibitively large, making functional fault grading computationally infeasible.

This paper presents a suite of optimization techniques aimed at accelerating fault injection and grading for memory designs. Building on established concepts such as fault collapsing, dynamic fault elimination, and parallelization introduced in earlier studies [2], [3], our approach integrates additional strategies—including design pruning, stimulus grading—to address the scalability challenges of functional fault grading. Some of these techniques can be applied independently, while others-such as design pruning-are designed to amplify the benefits

of clustering-based fault reduction. The proposed optimizations have been implemented in a commercial fault injection simulator and validated on a production-grade NAND Flash gate-level netlist. Experimental results demonstrate up to a 3.7x reduction in total simulation time. Although evaluated on a memory design, the proposed methods are generally applicable to fault simulation frameworks in scan-based ATPG and other logic test environments.

II. BACKGROUND

As semiconductor technologies scale, random defects increasingly threaten yield and reliability, making post-silicon screening essential. To mitigate these risks, DFT methodologies improve controllability and observability of internal nodes for structured testing, enabling early defect detection and guiding coverage analysis.

A. General DFT Methodology and Flow

A typical DFT-based test flow aims to maximize defect detection through structured testing. It consists of five key steps:

- **Scan Insertion:** Adds scan chains to improve controllability and observability of internal states.
- **ATPG Pattern Generation:** Creates structural patterns targeting modeled faults such as stuck-at and transition faults.
- **Fault Simulation using ATPG patterns:** Applies patterns to the gate-level netlist and analyzes fault propagation.
- **Fault Grading:** Evaluates the effectiveness of ATPG-generated patterns by assessing fault detection and overall coverage.
- **Post-Silicon Test Execution:** Deploys validated patterns during manufacturing test for defect screening.

This structured approach is highly effective for logic designs but introduces significant area overhead, making full adoption impractical for memory designs.

B. DFT Fault Grading

DFT Fault Grading evaluates the effectiveness of structural test patterns by measuring their ability to detect injected faults. It provides a detailed view of fault detectability beyond ATPG coverage metrics, ensuring high-quality test sets within the DFT flow.

The grading process typically involves:

- **Fault List Generation**: Create a list of faults to be modeled based on the gate-level netlist.
  Fault types include: stuck-at, time-delay, path-delay, and user-defined.
- **Fault Injection Simulation with Test Patterns:** Apply structural test stimuli and inject faults one at a time to observe fault activation, propagation, and detection behavior.
- **Detection Analysis**: Verify whether fault effects reach observable outputs.
- **Coverage Calculation**: Compute the percentage of detected faults to guide pattern refinement.

C. Functional Fault Grading

Conventional DFT flows focus on grading structural patterns generated by ATPG tools. However, these patterns often fail to detect faults in designs with limited scan access or complex functional behavior.

Functional Fault Grading addresses this limitation by evaluating the defect detection capability of functional patterns derived from real operational scenarios [4]. Unlike ATPG patterns, functional patterns reflect actual system behavior, making them essential for architectures where structural testing is constrained. As illustrated in Figure 1, functional fault grading enables coverage improvement by targeting untestable faults that ATPG patterns cannot detect, using functional patterns derived from real operational conditions.

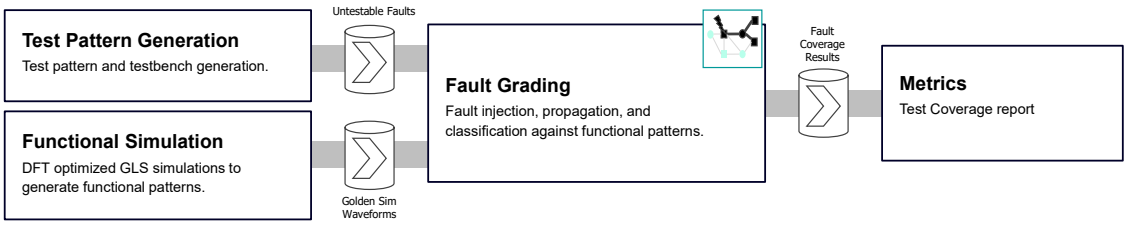

Figure 1. Functional Fault Grading within the DFT Fault Grading Framework

D. Motivation and Challenges for Functional Fault Grading in Memory Designs

Memory designs are subject to strict area constraints, making production cost efficiency a critical priority. Unlike logic designs, where scan-based DFT can be adopted with acceptable overhead, memory architectures cannot afford additional circuitry for controllability and observability.

Accordingly, functional test patterns—reflecting real operational behavior—offer a practical approach for post-silicon defect screening without incurring area penalties. Functional Fault Grading enables designers to evaluate the coverage of these patterns through fault injection and simulation under realistic stimuli.

However, functional patterns are inherently long and complex, unlike short ATPG-generated sequences. This significantly increases turnaround time (TAT) for fault injection simulation, creating a major bottleneck in coverage closure. Addressing this challenge requires an optimization-driven grading flow that reduces computational overhead and scales efficiently for large fault sets and extended stimuli sequences.

III. Proposed Methods

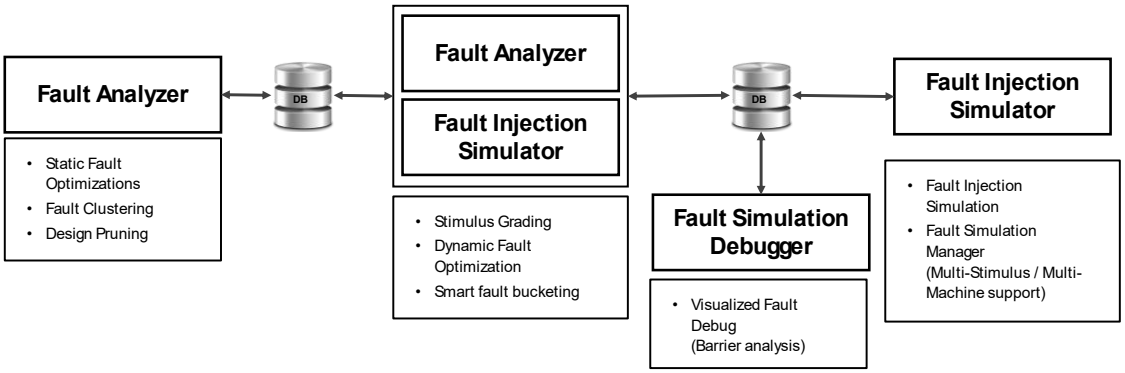

Figure 2. Fault Injection Simulation Flow with Integrated Optimization Techniques

To address the TAT challenges associated with fault injection simulation in memory designs, we propose an optimization-driven functional fault grading flow. This flow integrates multiple techniques across both analysis and simulation stages to reduce computational overhead and improve scalability. Figure 2 illustrates the overall structure of the proposed fault injection simulation flow, which integrates multiple optimization techniques to enhance the efficiency of functional fault grading. The flow consists of the following key components:

A. Static Fault Optimization

The static fault optimization stage aims to reduce the fault set size before simulation through static design analysis. First, untestable faults—those that cannot be activated or propagated under any functional condition—are identified and removed. As depicted in Figure 3, unused or blocked faults are identified as untestable during this stage. Second, fault collapsing is applied to group logically equivalent faults and defines a set of primary faults representing each equivalence class. Next, optimization applies both at device and functional levels, typically targeting stuck-at fault types as illustrated in Figure 4. This process produces a refined fault list for injection and evaluation, forming the basis for subsequent optimization steps.

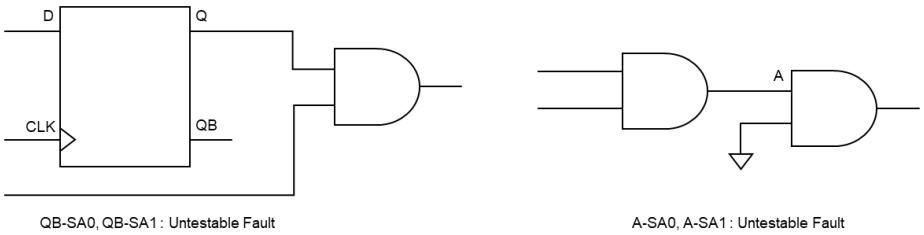

Figure 3. Untestable Fault Filtering

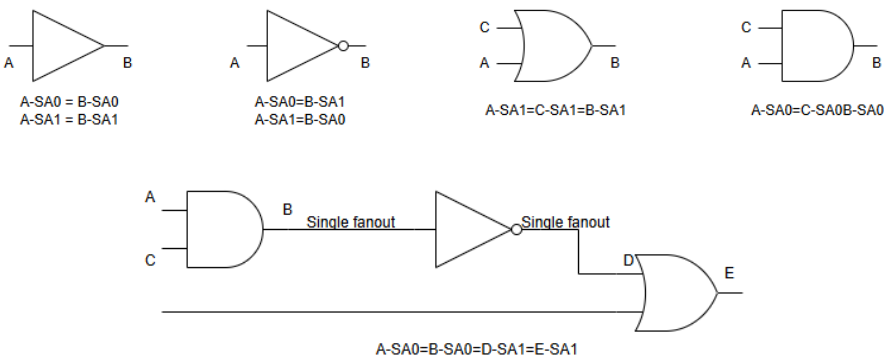

Figure 4. Fault Collapsing at Functional and Device Level

B. Fault Clustering

Fault clustering improves design optimization and simulation efficiency by grouping faults into structured sets based on common characteristics. Instead of handling faults individually, clustering organizes them according to structural attributes such as fanout and influencing logic, ensuring that faults with similar propagation behavior are grouped together. This approach enables localized optimizations and allows the simulation process to handle each cluster independently, reducing redundant computations.

Clusters also serve as the foundation for parallel fault simulation management, enabling efficient distribution of fault sets across available machine resources for balanced and scalable execution. In this work, instance-based clustering was adopted, where all faults within the same instance are grouped together. Faults that do not fall within any cluster are simulated separately. Figures 5 and 6 illustrate the clustering process: Figure 5 shows random fault injection, while Figure 6 depicts cluster boundaries created around faults.

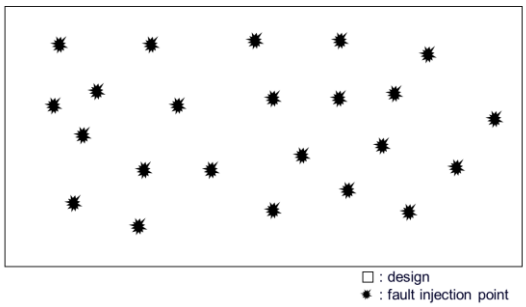

Figure 5. Traditional Fault Processing

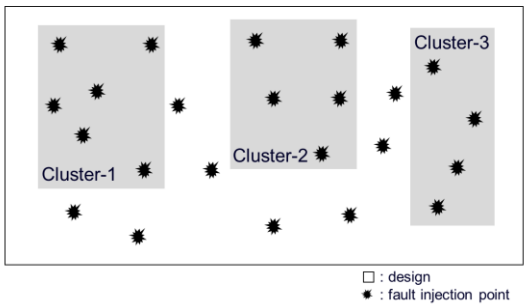

Figure 6. Fault Clustering Based on the Fault Propagation Path

C. Design Pruning

The design pruning stage focuses on reducing simulation complexity by creating optimized subcircuits for both golden and faulty simulations. Instead of simulating the entire gate-level design, the process trims the design to the logic cone that begins at the primary inputs and extends to the observation points relevant to the injected fault. This selective reduction preserves all necessary fault propagation paths while eliminating non-essential logic that does not influence fault detection, ensuring that computational resources are concentrated on critical regions of the design for faster fault injection simulations.

When combined with fault clustering, design pruning delivers optimal performance. Clustering groups faults with similar propagation paths, while pruning keeps only the relevant portion of the design active, shutting off non-interesting regions. Once clusters are formed, pruning and optimization data are applied at the cluster level, reducing redundant computations and improving runtime efficiency.

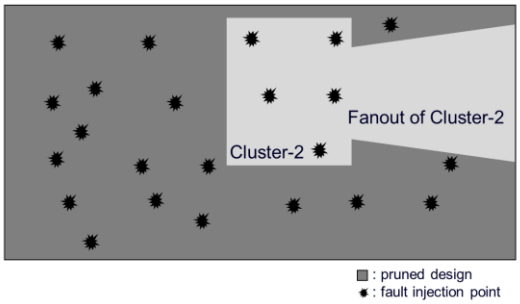

Figure 7. Design pruning for cluster 2 based on the fault propagation path

D. Stimulus Grading for Multiple Stimuli

Stimulus grading prioritizes test vectors based on their fault detection potential, ensuring that the most effective stimuli are applied first. Injectability is defined as the ability of a stimulus to activate and propagate a fault during simulation. Specifically, during the golden simulation phase, if no logic transition occurs at the fault injection point, the fault is classified as not injectable—which typically corresponds to either a stuck-at-0 or stuck-at-1 condition depending on the signal state. Conversely, if a transition occurs, the fault is considered injectable. The grading process begins by ranking stimuli according to the number of injectable faults they can activate. After each simulation run, injectability metrics are recalculated to reflect updated coverage, and stimuli are dynamically re-ranked based on their incremental contribution. This approach minimizes redundant simulations and accelerates fault grading convergence.

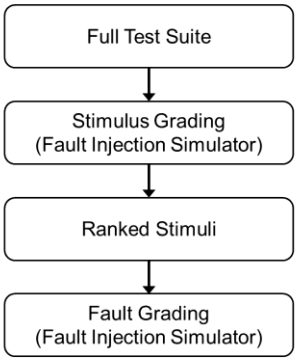

Figure 8. Stimulus Grading Flow in Fault Grading

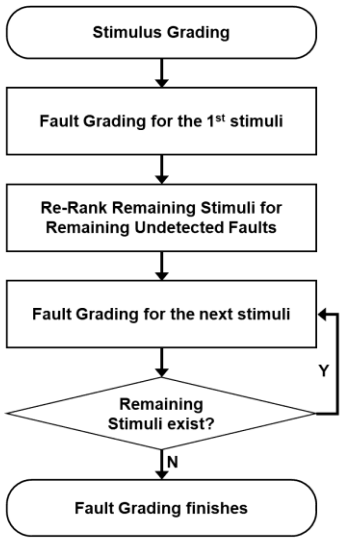

Figure 9. Dynamic Re-ranking flow in Stimulus Grading

E. Dynamic Fault Optimization

Dynamic fault optimization builds on the injectability analysis performed in the stimulus grading stage to refine the fault list for each stimulus. Using injectability results, faults that cannot be activated under the given stimulus are excluded from simulation. For the remaining injectable faults, propagation analysis determines whether the fault effect reaches the first observable receiver net; Faults that fail this check are classified as blocked and removed. This stimulus-specific filtering significantly reduces the simulation workload, ensuring that only faults with actual activation and propagation potential are considered. Combined with stimulus grading, this approach accelerates coverage closure and improves scalability for large fault sets.

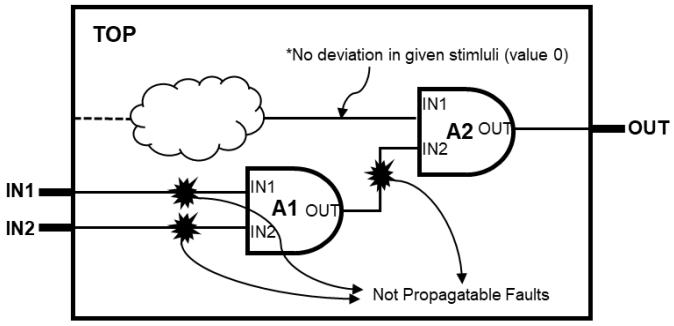

Figure 10. Dynamic Fault Filtering Based on Simulus Behavior

F. Fault Simulation Management for Parallelization

To maximize simulation throughput and resource utilization, the fault simulation manager efficiently distributes fault injection simulation runs for multiple stimuli across distributed machines. This parallelization strategy enables simultaneous evaluation of different fault sets, ensuring balanced workloads and optimal use of available computing resources. As a result, the flow becomes highly scalable for large designs and large fault volumes.

G. Visualized Fault Debug

Fault grading often involves analyzing thousands of faults, making manual root-cause analysis highly time-consuming. To address this challenge, the visualized fault debug methodology performs blocking net analysis following fault injection simulation, to identify fault propagation barriers and signal bottlenecks.

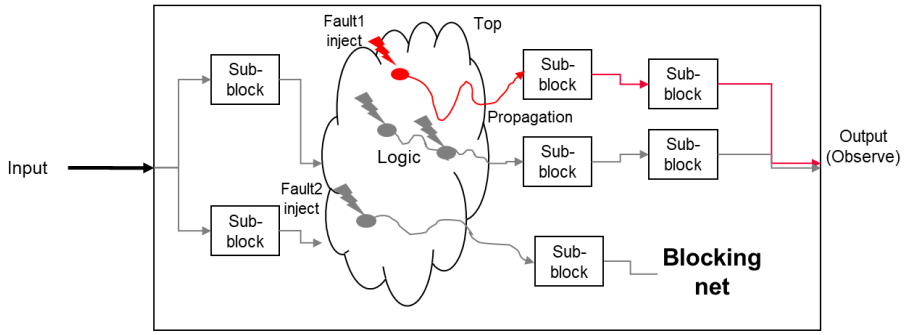

Figure 11. Aspect of Fault Propagation and Blocking

To enhance debugging efficiency, an interactive visualized debugger was implemented with the following capabilities:

- **Blocking Nets**: Summarizes all blocking nets in the fault propagation path for the selected fault, helping engineers quickly identify propagation bottlenecks.
- **Deviating Ports**: Lists all deviating ports for each instance, highlighting deviations caused by the injected fault for faster diagnosis.
- **Initiate Debug**: Allows engineers to trigger debug on the selected fault, automatically updating schematic, source, and DUT views to the fault location.
- **Initiate Debug with Logic Cone**: Provides a structural view of the affected logic cone, enabling context-aware analysis of fault impact.

These features collectively create an interactive, context-aware debugging environment that reduces manual effort and significantly improves fault traceability, thereby shortening debug turnaround time.

IV. Case Study and Results

To validate the proposed optimization-driven fault grading flow in a real-world scenario, we conducted experiments on a production-grade NAND Flash memory design. As illustrated in Figure.12, The design comprises

three major components: (i) a large memory cell array for physical data storage, (ii) a core circuit for executing read/write operations, and (iii) peripheral blocks that control and coordinate core behavior.

An instance within the peripheral logic was selected for fault injection in consideration of its critical role in memory control and accessibility for simulation-based analysis. For fault observability, the output port of this instance served as the observation point, ensuring that faults were considered detectable only if they propagated to the output interface—consistent with practical post-silicon test conditions. To represent realistic operational scenarios, 7 functional testbenches were applied, each covering different functional conditions of the target logic.

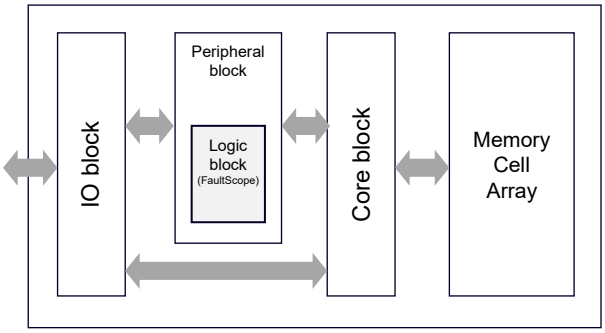

Figure 12. Simplified Block Diagram of the NAND Flash memory design

A. Static and Dynamic Fault Optimization

The first step in functional fault grading is to establish a comprehensive fault list representing potential failure points in the design. From the target design, we extracted 5,302 port-level faults, including both stuck-at-0 and stuck-at-1 types.

		Number of Faults						
Total Faults		5,302						
Statically Optimized Faults	Collapsed Faults	2,278						
	Untestable Faults	36						
Primary Faults (statically active)		**2988 (56.4%)**						
Dynamically Optimized Faults		TB1	TB2	TB3	TB4	TB5	TB6	TB7
	Not Injectable Faults	494	728	667	649	509	503	507
	Blocked Faults	246	278	272	287	260	244	262
Primary faults (statically and dynamically active)		**2,248 (42.4%)**	**1,982 (37.4%)**	**2,049 (38.6%)**	**2,052 (38.7%)**	**2,219 (41.9%)**	**2,241 (42.3%)**	**2,219 (41.9%)**

TABLE 1. Result of Static and Dynamic Fault Optimization

Static optimization, as described in Section III.A, was applied to reduce this initial set. Fault collapsing grouped 2,278 logically equivalent faults, and static filtering identified 36 untestable faults that cannot be activated or propagated under any condition. As a result, the fault list was reduced to 2,988 candidates for injection, representing 56.4% of the original 5,302 faults.

Dynamic optimization, as detailed in Section III.E, further refined the fault list through stimulus-specific analysis. For each testbench, faults were classified as non-injectable if no transition occurred at the injection point during golden simulation, or as blocked if fault propagation failed to reach the first observable receiver net. On average, 264 faults per testbench were excluded through this process.

Overall, the combined application of static and dynamic optimization reduced the fault population to 40.4% of its original size on average, significantly lowering the simulation workload and enabling more efficient fault grading for large-scale memory designs.

B. Stimulus Grading

To further optimize fault injection efficiency, stimulus grading was applied to rank testbenches based on their ability to activate faults during simulation. For each testbench, faults that could be activated and propagated, referred to as injectable faults, were analyzed and ranked in descending order of coverage potential following the approach in Section III.D. Table 2 summarizes the results, with TB1 ranked highest and can activate 2,494 faults. This ranking was then used to prioritize fault injection simulations, ensuring that testbenches with higher coverage potential were executed first. By detecting a large number of faults early, this approach minimizes redundant simulations and improves overall efficiency in coverage closure.

Rank	Stimulus Name	Injectable Faults Count
1	TB1	2,494
2	TB6	2,485
3	TB7	2,481
4	TB4	2,479
5	TB5	2,339
6	TB3	2,331
7	TB2	2,260

TABLE 2. Result of Stimulus Grading

C. Fault Clustering and Design pruning

Fault clustering and design pruning were applied to further accelerate fault injection simulation. Fault clustering, described in Section III.B, groups faults with similar propagation paths, reducing redundant simulations. Design pruning, introduced in Section III.C, restricts the design to fault-relevant logic cones and removes inactive regions.

To assess the impact of these techniques, memory consumption was measured for sample fault runs. As shown in Table 3, approximately 70% memory savings were achieved compared to the baseline approach without fault clustering and design pruning. This structural-level optimization not only improves runtime but also enhances resource efficiency, which is critical for scaling fault grading to large designs.

The runtime decreased from 66,667 seconds to 31,611 seconds after design pruning was applied, under identical computing resources with 160 parallel threads and the same experimental setup comprising 2,988 faults across seven testbenches. Both configurations achieved an identical fault detection rate of 67.2%. Overall, the combined application of fault clustering and design pruning reduced the total runtime by 61.0% compared to the baseline approach. This result underscored the critical role of structural-level optimization in enhancing scalability and resource efficiency for large fault sets.

Testbench	# of faults of Sample Fault Run	Active memory consumption during simulation (GB)	
		Full design	Pruned design
TB1	30	7.31	2.11
TB2	124	6.51	1.83
TB3	26	6.50	1.83
TB4	29	6.50	1.86
TB5	34	6.50	1.88
TB6	25	6.51	2.07
TB7	31	6.51	2.30

TABLE 3. Memory Consumption Comparison for Sample Fault Runs

D. Parallel Fault Simulation

Parallelization, as discussed in Section III.F, was applied to distribute fault injection simulations across multiple computing resources, significantly reducing overall runtime. Figure 13 shows the runtime scaling with increasing parallel thread count under identical conditions using 2,988 faults and a single testbench. Runtime decreases as thread count grows, but the improvement trend saturates near 160 threads, with a slight increase beyond that point. Since total runtime cannot drop below the execution time of a single fault run, a saturation point is inevitable. Moreover, as thread count rises, additional processing overhead for managing threads can increase runtime slightly. Therefore, identifying an optimal thread count is critical for maximizing efficiency while avoiding unnecessary resource allocation. In this experiment, the optimal point is observed near 160 threads.

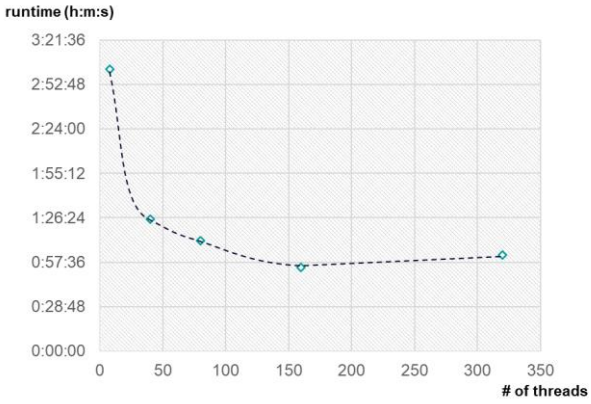

Figure 13. Runtime vs. Parallel Thread Count

E. Overall Results and Impact

The fault injection simulation conducted on the selected instance using 7 functional testbenches achieved a fault detection rate of 72.2%. To assess efficiency improvements, total runtime was compared between two configurations using 160 parallel threads: The unoptimized simulation required 114,214 seconds, whereas the full set of optimizations reduced the runtime to 30,932 seconds, corresponding to a 3.7x improvement. These results demonstrate the effectiveness of the proposed methodology in accelerating functional fault grading. The methodology integrates static and dynamic fault optimization, stimulus grading, and fault clustering with design pruning. It significantly improves scalability and confirms practical applicability to large-scale memory designs.

V. CONCLUSION

The optimization techniques presented in this paper have been successfully implemented within Siemens EDA's commercial fault injection simulator toolset. Through this integration, we have confirmed the practical applicability of the proposed methodology to real-world designs.

In particular, the experiments demonstrated that the proposed flow can be effectively applied to memory-oriented designs, including NAND Flash memory designs. This validation highlights the feasibility of functional fault grading in environments where conventional DFT methodologies face limitations due to area constraints.

Moreover, the underlying optimization strategies are not limited to memory designs. They are broadly applicable to logic designs and System-On-Chip (SoC) architectures, where scalable fault grading is essential for achieving comprehensive defect coverage.

Beyond general applicability, the proposed methodology also shows promise in the context of Functional Safety. Fault injection simulation plays a critical role in safety-critical systems, and the ability to efficiently evaluate fault coverage using functional patterns can significantly enhance safety validation workflows.

While the proposed techniques have demonstrated significant improvements in simulation efficiency and scalability, there remains a continued need for further enhancement of simulation runtime. Functional test patterns, by nature, are longer and more complex than structural patterns, and this inherent characteristic introduces persistent challenges in fault injection simulation. Despite the optimizations applied, the overall simulation cost can still be substantial, especially for large-scale designs or extensive test suites. We are exploring methods to automatically generate efficient functional test patterns that maximize defect detection while minimizing simulation overhead as a future work. This direction aims to further improve the practicality and responsiveness of functional fault grading in both general-purpose and safety-critical applications.

REFERENCES

[1] L.-T. Wang, C.-W. Wu, and X. Wen, VLSI Test Principles and Architectures: Design for Testability, Elsevier, 2006.
[2] A. Verreault, E.M. Aboulhamid, Y. Karkouri, “Multiple Fault Analysis Using a Fault Dropping Technique,” Proc. Int. Symp. on Fault-Tolerant Computing, pp. 162–169, 1991.
[3] D.G. Saab. I.N. Hajj and J.T. Rahmeh. "Parallel-Concurrent Fault Simulation", Int'l Conference on Computer Design, pp. 298-301, 1989.
[4] J. Wiltgen and L. Harrison, “Accelerating DFT sign-off with Questa One”, Siemens Verification Academy, 2025.
[5] Siemens EDA, “Questa One Safety Analyzer User Guide Software Version 2025.2”, 2025.
[6] Siemens EDA, “Questa One Fault Simulator User Guide Software Version 2025.2”, 2025.