IMPROVING RUNTIME EFFICIENCY IN EMULATION MODELS OF SERVER IPs

Endterm report submitted in partial fulfillment of the requirement for the Award of the Degree of

MASTER OF TECHNOLOGY

in VLSI Design

Submitted By

Saloni Garg

6024620018

Under Supervision of

Dr. Sanjay Kumar and Dr. Hari Shankar Singh

Associate Professors

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
(A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB
JUNE, 2026

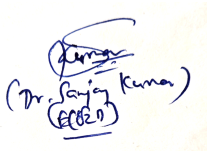
DECLARATION

I, **Saloni Garg** hereby declare that the work presented in this thesis entitled **"Improving Runtime Efficiency in Emulation Models of Server IPs"** in partial fulfillment of the requirement for the award of degree of **Master of Technology (VLSI Design**) submitted at **ECED**, Thapar Institute of Engineering & Technology (Deemed to be University), Patiala is an authentic record of work carried out under supervision of **Abinav Tiwari (Manager, IPSCG EMU Capabilities, Intel Corporation) , Dr. Sanjay Kumar (Associate Professor, ECED, TIET)** and **Dr. Hari Shankar Singh (Associate Professor, ECED, TIET)** from **16 June, 2025** to **12 June, 2026**. The matter presented in this has not been submitted either in part or full to any other university or institute for the award of any other degree.

Sign

Date: 12 June, 2026

(Saloni Garg)
(6024620018)

(Industry Supervisor)  Abinav Tiwari Manager IPSCG EMU Capabilities, Intel Technology India Pvt. Ltd., Bengaluru Date: 12 June, 2026	(TIET Supervisor-I)  Dr. Sanjay Kumar Associate Professor Department of Electronics and Communication Engineering, Thapar Institute of Engineering & Technology, Patiala Date: 12 June, 2026
	(TIET Supervisor-II)  Dr. Hari Shankar Singh Associate Professor Department of Electronics and Communication Engineering, Thapar Institute of Engineering & Technology, Patiala Date: 12 June, 2026

CERTIFICATE

Tel: +91-80-6211 3000
Fax: +91-80-6280 9010

To Whomsoever It May Concern

Employee Name: Saloni Garg

Internship Dates: 06/16/2025 to 06/12/2026

The letter is to confirm the mentioned above has undergone internship at Intel Technology India Private Limited. We wish you all the best for your future assignments.

Yours Sincerely,
For Intel Technology India Pvt. Ltd.

Harishkumar S
Authorized Signatory

Registered Office:
Intel Technology India Pvt Ltd.
#23-56P, Outer Ring Road, Devarabeesanahalli, Varthur Hobli, Bellandur Post
Bengaluru 560 103, Karnataka, India
CIN – U85110KA1997PTC021606

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to Intel Corporation for providing me with the opportunity to work as an intern with the IPSCG EMU Capabilities team. This experience has been invaluable in understanding the complexities of modern hardware validation and emulation technologies.

I extend my heartfelt thanks to my mentor, Mr. Abinav Tiwari, for his continuous guidance, support, and mentorship throughout this project. His expertise in emulation technologies helped me navigate the technical challenges and achieve meaningful results. I am grateful to the entire IPSCG EMU Capabilities team at Intel for their collaboration and knowledge sharing. My heartfelt thanks to Mr. Varnit Goswami, Emulation Engineer, and Mr. K N Raghavendra, Senior Emulation Engineer, for their valuable inputs in completing this project.

I also acknowledge the support provided by Dr. Sanjay Kumar and Dr. Hari Shankar Singh, my college supervisors, for their guidance throughout this project. I thank my academic institution and faculty members, including Dr. Kulbir Singh, Head of Department, Electronics and Communication Engineering, who encouraged this industrial collaboration and provided the necessary framework for this work.

Finally, I express my appreciation to my family for their constant support and encouragement.

Saloni Garg
Intel IPSCG EMU Capabilities Team Intern
Date: 12 June, 2026

ABSTRACT

This project explores improving runtime efficiency in emulation models of server Intellectual Properties (IPs) for modern Very Large Scale Integration (VLSI) design. Emulation involves mimicking functionality and behavior of real-time hardware systems, operating significantly faster than simulation at clock frequencies of several GHz. This enables execution of extensive test cases in reduced time, uncovering deep corner-case bugs that simulation typically fails to detect. Since emulation utilizes the same Register Transfer Level (RTL) code deployed on actual silicon, comprehensive bug elimination becomes critical for server IP quality.

The primary objective is improving runtime efficiency of emulation models by optimizing total time spent by software and hardware components during emulation runs to achieve better resource utilization. The secondary objective focuses on developing Python-based automation scripts to reduce manual effort during validation and improve overall workflow efficiency. The work addresses challenges posed by high costs and limited availability of emulation platforms through systematic optimization of testbench and software components in collaboration with Ultra eXtensible Interconnect Verification Intellectual Property teams. Through systematic optimization phases spanning multiple development cycles, progressive improvements were achieved using data-driven analysis and targeted enhancement strategies.

A comprehensive five-component automation validation framework was developed encompassing hardware configuration parsing, memory region analysis, test configuration generation, workflow coordination, and parallel execution management. This approach resulted in a remarkable 41% reduction in runtime, substantially exceeding the initial target of 30% improvement, while establishing scalable automation methodologies that transform validation preparation from manual processes into systematic, repeatable workflows for server IP validation.

TABLE OF CONTENTS

DECLARATION…….. ii
CERTIFICATE……iii
ACKNOWLEDGEMENT…….. iv
ABSTRACT…….. v
TABLE OF CONTENTS…… vi
LISTS OF TABLES ……...vii
LISTS OF FIGURES…….... viii
LIST OF ABBREVIATION……... ix
CHAPTER 1 ……...……11
INTRODUCTION…….. 11
1.1 PROBLEM STATEMENT……... 12
1.2 OBJECTIVES……...13
CHAPTER 2…….. 14
LITERATURE REVIEW…….....14
2.1 PROPOSED MODEL ARCHITECTURE……... 14
2.2 ZEBU EMULATORS……...16
CHAPTER 3……19
EMULATION MODEL VALIDATION……... 19
3.1 COHERENCY VALIDATION TESTBENCH……19
3.2 TRANSACTION LIFE-CYCLE…….. 21
3.3 RUNTIME IMPROVEMENT OF THE EMULATION MODEL……... 22
CHAPTER 4……25
AUTOMATION FOR VALIDATION…….. 25
4.1 AUTOMATION FRAMEWORK OVERVIEW…….. 26
4.2 HARDWARE CONFIGURATION PARSER AND INJECTION GENERATOR….. 27
4.3 MEMORY REGION ANALYSIS AND ADDRESS-SPACE MAPPING…….. 29
4.4 TEST CONFIGURATION GENERATION AND RESOURCE OPTIMIZATION… 30
4.5 WORKFLOW MANAGEMENT AND PROCESS COORDINATION……. 32
4.6 PARALLEL EXECUTION AND SCALABILITY MANAGEMENT……... 33
4.7 INTEGRATION IMPACT AND SYSTEM PERFORMANCE…….. 34
CHAPTER 5……36
RESULTS AND ANALYSIS……..36
5.1 RUNTIME OPTIMIZATION PERFORMANCE RESULTS…….. 36
5.2 AUTOMATION FRAMEWORK VALIDATION RESULTS……..43
5.3 SYSTEM PERFORMANCE AND VALIDATION RESULTS……... 45
CHAPTER 6……47
CONCLUSION……... 47
REFERENCES……... 50

LISTS OF TABLES

Sr. No	Table Details	Page No
Table 5.1	*Initial Performance Profiling Results for Test Case 1……*	*37*
Table 5.2	*Initial Performance Profiling Results for Test Case 2……*	*37*
Table 5.3	*Runtime Comparison ……*	*38*
Table 5.4	*Runtime Comparison with 6% Improvement……*	*38*
Table 5.5	*Runtime Comparison with 14% Improvement ……*	*39*
Table 5.6	*BFM Version Comparison ……*	*40*
Table 5.7	*Final Optimization Summary……*	*42*

LISTS OF FIGURES

Sr. No	Figure Details	Page No
Figure 1.1	*Simplified Hardware Verification and Validation Flow……*	*11*
Figure 2.1	*IP-Level Emulation Model Architecture ……*	*15*
Figure 2.2	*Synopsys ZeBu Server 5 Emulation Platform ……*	*17*
Figure 3.1	*UVM-SC Testbench Runtime Flow for Cache Coherency Validation ……*	*20*
Figure 3.2	*zTune Profiler ……*	*22*
Figure 3.3	*zTune Cell Tree View ……*	*23*
Figure 4.1	*Overall structure of the validation automation framework ……*	*26*
Figure 4.2	*Hardware Configuration Parser and Injection Script Generation Flow……*	*28*
Figure 4.3	*Memory Region Analysis and Address Space Mapping Flow ……*	*30*
Figure 4.4	*Test Configuration Generation and Address Region Selection Flow..*	*31*
Figure 4.5	*Validation Pipeline Workflow and Stage Coordination Flow ……*	*33*
Figure 4.6	*Parallel Validation Setup Execution and Scalability Management Flow……*	*34*
Figure 5.1	*Overall Time Data of Test 1 ……*	*36*
Figure 5.2	*Overall Time Data of Test 2 ……*	*37*
Figure 5.3	*Initial zTune Data for the testrun ……*	*38*
Figure 5.4	*Improved zTune Data for the testrun ……*	*39*
Figure 5.5	*Runtime Data for the Model ……*	*40*
Figure 5.6	*Final Optimization Summary……*	*43*
Figure 5.7	*Generated Hardware Injection Script……*	*44*
Figure 5.8	*Generated Address Space Map……*	*44*
Figure 5.9	*Generated Validation Configuration……*	*45*

LIST OF ABBREVIATION

ASF	Address Snoop Filter
BFM	Bus Functional Model
CA	Caching Agent
CL	Cache Line
CMI	Converged Memory Interface
DDR	Double Data Rate
DMR	Diamond Rapids
DUT	Device Under Test
EDA	Electronic Design Automation
FPGA	Field Programmable Gate Array
HA	Home Agent
HAMVF	Home Agent Memory Virtualization Framework
HSF	Home Snoop Filter
IP	Intellectual Property
MC	Memory Controller
MESI	Modified, Exclusive, Shared, Invalid
NIP	Network Interface Protocol

RTL	Register Transfer Level
RTID	Requestor Transaction ID
SoC	System on Chip
T2M	Time-to-Market
TAT	Turn Around Time
UVM-SC	Universal Verification Methodology SystemC
UXI	Ultra eXtensible Interconnect
VIP	Verification Intellectual Property
VLSI	Very Large Scale Integration

CHAPTER 1

INTRODUCTION

The semiconductor industry has experienced significant growth in the complexity of modern processor and System on Chip (SoC) designs. The increasing demand for cloud computing, artificial intelligence, high-performance computing, and data center applications has driven the development of processors containing billions of transistors integrated into a single device. As the scale and complexity of these designs continue to increase, ensuring functional correctness throughout the design cycle has become a critical challenge. Any defect that escapes into silicon can lead to substantial financial losses, delayed product launches, increased validation effort, and reduced product reliability. As a result, verification and validation activities constitute a major portion of the overall hardware development cycle.

Verification aims to ensure that the implemented design behaves according to its intended specifications under a wide range of operating conditions. Traditionally, simulation-based verification has been extensively used during the early stages of design validation. However, as modern server IPs become increasingly complex, simulation alone is often insufficient to achieve the desired verification coverage within practical timelines. The execution speed of simulators can become a limiting factor when validating large software workloads and complex transaction scenarios. A simplified hardware verification and validation flow is shown in Figure 1.1.

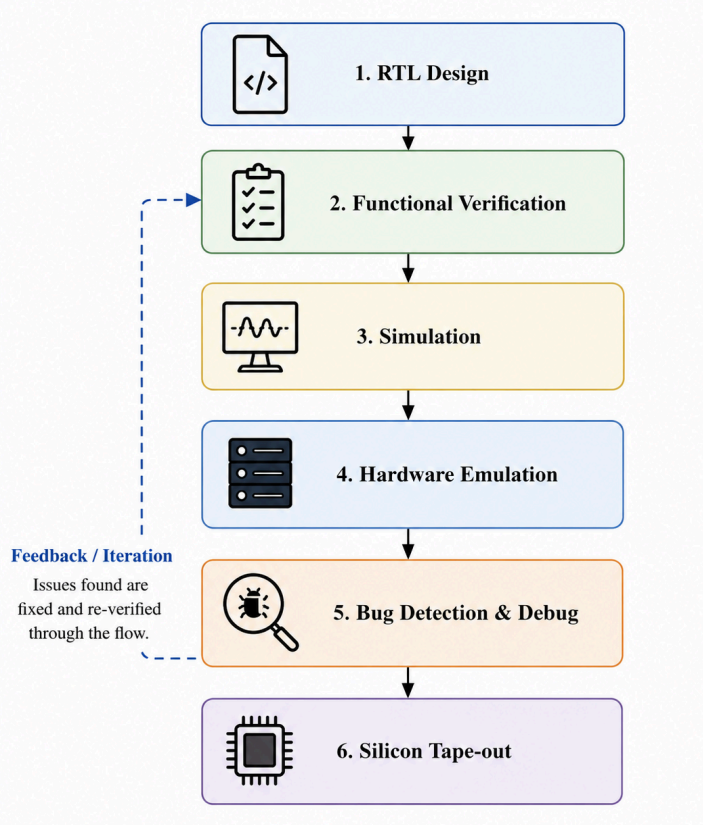


Figure 1.1: Simplified Hardware Verification and Validation Flow

To overcome these limitations, hardware emulation has become an important component of modern semiconductor verification flows. Emulation platforms provide significantly higher execution speeds than traditional simulation while maintaining sufficient visibility into internal design behavior. This enables validation teams to execute software-driven workloads, perform hardware-software co-validation, and identify functional issues much earlier in the development cycle.

Server Intellectual Properties (IPs) represent some of the most complex building blocks within modern data center processors. These IPs are responsible for managing critical operations such as cache coherency, memory transactions, data movement, and communication between multiple processing agents. Since several agents may access shared resources simultaneously, maintaining data consistency across the system becomes essential. Consequently, extensive validation is required to ensure correct operation under different traffic conditions and workload scenarios. Hardware emulation plays an important role in supporting such validation activities by enabling realistic workloads to be executed within practical timeframes while maintaining the required level of design visibility.

The continuous evolution of RTL designs and the adoption of weekly development cycles have further increased the importance of efficient validation methodologies. Verification teams are required to repeatedly build, deploy, and execute emulation models to validate newly introduced features and bug fixes. While improvements in model build time contribute significantly to engineering productivity, runtime efficiency remains equally important because emulator resources are expensive and limited in availability. Efficient utilization of emulator resources directly impacts validation throughput, regression execution, and overall project schedules. Therefore, reducing runtime overhead has become an important focus area for improving the effectiveness of modern emulation environments.

Another important aspect of emulation-based validation is the interaction between hardware and software components. Modern validation environments consist of RTL designs, verification IPs, software workloads, protocol checkers, scoreboards, monitors, and automation infrastructure. The performance of the complete validation environment therefore depends not only on the hardware design but also on the efficiency of the associated software and testbench components. Any inefficiency within these supporting elements can increase runtime and reduce overall emulator utilization. As design complexity continues to grow, improving the efficiency of emulation-based validation environments has become increasingly important for maintaining productivity, reducing development effort, and supporting timely product delivery.

1.1 PROBLEM STATEMENT

Despite significant advancements in verification methodologies, bug escape to silicon can still result in billions of dollars in losses and substantial delays in Time-to-Market (T2M) for product launches. Therefore, early detection of functional issues remains a critical objective during the validation phase. Improving the quality of Intellectual Property (IP) directly contributes to the overall quality and reliability of the final

System on Chip (SoC), making efficient validation an important requirement throughout the development cycle.

Our team has made significant efforts to improve model build time consumption in the previous semester. We were able to reduce the Turn Around Time (TAT) for our models by 62% as compared to our initial target. This has positively impacted our workflow as we have model builds on a weekly cadence. However, we now face challenges related to runtime efficiency due to the high cost and limited availability of emulators. With the increased operating costs associated with using these emulators and a growing demand for resources and a need to maintain feature quality, we cannot afford to reduce Register Transfer Level (RTL) complexity.

To address these challenges, we need to focus on optimizing the testbench and software components of our workflow. We aim to identify strategies that will reduce the time spent on testbench collaterals by collaborating with Ultra eXtensible Interconnect Verification Intellectual Property (UXI VIP) team, ultimately enhancing our overall emulation efficiency while preserving the integrity of our design features.

1.2 OBJECTIVES

The goal of this project is to improve the runtime efficiency of the emulation model by 30%. Since emulator resources are expensive and limited in availability, improving runtime efficiency directly contributes to better resource utilization and increased validation throughput. The key parameters used to measure this efficiency improvement are:

- Optimizing the total time spent by software and hardware components during emulation runs to achieve better resource utilization.
- Developing python based automation scripts to reduce manual effort during validation and improve overall workflow efficiency.

CHAPTER 2

LITERATURE REVIEW

2.1 PROPOSED MODEL ARCHITECTURE

The emulation model consists of a Device Under Test (DUT) comprising Home Agent Memory Virtualization Framework (HAMVF), Address Snoop Filter (ASF), Home Snoop Filter (HSF), and supporting software components used for validation [2]. Cache coherency validation is one of the most critical activities in modern server processor verification because multiple processing agents may access shared memory resources simultaneously. Traditionally, coherency validation is performed at the System on Chip (SoC) level; however, this approach often results in increased emulator utilization, longer model compilation times, higher storage requirements, and greater engineering effort. These challenges can reduce debugging efficiency and affect overall project schedules. To address such limitations, IP-level validation methodologies based on Universal Verification Methodology SystemC (UVM-SC) have gained importance due to their ability to provide scalable and efficient verification environments [3][4].

Modern server processors contain several interconnected components that communicate continuously through high-speed interconnects and memory subsystems. As processor core counts continue to increase, maintaining cache coherency across multiple agents becomes increasingly complex. Validation environments must therefore be capable of exercising a wide range of coherency scenarios while providing sufficient observability and controllability. IP-level emulation models help achieve these goals by enabling focused validation of coherency behavior while reducing overall verification complexity compared to full SoC validation [2][3].

The Diamond Rapids (DMR) Data Center processor architecture includes key components such as Home Agents (HA), cache controllers, caching agents, and memory subsystems that collectively maintain data consistency throughout the system [2][3]. The Home Agent plays a central role in coherency management by ensuring that the most recent copy of requested data is returned to the requesting agent. Depending on system state, the requested data may be obtained from another caching agent or from main memory. The Home Agent is also responsible for invalidating stale copies and granting ownership permissions when required, thereby maintaining correctness across all participating agents.

In modern coherency architectures, maintaining data consistency becomes increasingly challenging as the number of processing agents grows. Read requests, write requests, ownership transfers, cache evictions, and memory accesses generate a large number of coherency transactions that must be handled correctly. Validation of these transactions is essential because coherency-related issues may only appear under specific traffic conditions or corner-case scenarios. Consequently, verification environments must support comprehensive testing of coherency operations under a wide range of workloads [2][3].

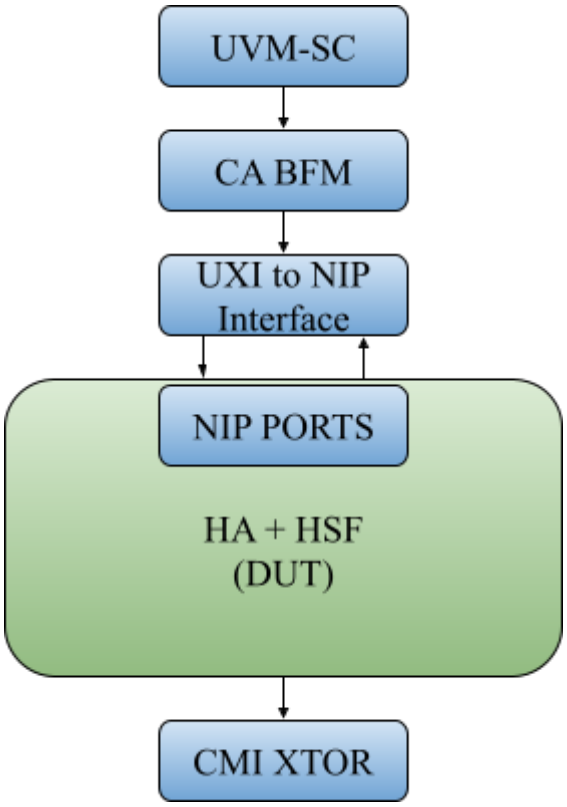


Figure 2.1: IP-Level Emulation Model Architecture [2] (Intel Property)

Intel utilizes the Ultra eXtensible Interconnect (UXI) protocol as a standard interface for both coherent and non-coherent communication between system components [2][4]. UXI provides a common framework for exchanging requests, responses, and data among participating agents. Within the architecture, HAMVF receives read and write transactions from Caching Agents (CAs) through the Network Interface Protocol (NIP). Based on the current coherency state of the requested cache line, appropriate responses are generated and returned to the requesting agent [2][4].

Coherency management within such architectures commonly follows the Modified, Exclusive, Shared, Invalid (MESI) protocol [2][4]. The MESI protocol defines the ownership and sharing status of cache lines and enables multiple caches to maintain a consistent view of system data. A cache line may exist in one of four states: Modified, Exclusive, Shared, or Invalid. Transitions between these states occur in response to memory operations and coherency requests generated throughout the system. Correct implementation and validation of MESI state transitions are important because improper state management can lead to stale data, ownership conflicts, or incorrect memory behavior [2][4].

The Network Interface Protocol (NIP) serves as a transport mechanism for communication between different agents within the coherency architecture [1][2]. NIP consists of two primary interface types: DROP and ADD ports. DROP ports are used to receive incoming transaction packets, while ADD ports are used to transmit outgoing packets back into the fabric. Such an interface structure provides an organized mechanism for handling transaction flow between system components. Incoming requests undergo protocol processing and

coherency evaluation before corresponding responses are generated and transmitted through the appropriate interfaces [1][2].

HAMVF operates in conjunction with the Home Snoop Filter (HSF), which assists in coherency resolution by maintaining information about caching agents that may contain copies of specific cache lines [2][3]. By tracking cache-line ownership and sharing information, HSF helps reduce unnecessary snoop traffic within the system. Instead of broadcasting snoop requests to all agents, the coherency subsystem can identify only the relevant agents that may contain the requested data. This targeted approach improves transaction efficiency and reduces coherency overhead [2][3].

When information about a requested cache line is available within HSF, coherency decisions can be performed more efficiently. If the required cache line is not present, memory access is performed through the Converged Memory Interface (CMI) to retrieve the requested data while maintaining coherency requirements [2][3]. The coordinated operation of HAMVF, HSF, and memory interfaces enables efficient handling of coherency transactions and supports reliable data delivery under varying workload conditions.

2.2 ZEBU EMULATORS

Zebu emulators, developed by Synopsys, are widely used in the semiconductor industry for validating complex System on Chip (SoC) designs due to their scalability, performance, and ability to execute large Register Transfer Level (RTL) designs efficiently [5][6]. As modern processors continue to increase in complexity, traditional simulation-based verification often becomes time-consuming when executing large software workloads and long validation scenarios. Hardware emulation addresses these limitations by providing significantly higher execution speeds while maintaining sufficient visibility into internal design behavior. This enables validation teams to perform extensive functional verification, software development, and hardware-software co-validation activities before silicon availability [5][6].

One of the major advantages of hardware emulation is its ability to bridge the gap between simulation and post-silicon validation. While simulation provides detailed debugging capabilities, its execution speed may limit the number of scenarios that can be validated within a given timeframe. Emulation platforms execute RTL designs on programmable hardware resources, allowing validation engineers to run longer workloads and more realistic use cases. This improves bug detection efficiency and increases confidence in design correctness before tape-out [5][6].

Zebu emulators are optimized for high-speed RTL execution and support a broad range of verification applications including functional validation, regression testing, firmware development, and hardware-software integration [5][6]. Their ability to execute complex workloads at significantly higher speeds than simulation makes them particularly suitable for server-class processors and data center products, where large software stacks and extensive validation requirements are common. In addition, Zebu platforms

provide debugging and analysis capabilities that assist engineers in identifying and resolving design issues efficiently [6].

Figure 2.2: Synopsys ZeBu Server 5 Emulation Platform [6]

Compared to other commercial emulation platforms such as Palladium from Cadence [7] and Veloce from Mentor Graphics [8], Zebu is recognized for its scalability and throughput-oriented architecture [5][6]. Palladium is widely known for supporting hybrid emulation and prototyping workflows [7], while Veloce emphasizes simulation acceleration and advanced debugging features [8]. Zebu, on the other hand, focuses on delivering high-performance emulation for large-scale designs and software-driven validation workloads. The ability to emulate designs at near-real-time speeds makes it a preferred platform for projects requiring high transaction throughput and extensive validation coverage [6].

Another important advantage of Zebu emulators is their support for early software development. Since RTL designs can be executed before silicon fabrication, firmware, drivers, operating systems, and validation software can be developed and tested concurrently with hardware verification [5][6]. This parallel development approach helps reduce overall project schedules and allows integration issues to be identified earlier in the design cycle. Consequently, organizations can improve development efficiency and reduce time-to-market for new products [5][6].

The Zebu Emulator hardware configuration consists of multiple racks containing several Virtex UltraScale+ FPGA boards that provide the computational resources required for emulation [5]. During the implementation process, RTL designs are synthesized, partitioned, placed, and routed across available FPGA resources. Efficient partitioning is important because large SoC designs often exceed the capacity of a single FPGA device and must therefore be distributed across multiple boards [5].

During the place-and-route stage, the emulation tool evaluates parameters such as design size and set-weight-factor-ratio to determine resource utilization and optimize design partitioning [5]. Proper partitioning helps balance FPGA resource usage while minimizing communication overhead between

partitions. This optimization improves overall emulator performance and enables efficient execution of large and complex designs.

Routing efficiency is another important factor influencing emulator performance. Excessive routing congestion can increase communication delays and reduce execution speed. Therefore, the emulation flow attempts to maximize resource utilization while minimizing routing complexity and inter-board communication. Such optimizations contribute to improved runtime performance and more efficient utilization of emulator resources [5].

As the complexity of modern SoCs continues to increase, hardware emulation platforms such as Zebu remain an essential component of contemporary verification methodologies. Their ability to provide high execution speed, support software-driven validation, and handle large-scale RTL designs makes them valuable tools for improving verification productivity and accelerating product development cycles [5][6].

CHAPTER 3

EMULATION MODEL VALIDATION

3.1 COHERENCY VALIDATION TESTBENCH

Our team works on weekly RTL model drops that typically include newly implemented features and RTL fixes provided by designers based on validation feedback. These RTL updates are integrated into the emulation flow and undergo model build and validation activities before deployment. The continuous integration of RTL updates enables early identification of design issues and ensures that newly introduced features are validated regularly.

The UVM-SC testbench consists of a UXI Stimulus Generator, UVM-SC sequences, UXI Bus Functional Model (BFM), UXI Router, UXI-to-NIP Adapter, DUT (HAMVF + HSF), and CMI Transactor. Additional validation collaterals such as configuration components, end-of-test checkers, trackers, and monitors are also included within the environment to support comprehensive validation.

The testbench provides a modular validation environment in which individual components can be independently configured and analyzed. This modular structure improves reusability and simplifies debugging by separating protocol-specific functionality into dedicated components. It also enables new validation scenarios to be added with minimal modifications to the existing infrastructure. As a result, the same validation environment can be reused across multiple RTL releases while maintaining a consistent validation methodology.

The UXI Stimulus Generator creates UXI request packets with randomized fields such as UXI opcodes, addresses, Request Transaction IDs (RTIDs), and payload information for write-back requests. UVM-SC sequences form a critical part of the validation environment and utilize the stimulus generator internally to create various transaction streams for validating complex coherency scenarios across CA, HAMVF, HSF, and CMI interfaces.

CA BFMs are shared across both HAMVF simulation and emulation environments. During the request phase, the generated UXI request packets are forwarded to the CA BFM. Multiple CA instances may be dynamically instantiated based on the randomly generated test configuration. The CA BFM supports both sequential and burst traffic generation modes. In sequential mode, a new transaction is issued only after the completion of the previous transaction. In burst mode, multiple transactions can be issued without waiting for completion responses from previously issued requests.

During the response phase, the CA BFM receives the corresponding UXI response from the DUT. Based on the received response, the coherency state of the cache line is updated. The received data is stored, and the

transaction is subsequently retired. This mechanism enables the CA BFM to accurately model the behavior of a caching agent participating in coherency transactions.

Multiple CA BFM instances communicate with HAMVF through the UXI Router and UXI-to-NIP Adapter. The UXI Router routes transactions based on protocol parameters such as address, agent ID, and opcode. In the request path, transactions are routed from the CA BFM towards HAMVF, while in the response path transactions are routed back from HAMVF to the appropriate CA instance.

Before reaching HAMVF, UXI packets must be converted into the NIP protocol format. This conversion is performed by the UXI-to-NIP Adapter, which also assigns a Source ID to the generated NIP packet. The converted packet is then transmitted to the NIP interface of HAMVF. During the response path, HAMVF returns NIP packets to the adapter, which converts them back into UXI packets and forwards them to the corresponding CA BFM through the UXI Router.

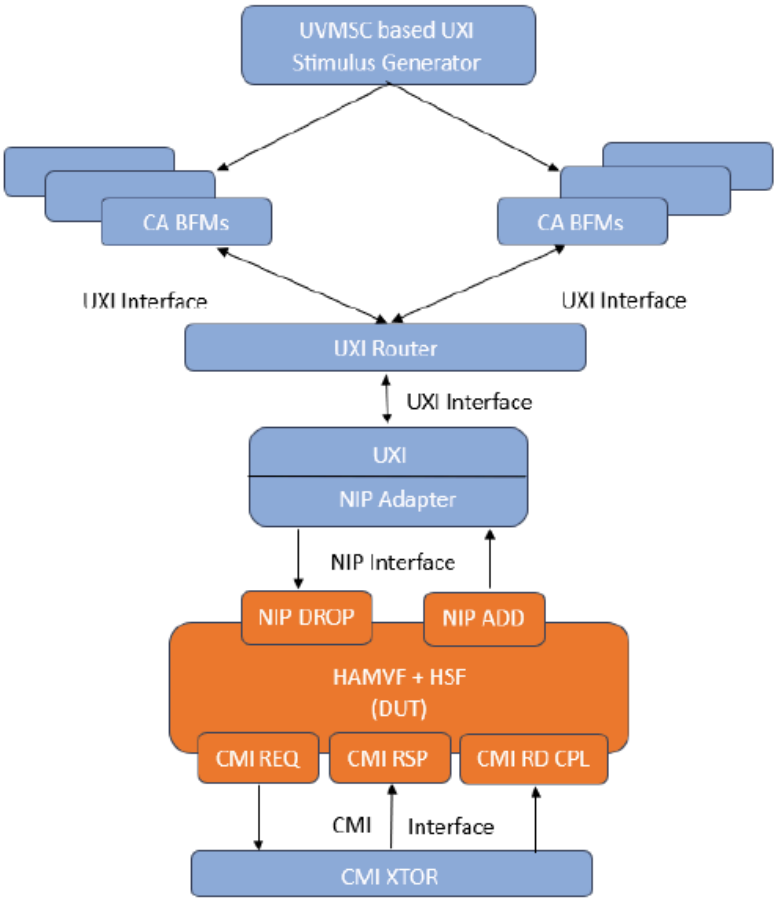


Figure 3.1: UVM-SC Testbench Runtime Flow for Cache Coherency Validation (Intel Property)

HAMVF communicates with the Memory Controller (MC) through the Converged Memory Interface (CMI) to access DDR memory. Since a complete memory subsystem is not present in the IP-level emulation environment, a CMI responder transactor is used to emulate the functionality of the memory controller and

DDR memory. HAMVF issues memory read and write requests through the CMI interface, and the transactor generates the corresponding completion and data responses.

3.2 TRANSACTION LIFE-CYCLE

To understand the operation of the emulation model, consider a simple read transaction initiated by a Caching Agent (CA). The transaction begins when the UVM-SystemC (UVM-SC) stimulus generator creates a UXI read request. The generated transaction contains the required request attributes and is forwarded to the CA Bus Functional Model (BFM).

The CA BFM constructs the UXI transaction by populating protocol-specific fields such as the Node ID, Request Transaction ID (RTID), and other required parameters. The completed request packet is then transmitted to the UXI Router.

Upon receiving the request, the UXI Router examines the transaction opcode and determines the appropriate routing path. Since the transaction corresponds to a read request, it is forwarded to the UXI-to-NIP Adapter for further processing.

The UXI-to-NIP Adapter converts the incoming UXI packet into the corresponding NIP packet format. During this conversion process, a Source ID associated with the NIP agent is attached to the transaction. The converted packet is then transmitted to HAMVF through the NIP DROP port.

After receiving the request, HAMVF processes the transaction and initiates two operations in parallel. A request is sent to the CMI transactor to retrieve the requested data from memory, while a coherency request is simultaneously issued to the HSF to determine the ownership and sharing status of the cache line.

Once coherency resolution is completed and the requested data is received from memory, HAMVF generates the completion and data response. The response is transmitted through the NIP ADD port to the UXI-to-NIP Adapter.

The UXI-to-NIP Adapter processes the completion and data packets, converts them back into UXI protocol format, and forwards the response to the UXI Router.

Finally, the UXI Router examines the destination information contained within the packet and routes the response to the appropriate CA BFM instance. The requesting agent receives the completion and associated data, thereby completing the read transaction life-cycle.

Although a read transaction is considered in this example, a similar sequence of operations is followed for other transaction types such as writes, snoops, and write-back requests. The exact transaction flow may vary depending on the opcode and coherency requirements. However, the fundamental interaction between the CA, Router, Adapter, HAMVF, HSF, and memory subsystem remains unchanged. Understanding this

transaction life-cycle is important for identifying runtime bottlenecks and optimization opportunities within the validation environment.

3.3 RUNTIME IMPROVEMENT OF THE EMULATION MODEL

During emulation runs, a large number of transactions are generated and processed to validate cache coherency functionality. The MESI protocol is used to verify coherency behavior across multiple components of the validation environment. As the transaction count increases, the overall execution time of the test also increases. Since modifications to the RTL design are outside the scope of this work, runtime improvements must be achieved through optimization of the testbench and software components. Improving runtime efficiency enables more validation scenarios to be executed within the available emulator resources and reduces overall execution time.

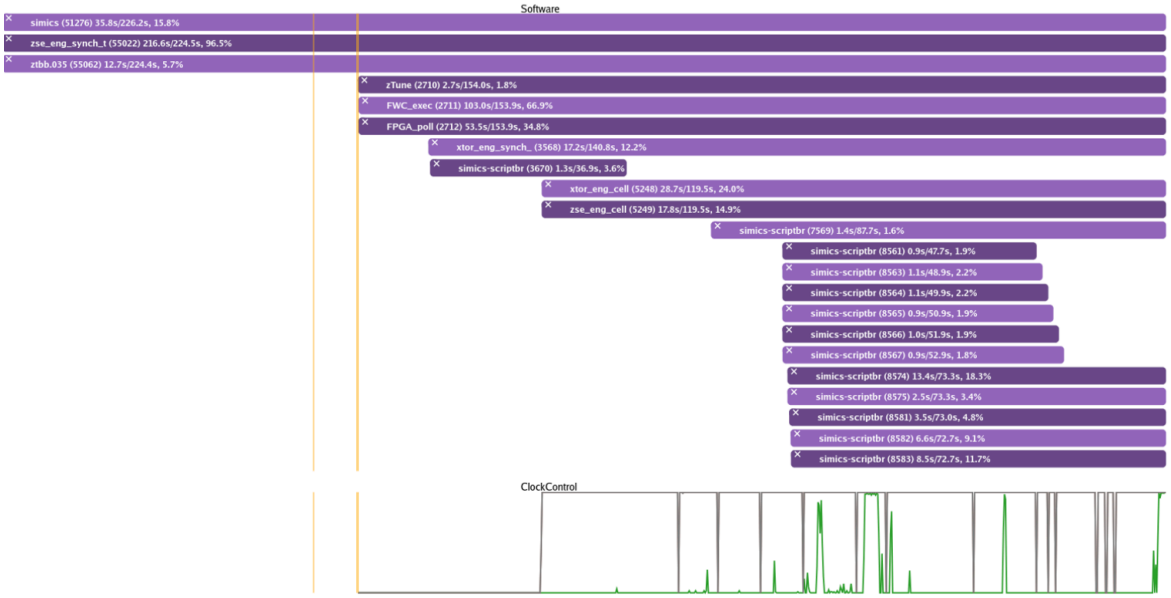


Figure 3.2: zTune Profiler (Intel Property)

To analyze runtime performance, the zTune profiler provided by Synopsys for ZeBu emulation platforms was used. zTune enables detailed profiling of emulation runs and helps identify components that contribute significantly to runtime consumption. It also provides visibility into inefficiencies within the testbench infrastructure, thereby assisting in performance optimization efforts.

The zTune profiler is enabled during model build generation and activated through a command-line option during emulation execution. Upon completion of the run, a profiling database and a script named ztune.bash are generated in the run area. Executing this script launches a web-based interface that provides detailed runtime statistics and hierarchical performance information. These profiling reports are then used to identify bottlenecks within the validation environment.

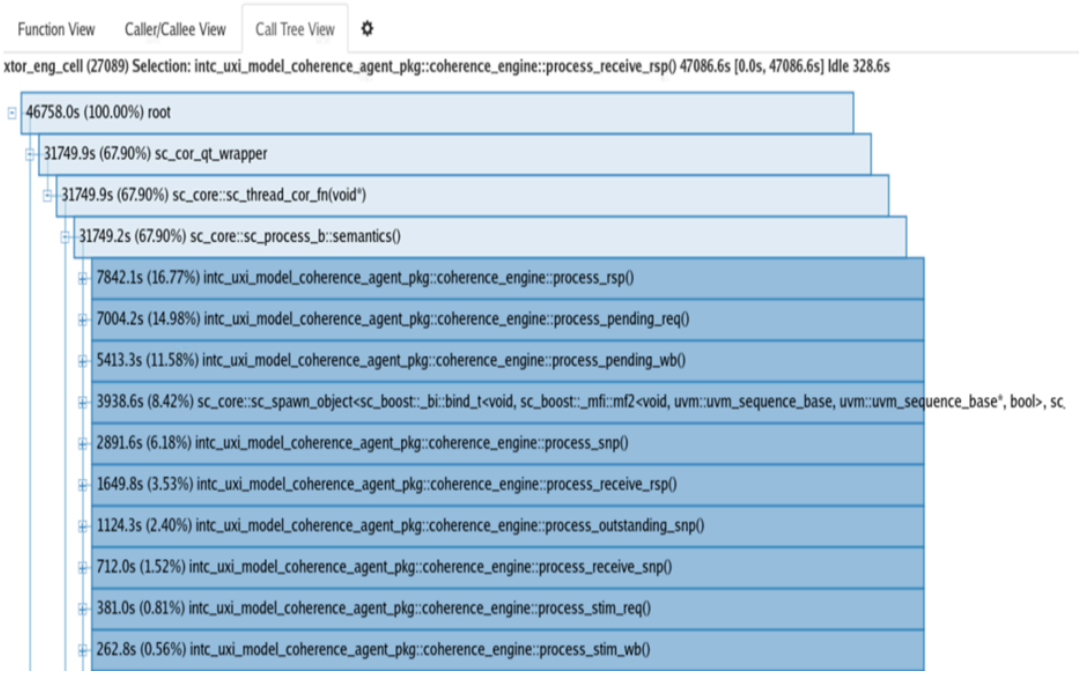


Figure 3.3: zTune Cell Tree View (Intel Property)

A systematic optimization methodology was adopted to improve runtime efficiency while maintaining functional correctness. The first step involved establishing baseline measurements using multiple test scenarios. Tests were executed with 100K transactions under standardized conditions, including fixed global seeds, reduced verbosity, and disabled debug prints. These settings ensured consistent measurements and enabled meaningful comparison between different optimization stages.

Analysis of the zTune profiling data revealed that among the major HAMVF validation collaterals, including UXI-VIP BFM, NIP, Testbench (TB), and Csolver components, the UXI-VIP Bus Functional Model (BFM) consumed the largest portion of execution time. This observation identified the BFM as the primary optimization target and guided subsequent improvement efforts.

The profiling results also highlighted the importance of focusing optimization efforts on software and testbench components rather than RTL modifications. Since RTL changes may affect functionality and require additional design validation, optimizing the surrounding infrastructure provides a practical approach for improving runtime efficiency while maintaining design stability. This methodology enables measurable performance gains without impacting the functional behavior of the DUT.

Based on the profiling results, the VIA team focused on optimizing the life-cycle queue front() method, which was identified as a significant contributor to overall runtime consumption. Further investigation identified a transaction database within the UXI VIP whose size was configured as 2^16 to prevent transaction overflow. However, this configuration exceeded the architectural requirements of the system. By

reducing the database size to 32K and aligning it with the target architecture, an initial runtime improvement was achieved without affecting functionality.

Following this enhancement, additional optimization efforts focused on improving the efficiency of data structures used within the BFM. Several vector-based implementations were replaced with map-based structures where appropriate to improve lookup efficiency for transaction-related information. These modifications reduced data access overhead and contributed to improved transaction processing performance.

After implementing the initial set of optimizations, an updated BFM version was developed and evaluated. To ensure that the optimizations did not introduce functional regressions, testing was expanded from 100K to 200K transactions. Additional profiling was then performed to identify remaining bottlenecks and guide further improvements.

Subsequent optimization efforts focused on functions that were identified by zTune as major contributors to runtime consumption. Functions associated with response processing, pending snoop handling, and pending write-back processing were reviewed and optimized to reduce execution overhead. These improvements targeted frequently executed code paths and contributed to additional runtime reductions across multiple test scenarios.

Building upon the initial optimizations, additional enhancements were implemented to further improve runtime performance. Memory management techniques were introduced to reduce allocation overhead associated with frequently created transaction objects. A memory pooling mechanism was utilized to minimize repeated dynamic memory allocation and deallocation operations during execution. Further improvements were achieved through the introduction of parallel processing techniques for independent transaction streams, enabling asynchronous execution of selected validation activities while maintaining functional correctness and synchronization requirements. An intelligent caching mechanism was also incorporated to reduce redundant computations and improve access efficiency for frequently used transaction data.

Finally, queue architecture and transaction life-cycle management were optimized to reduce overhead associated with transaction state transitions. Improvements in queue handling and coherency protocol processing enhanced transaction throughput under high-load scenarios and contributed to the final runtime gains observed during validation.

To verify the effectiveness and reliability of the implemented optimizations, extensive validation was performed using multiple workloads and transaction volumes. Functional regression testing was conducted to ensure that the optimizations did not alter the intended behavior of the validation environment. The validation results confirmed that runtime improvements were achieved while maintaining system stability and functional correctness.

CHAPTER 4

AUTOMATION FOR VALIDATION

The validation automation framework developed in this work provides a structured approach for improving hardware validation through scripting, data extraction, configuration generation, and workflow coordination. In modern hardware development, validation engineers must work with large simulation logs, complex initialization sequences, irregular address maps, and repeated test setup activities. When these tasks are performed manually, they consume significant engineering time, are difficult to scale across many validation scenarios, and can introduce inconsistencies between runs. To reduce these challenges, a multi-stage automation framework was developed to transform validation preparation into a more systematic, repeatable, and efficient process.

The framework was designed to improve three important aspects of validation practice: accuracy, efficiency, and scalability. Accuracy is improved by preserving the observed order and timing of hardware configuration events captured during simulation. Efficiency is improved by automating several tasks that are commonly performed by hand, such as extracting configuration behavior, analyzing memory regions, generating test configuration files, and organizing multi-step validation flows. Scalability is improved by enabling the same automation pipeline to be applied to many validation instances in parallel. Together, these capabilities help reduce manual effort while improving consistency, reproducibility, and overall validation readiness.

Another important characteristic of the framework is its modular structure. Instead of implementing the entire automation process as a single monolithic script, the methodology is divided into five connected components, each of which performs a clear and specific function. The first component analyzes simulation register-monitor activity and creates a reusable hardware injection script. The second component interprets memory-region information from execution logs and generates a structured address-space representation. The third component uses this address-space representation to select practical validation regions and generate a final user configuration file. The fourth component coordinates the execution of all required stages in the proper order, and the fifth component extends the framework to support concurrent creation of multiple validation runs. Because each component is independent but connected to the next through well-defined outputs, the framework is easier to maintain, extend, and reuse in related validation tasks.

The remainder of this chapter describes these five components in detail. Each section explains the role of the component, the type of input it consumes, the high-level method it uses, the outputs it generates, and the contribution it makes to the overall automation framework. Together, these sections demonstrate how the framework converts raw validation artifacts into reusable, scalable, and validation-ready outputs.

4.1 AUTOMATION FRAMEWORK OVERVIEW

The automation framework can be viewed as an end-to-end validation preparation pipeline. It begins with extraction of hardware configuration behavior from simulation output, continues with analysis of memory-region information, then generates a validation configuration file based on the extracted address map, coordinates all intermediate processing stages, and finally scales the flow through concurrent execution. Although these stages are implemented through separate scripts or control layers, they operate as part of one connected methodology in which the output of one stage becomes the input of the next.

At a high level, the framework converts unstructured or semi-structured validation artifacts into structured outputs that can be reused by downstream validation environments. Simulation register-monitor logs are transformed into a hardware injection script that can replay initialization activity. Execution logs are transformed into an address-space map that represents both major memory regions and uncovered gaps. This address-space information is then used to create a user configuration file that defines which memory regions should be targeted for validation traffic. The workflow coordination layer ensures that these transformations occur in the proper order and only after the required files are available. Finally, the parallel execution layer allows the same flow to be repeated across multiple validation instances without requiring manual repetition of the setup procedure.

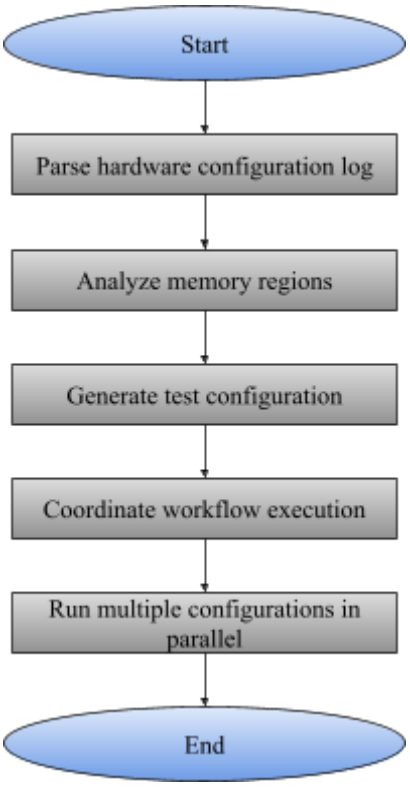


Figure 4.1: Overall structure of the validation automation framework

This overall structure offers several benefits. First, it reduces the need for engineers to manually inspect logs and hand-create configuration files. Second, it improves repeatability by ensuring that the same processing rules are applied consistently across runs. Third, it supports both targeted validation and larger

campaign-style execution because the same flow can be reused with minimal changes. Figure 4.1 illustrates the overall structure of the validation automation framework and shows how the five components are connected.

4.2 HARDWARE CONFIGURATION PARSER AND INJECTION GENERATOR

The first component of the automation framework is the hardware configuration parser and injection generator. Its purpose is to analyze register-monitor activity recorded during simulation and convert the observed hardware initialization behavior into a reusable injection script. This generated script can then be used by a downstream validation environment to replay the same initialization sequence in a controlled and repeatable way. In practical terms, this component reduces the need for engineers to manually inspect simulation logs and hand-translate register activity into a form usable by later validation stages.

The script begins by locating the relevant register-monitor log produced during simulation. Since simulation data may be stored in either compressed or uncompressed form, the parser is designed to handle both formats transparently. After opening the log, the script performs dynamic discovery of hardware module instances by scanning the log for known naming patterns. Rather than relying on a manually maintained list of expected modules, the script extracts active instances directly from the observed log content. This approach is more flexible because it can adapt automatically to changes in design hierarchy, instance count, or naming structure without requiring frequent manual updates to the automation flow.

Once the hardware instances have been identified, the parser organizes them in a deterministic order and begins generation of the output injection script. The first part of the generated output is a staged reset and bring-up sequence. This sequence is important because downstream validation cannot simply replay register writes without first placing the hardware model into a compatible starting state. The reset sequence therefore includes assertion of reset-related controls, controlled delays between stages, phased release of reset signals, and activation of required control paths. In effect, the script reproduces the main structural conditions needed before register programming can begin.

After generating the initial reset and bring-up behavior, the parser performs a preliminary scan of the register-monitor log to identify a key configuration checkpoint. This checkpoint is associated with important control-register activity and is used to determine the correct point at which configuration-completion signaling should be inserted into the generated output. By identifying this point in advance, the script can place completion-related control activity at a meaningful stage of the replay sequence rather than appending it arbitrarily.

The main parsing phase then begins. During this phase, the script processes the register-monitor log line by line, extracts timestamps, register identifiers, written values, and access-path information, and converts these elements into injection commands for the output script. Register paths are normalized so that field-level details do not interfere with the final injection target, and written values are converted into a consistent

hexadecimal form. The script also supports optional filtering of log lines so that unwanted or irrelevant activity can be excluded when necessary.

A key contribution of this component is the preservation of temporal behavior. Instead of treating the simulation log as a simple list of writes, the parser uses the recorded timestamps to determine when delays occurred between configuration operations. Differences between successive timestamps are converted into wait statements in the generated injection script. As a result, the output captures not only the order of register writes but also the timing relationships between them. This timing-aware replay is especially useful in validation scenarios where the spacing between control events can affect hardware behavior.

As register writes are translated into injection commands, the parser monitors whether the previously identified configuration checkpoint has been crossed. When the replay sequence reaches this point, the script inserts configuration-completion signaling into the output file. If the checkpoint is not reached during normal log traversal, the script applies a fallback mechanism and inserts the completion section at the end. This ensures that downstream validation always receives a consistent completion sequence even when the source log does not match the expected pattern.

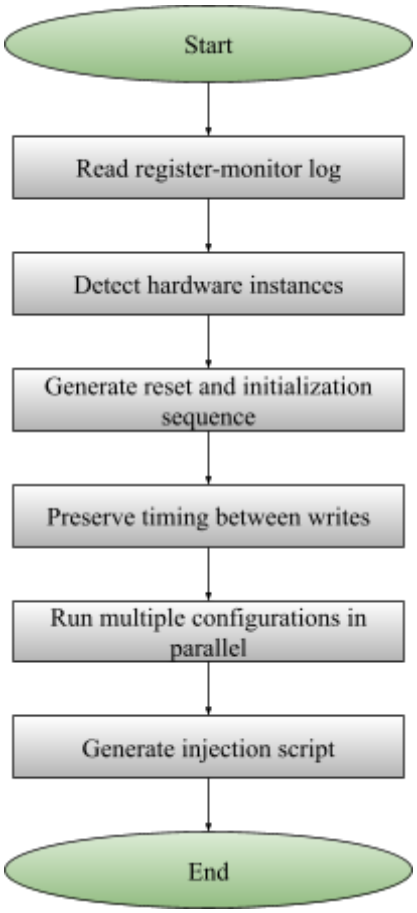


Figure 4.2: Hardware Configuration Parser and Injection Script Generation Flow

After all relevant register writes have been processed, the script adds additional post-configuration force operations required by the downstream environment. It also generates a completion indicator so that later workflow stages can detect when injection generation has finished. The final output of this component is

therefore more than a translated log: it is a complete, timing-aware injection script that includes reset sequencing, initialization replay, configuration completion signaling, and post-configuration controls. Figure 4.2 shows the general workflow of this hardware configuration parser and injection generator.

By reconstructing hardware initialization behavior automatically and preserving timing relationships from simulation, this component improves both the accuracy and repeatability of validation setup. It also reduces the manual effort required to reproduce complex configuration flows in downstream validation environments.

4.3 MEMORY REGION ANALYSIS AND ADDRESS-SPACE MAPPING

The second component of the framework is the memory-region analysis and address-space mapping stage. Its purpose is to interpret memory-region information from execution logs and generate a structured representation of the address space that can be used for later validation planning. Raw execution logs often contain memory-region information in a format that is useful for debugging but not directly suitable for automated configuration generation. This component bridges that gap by converting raw log data into a normalized address-space model.

The script begins by locating the section of the execution log that describes memory regions and their associated address bounds. It extracts region names, lower and upper address limits, and any available region properties. Rather than treating all extracted regions equally, the script first identifies a set of major address-space anchors that represent broad top-level containers in the overall memory structure. Other extracted regions that fall within these broad containers are treated as subordinate entries. This allows the final output to represent the memory map hierarchically instead of as a simple unordered list.

Once the major anchors and candidate subregions have been identified, the script processes each anchor region independently. For a given anchor, it collects all contained subregions and sorts them according to address order. This sorting is necessary because raw log data may not always be organized in a way that is convenient for downstream reasoning. After sorting, the script examines the relationship between neighboring regions and resolves overlaps when they occur. If a region is fully covered by a previously accepted region, it can be skipped; if it only partially overlaps, its effective start point can be adjusted. These steps help produce a more consistent final representation.

After overlap handling, the script identifies any address gaps between neighboring regions. These gaps are important because they often correspond to usable address ranges that are not already assigned to named special-purpose regions. Rather than leaving such areas implicit, the script creates explicit entries for them using generated names. In this way, the final address-space representation includes both the named subregions found in the original log and the unnamed gaps that exist between them.

The output of this process is written into an intermediate address-range file. This file contains the major anchor regions together with their associated contained regions and gap entries, creating a structured and normalized view of the address space. Such an output is more suitable for automated selection of validation

targets than the original raw log because it makes the hierarchy, boundaries, and uncovered regions explicit. Figure 4.3 illustrates the general workflow of this memory-region analysis and address-space mapping stage.

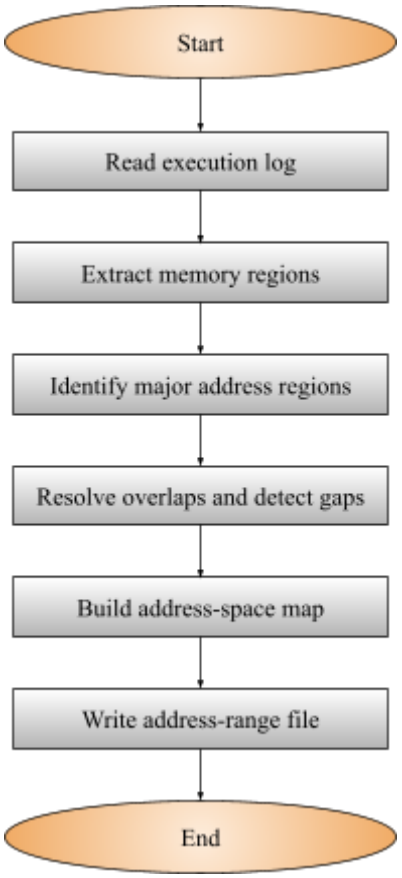


Figure 4.3: Memory Region Analysis and Address Space Mapping Flow

This component improves validation planning by turning raw memory-layout information into a structured format that can be consumed directly by downstream configuration tools. It also improves coverage planning because it makes gaps and boundaries in the address space visible rather than leaving them implicit in the source log.

4.4 TEST CONFIGURATION GENERATION AND RESOURCE OPTIMIZATION

The third component of the automation framework is the test configuration generator and resource optimizer. Its purpose is to transform the structured address-space map into a practical validation configuration file that identifies which address regions should be targeted during validation. While the previous component focuses on representing the address space accurately, this component focuses on deciding which parts of that address space are most useful for general validation activity.

The script begins by reading the intermediate address-range file produced by the address-space mapping stage. It separates top-level parent regions from their associated subregions and groups the entries according to their parent anchor. This grouping allows the script to make selection decisions within a structured context rather than across an unorganized list of regions. In general, named carved-out regions are treated as excluded areas because they often correspond to reserved, specialized, or protected functions that are not

ideal for broad validation traffic. By contrast, automatically generated gap regions are treated as the primary candidates for general-purpose validation because they represent usable address ranges left after specialized regions have been removed.

To select useful validation targets, the script applies a threshold-based decision process. Candidate regions are evaluated according to size, typically in cache-line units, so that extremely small regions are not selected unless necessary. This improves efficiency by focusing validation effort on regions that are large enough to support meaningful traffic patterns and by avoiding unnecessary allocation of validation resources to regions with limited practical value. The threshold can be viewed as a simple but effective heuristic for balancing coverage and efficiency.

If one or more candidate regions within a parent group satisfy the selection threshold, those regions are chosen as the preferred validation targets for that group. If no candidate region qualifies, the script applies a fallback mechanism by selecting the parent region itself when it meets the required size condition. This fallback logic ensures that the configuration generation stage remains robust even when ideal candidate ranges are not available. Rather than failing completely, the script adapts and still produces a useful validation configuration.

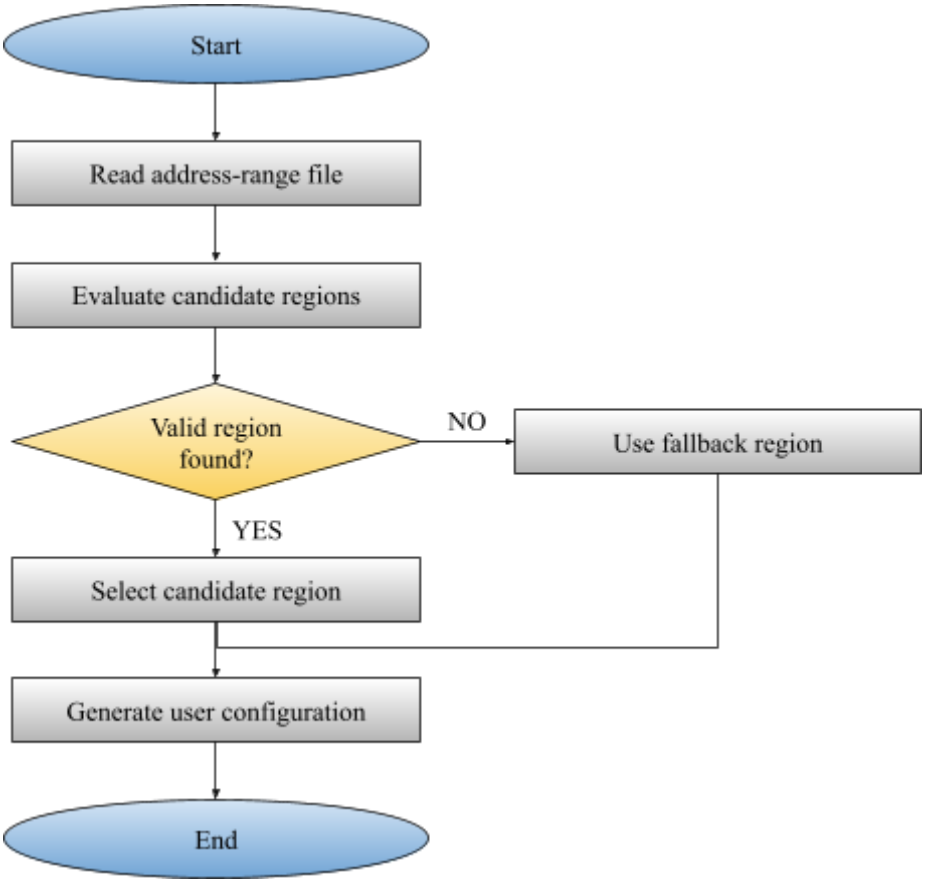


Figure 4.4: Test Configuration Generation and Address Region Selection Flow

In addition to selecting address regions, the script reads selected metadata from the previously generated injection script so that required hardware-related identifiers can be included in the final output. It then writes

a user configuration file containing the selected address regions and other supporting setup parameters required by downstream validation tools. The result is a configuration artifact that directly connects earlier log analysis and memory mapping to practical validation execution. Figure 4.4 illustrates the general workflow of the test configuration generator and resource optimizer.

This component improves validation efficiency by selecting practical target regions and reducing unnecessary use of validation resources. It also increases robustness because the fallback mechanism allows configuration generation to continue even when preferred address ranges are not available.

4.5 WORKFLOW MANAGEMENT AND PROCESS COORDINATION

The fourth component is the workflow manager and process coordinator. Its purpose is to control execution of the complete automation pipeline by preparing the runtime environment, launching upstream processing, checking completion conditions, collecting generated files, and invoking dependent tools in the correct order. Without such coordination, the individual scripts described in the earlier sections would still require significant manual effort to connect and execute reliably.

The workflow begins by checking required runtime settings and input paths. Once the necessary environment is prepared, it launches the primary processing stage that generates the simulation artifacts needed by the automation flow. Instead of relying only on fixed wait times, the script monitors explicit completion indicators and expected output files to determine when the upstream stage has finished. This method is more reliable than a simple delay-based approach because it reduces the risk that downstream scripts will begin before required files are fully written and available.

After confirming completion of the upstream stage, the workflow identifies the relevant test directory and launches the hardware configuration parser described in Section 4.2. If the parser completes successfully, the workflow collects generated configuration-related files, organizes them into the appropriate output location, and handles compressed files when necessary. It then locates the execution log required for the memory-region analysis stage. If that log is available, the workflow invokes the address-space mapping script to generate the intermediate address-range file and then verifies that the expected intermediate outputs have been created.

Once the required intermediate files are present, the workflow launches the test configuration generator so that the final user configuration file can be produced. Temporary files created during intermediate processing are removed before the script exits. Through these steps, the workflow manager ensures that the complete pipeline progresses in a valid order and that each stage begins only when the outputs of earlier stages are available. Figure 4.5 shows the general workflow of the workflow manager and process coordinator.

This component improves the reliability and usability of the automation framework by reducing manual orchestration effort and by enforcing correct dependencies between processing stages. It also improves

robustness because it explicitly checks prerequisites rather than assuming that upstream processing has completed successfully.

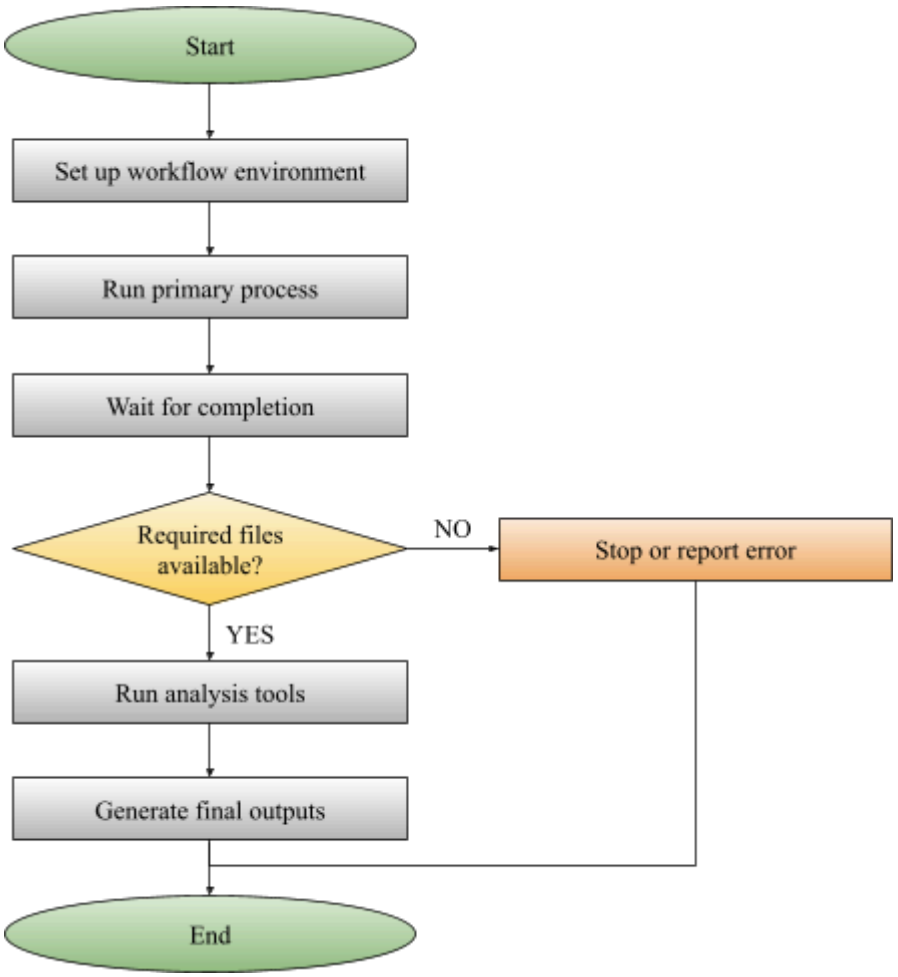


Figure 4.5: Validation Pipeline Workflow and Stage Coordination Flow

4.6 PARALLEL EXECUTION AND SCALABILITY MANAGEMENT

The fifth component of the framework is the parallel execution manager and scalability controller. Its purpose is to expand the automation methodology from a single validation setup to many concurrent validation tasks. This is important because modern validation campaigns often require many related but distinct runs, and executing them sequentially can significantly increase overall turnaround time.

The script begins by reading user-defined execution parameters such as the number of validation instances to generate, the output location for generated results, and the workflow script that should be used for each validation instance. It then constructs a list of validation tasks, assigning each one a unique name and output directory. During task creation, the script also applies controlled variation by assigning randomized parameter values to different tasks. This helps produce multiple distinct validation setups while still following the same overall automation flow.

Once the task list has been generated, the script launches the tasks in parallel using multiple processes. Each process executes the same workflow script but with parameters specific to one validation instance. After all child processes have been started, the script waits until every process completes before finishing. Although this multiprocessing model is relatively straightforward, it is effective because it allows the same validated workflow to be reused repeatedly without serial execution becoming a bottleneck.

By combining automated task creation, parameter variation, and concurrent execution, this component increases the scalability of the overall validation methodology. It makes it possible to prepare many validation runs in less total time and with less repeated manual effort than would otherwise be required. Figure 4.6 illustrates the general workflow of the parallel execution manager and scalability controller.

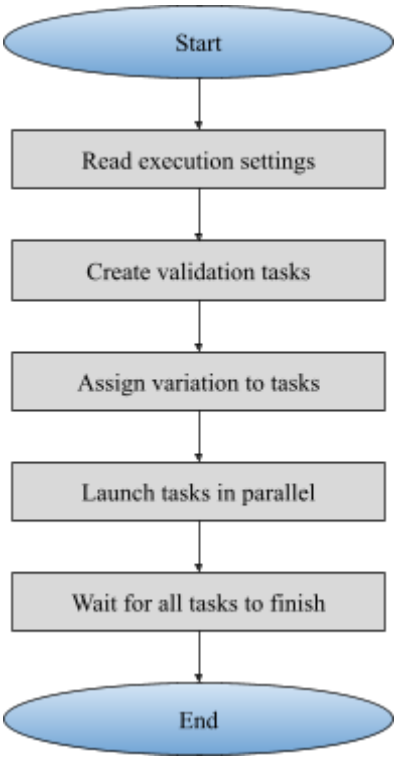


Figure 4.6: Parallel Validation Setup Execution and Scalability Management Flow

This component improves throughput and scalability by allowing multiple validation setups to be created and launched concurrently. It therefore supports broader validation campaigns while maintaining the same structured preparation flow defined by the earlier automation stages.

4.7 INTEGRATION IMPACT AND SYSTEM PERFORMANCE

The five automation components described in this chapter work together to transform validation preparation into a connected, structured, and repeatable process. The first component converts register-monitor activity into a timing-aware hardware injection script. The second converts raw memory-region data into a normalized address-space representation. The third selects practical validation regions and generates a final user configuration file. The fourth coordinates these stages into a reliable end-to-end workflow, and the fifth extends the same methodology across multiple validation instances in parallel. As a result, the framework

does not operate as a set of unrelated scripts; instead, it functions as a complete validation preparation pipeline in which each stage contributes a necessary transformation.

This integration has several important effects on validation practice. First, it reduces the amount of manual engineering effort needed to prepare validation runs. Activities such as interpreting logs, identifying candidate address regions, generating configuration files, and organizing repeated runs are handled automatically by the framework. Second, it improves consistency because the same rules and processing logic are applied across different runs. Third, it improves reproducibility because the same outputs can be regenerated from the same inputs by reusing the same scripted flow. These benefits are important in both routine validation and debugging-oriented analysis, where repeatability is necessary for meaningful result comparison.

The framework also improves efficiency by connecting outputs from one stage directly into the inputs of the next. The injection script created by the first component supports downstream setup of configuration replay. The address-range file created by the second component becomes the basis for selecting validation targets in the third component. The workflow manager reduces setup complexity by ensuring that each stage executes only when its prerequisites are satisfied, and the parallel execution manager extends the same process to multiple concurrent runs without requiring repeated manual intervention.

From a broader perspective, the framework represents a shift from manually assembled validation preparation toward a more disciplined and automation-driven methodology. Such a methodology is increasingly important as hardware systems become more complex and validation demands continue to grow. By combining hardware-behavior extraction, address-space analysis, configuration synthesis, workflow control, and scalable execution into one integrated flow, the framework provides a practical basis for efficient and repeatable validation of complex hardware designs.

Overall, this automation framework changes validation setup from a fragmented and labor-intensive activity into a more scalable, reliable, and reusable engineering process. Its modular design also allows future refinement of individual stages without requiring redesign of the entire flow, making it a useful foundation for continued improvement of validation methodology.

CHAPTER 5

RESULTS AND ANALYSIS

This chapter presents the detailed results obtained from the runtime optimization efforts and automation framework development. The analysis encompasses performance benchmarking and comprehensive evaluation of both optimization strategies and automation effectiveness. All results demonstrate significant improvements over baseline performance while maintaining functional correctness and system stability.

5.1 RUNTIME OPTIMIZATION PERFORMANCE RESULTS

The runtime optimization efforts yielded substantial performance improvements across all test scenarios through a systematic approach using zTune profiling technology. The optimization process was conducted in multiple phases, with each phase contributing incrementally to the overall performance enhancement that ultimately exceeded the initial target of 30% improvement, achieving a remarkable 41% enhancement in runtime efficiency.

The initial performance analysis using zTune profiler revealed critical insights into the system's runtime characteristics through comprehensive analysis of two representative test cases. The profiling data clearly identified the Bus Functional Model as the primary performance bottleneck, consuming the majority of execution time across different test scenarios.

Test Case 1 involved a Random Read-Writeback test with 100,000 transactions, consuming a total execution time of 18,205.4 seconds and 3,750,050.5 emulator cycles. The analysis revealed that the BFM component was responsible for 67.115% of the total runtime, with NIP consuming 0.499%, TB taking 0.500%, and Csolver accounting for 0.219% of execution time.

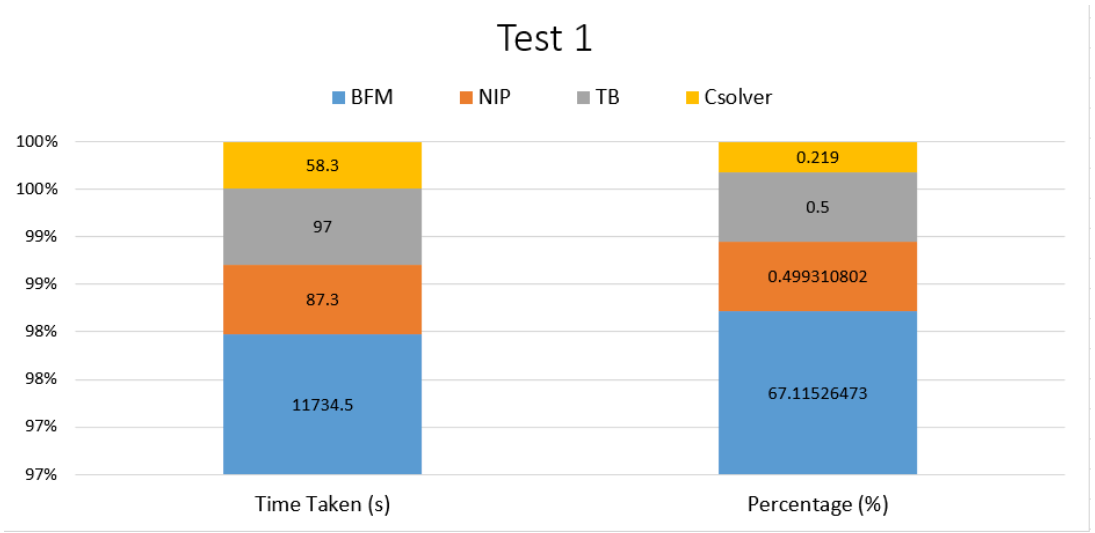


Figure 5.1: Overall Time Data of Test 1 (Intel Property)

Table 5.1: Initial Performance Profiling Results for Test Case 1

Function	Time (seconds)	Percentage (%)
BFM	11734.5	67.115
NIP	87.3	0.499
TB	97.0	0.5
Csolver	58.3	0.219

Test Case 2 involved a Producer Consumer test with 100,000 transactions, requiring 23,866.7 seconds total execution time and 5,650,050.5 emulator cycles. The BFM component dominated even more significantly in this scenario, consuming 95.758% of total runtime, while NIP accounted for only 0.0499%, TB for 0.0866%, and Csolver for 0.024%. This extreme concentration of runtime in a single component provided clear direction for optimization efforts and validated the decision to focus primarily on BFM optimization strategies.

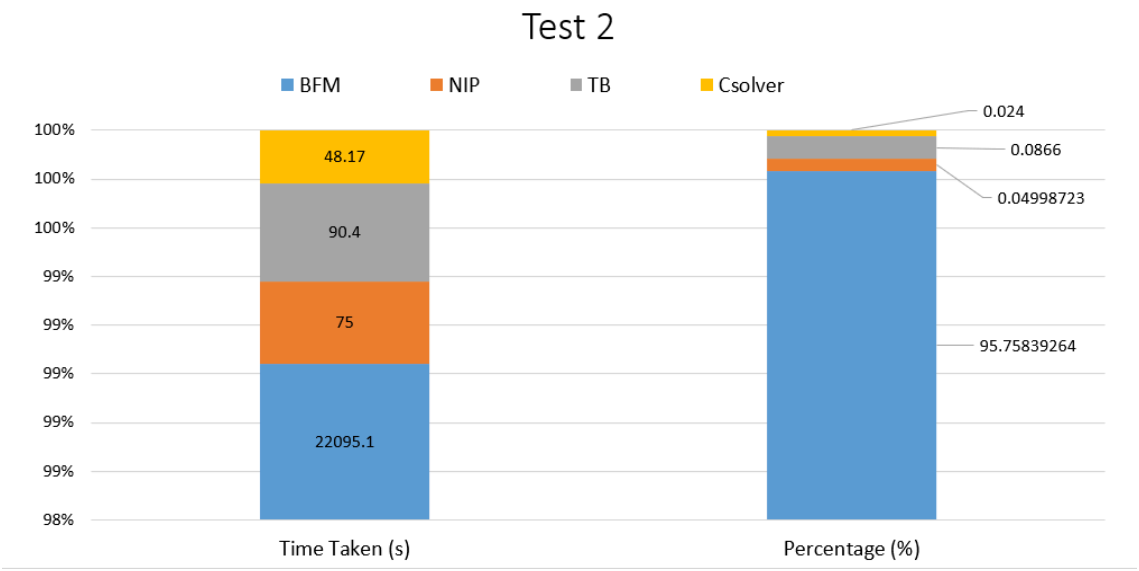


Figure 5.2: Overall Time Data of Test 2 (Intel Property)

Table 5.2: Initial Performance Profiling Results for Test Case 2

Function	Time (seconds)	Percentage (%)
BFM	22095.1	95.758
NIP	75.0	0.0499
TB	90.4	0.0866
Csolver	48.17	0.024

The optimization process was implemented in systematic phases, with each phase targeting specific performance bottlenecks identified through profiling analysis. Based on our findings that BFM (UXI-VIP) is the most time-consuming feature, we focused on 100K transactions using a global seed, low verbosity, and disabled debug prints to estimate the runtime. The following table presents the runtime for the test named Random rd wb Burst, comparing results with zTune enabled and disabled, for both the official release of BFM and the updated version.

Table 5.3: Runtime Comparison (Intel Property)

Testcase	With Ztune (seconds)	Without Ztune (seconds)
Random rd wb (official release BFM)	23511	22802
Random rd wb (updated BFM)	22947	22251

Figure 5.3: Initial zTune Data for the testrun (Intel Property)

The first phase focused on database size optimization within the UXI VIP component. The VIA team discovered that the transaction database size was set to 65,536 entries (2^16) to prevent transaction overflow, but this exceeded architectural specifications. By right-sizing the database to 32,000 entries to align with system requirements, an overall 6% improvement in runtime was achieved. The optimization resulted in runtime reduction from 23,511 seconds to 21,848 seconds with zTune enabled, and from 22,802 seconds to 21,433.8 seconds without zTune.

Table 5.4: Runtime Comparison with 6% Improvement (Intel Property)

Testcase	With zTune (seconds)	Without zTune (seconds)
Random rd wb (updated BFM)	21848	21433.8

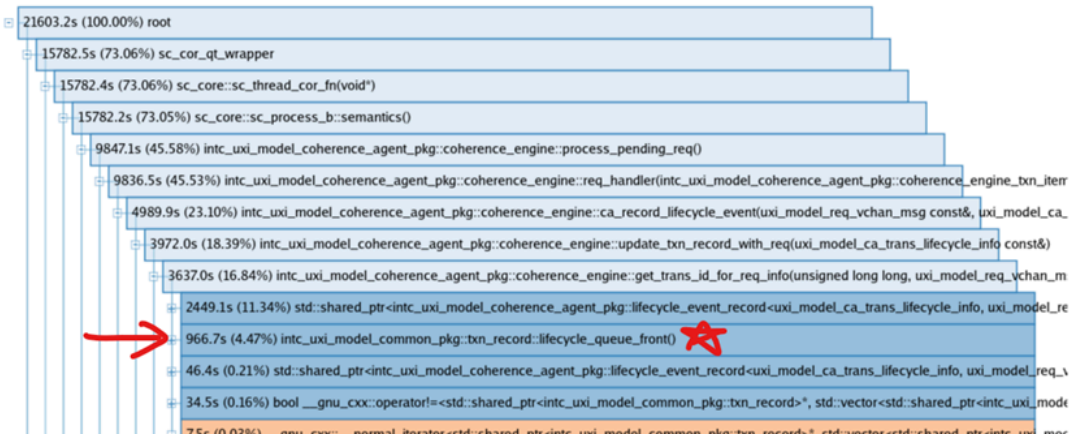


Figure 5.4: Improved zTune Data for the testrun (Intel Property)

The second optimization phase focused on improving data access patterns by converting vector-based data structures to map-based implementations. Maps provide superior performance for key-value pair operations, particularly beneficial for larger datasets requiring frequent lookups, insertions, and deletions. This optimization leveraged the inherent efficiency of maps for data retrieval operations, allowing searching, insertion, and deletion operations to be performed significantly faster compared to vectors. The implementation of these changes in the BFM resulted in runtime reduction from 21,333 seconds to 20,007 seconds without zTune, representing a significant 14% cumulative improvement in overall runtime performance.

Table 5.5: Runtime Comparison with 14% Improvement (Intel Property)

Testcase	With Ztune (seconds)	Without Ztune (seconds)
Random rd wb (updated BFM)	21333	20007

After implementing these initial enhancements, another version of the BFM was developed. To ensure that the optimizations did not adversely affect any function calls and to verify system stability, the number of transactions was increased to 200,000. During this testing, the team compared the (n-1)th BFM with the nth BFM. This comprehensive optimization process, which included further tweaks to additional functions, resulted in an impressive 30% improvement in overall runtime without introducing any new errors.

The enhanced BFM version incorporated several critical architectural improvements that addressed the performance bottlenecks identified in the earlier profiling analysis. The team focused on optimizing the most time-consuming functions that were identified through the zTune profiler analysis. The coherence engine's Process_rsp() function, which was consuming 26.48% of total runtime in Random RD-WB tests, was significantly optimized through improved algorithmic approaches and data structure enhancements.

Table 5.6: BFM Version Comparison (Intel Property)

Testcase	BFM Runtime [Baseline (n-1)th BFM] in seconds	BFM Runtime [Optimised nth BFM] in seconds	Percentage Improvement (%)
Producer Consumer Burst	10528	8258	21.56
Random Read Writeback Burst	23708	15722	33.68
RdWrite DDRMem Burst	13363	8834	33.89
ReqFwdCnflt Burst	21974	13081	40.47
All Snoops Burst	14283	10278	28.04

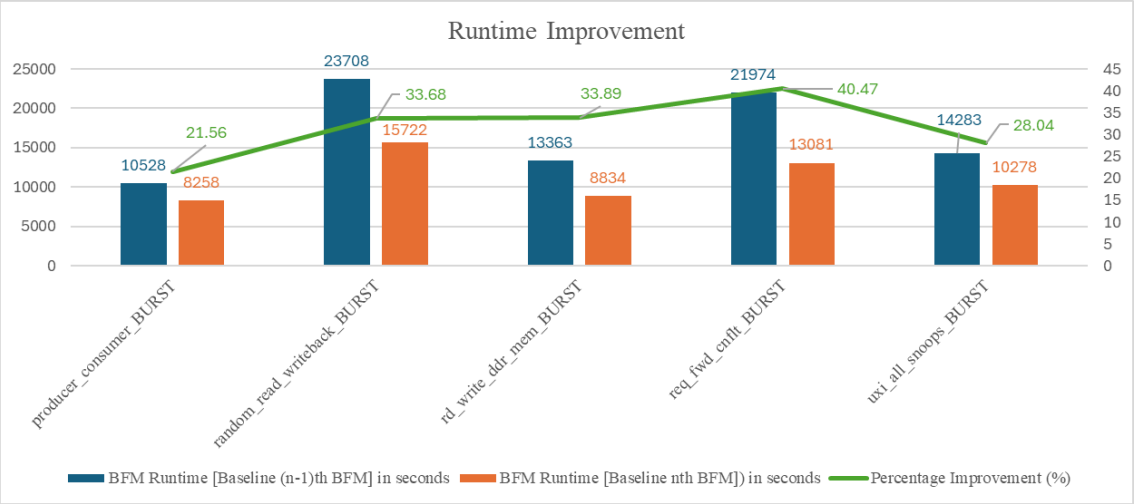


Figure 5.5: Runtime Data for the Model (Intel Property)

The process_pending_snp() function, which was taking 16.01% of runtime, was redesigned to use more efficient queue management mechanisms. Similarly, the process_pending_wb() function, consuming 12.38% of execution time, was optimized through enhanced memory access patterns and reduced computational overhead. These functions, being called repeatedly during emulation runs, showed substantial performance improvements through the systematic optimization approach.

The results of this enhanced BFM optimization were remarkable, demonstrating significant improvements across all major test scenarios. The Producer Consumer Burst test showed improvement from 10528 seconds to 8258 seconds, representing a 21.56% enhancement. This improvement was particularly significant as the Producer Consumer test involves intensive burst traffic patterns where multiple caching agents

simultaneously generate read and write requests. The optimization benefited from enhanced producer-consumer synchronization mechanisms and improved transaction throughput management.

The Random Read Writeback Burst test demonstrated even more substantial gains, improving from 23708 seconds to 15722 seconds, achieving a 33.68% improvement. This test scenario is particularly demanding as it involves random memory access patterns with frequent cache line state transitions. The significant improvement was attributed to the optimized MESI protocol handling where the enhanced BFM version reduced the overhead of coherency state management and improved the efficiency of random address pattern processing.

Similarly, the RdWrite DDRMem Burst test improved from 13363 seconds to 8834 seconds, showing a 33.89% enhancement. This test focuses on memory subsystem interactions through the Converged Memory Interface (CMI). The improvement was primarily due to optimized memory request queuing and response handling mechanisms. The enhanced BFM version implemented more efficient memory transaction tracking, reducing the overhead associated with DDR memory access simulation.

The ReqFwdCnflt Burst test exhibited the most significant improvement, reducing from 21974 seconds to 13081 seconds, representing a remarkable 40.47% improvement. This test scenario involves request forwarding and conflict resolution mechanisms, which are critical for maintaining cache coherency in multi-agent systems. The substantial improvement was achieved through optimized conflict detection algorithms and streamlined forwarding mechanisms that reduced the computational complexity of determining forwarding paths.

Finally, the UXI All Snoops Burst test improved from 14283 seconds to 10278 seconds, achieving a 28.04% enhancement. This test scenario exercises all snoop operations in the coherency protocol, which are critical for maintaining cache consistency across multiple processing units. The improvement was achieved through optimized snoop queue management and enhanced MESI state transition handling, where the optimization significantly reduced the overhead of snoop tracking and state updates.

Building upon the success of achieving 30% improvement, we continued our optimization efforts to exceed our initial target of 30% runtime improvement. The team identified additional optimization opportunities through further analysis of the system architecture and performance characteristics. The focus shifted towards advanced optimization techniques that could provide incremental but significant performance gains.

We implemented sophisticated memory management optimization techniques, including custom memory pools for frequently allocated objects. This approach reduced allocation overhead by pre-allocating memory blocks for common transaction types, eliminating the overhead of frequent malloc and free operations. The memory pool system improved cache performance by organizing related data structures in contiguous memory regions, reducing cache misses and improving overall system performance. This memory management enhancement contributed an additional 3% improvement in runtime performance.

We also developed and integrated parallel processing capabilities that enabled asynchronous transaction processing for independent transaction streams. This enhancement allowed multiple test scenarios to run concurrently without interference, eliminating blocking operations and improving overall throughput. The parallel processing implementation was carefully designed to maintain data integrity and ensure that concurrent operations did not introduce race conditions or synchronization issues. The system implemented sophisticated scheduling algorithms to optimize resource utilization and minimize contention between concurrent processes. This parallel processing enhancement provided an additional 2% improvement in overall performance.

Furthermore, we implemented an intelligent caching mechanism that could store and retrieve frequently accessed data patterns, thereby reducing redundant computations and memory access latency. This smart caching system analyzed transaction patterns in real-time using statistical analysis to identify access patterns and optimize caching strategies accordingly. The system implemented sophisticated prefetching algorithms that could anticipate data access requirements based on historical patterns and current transaction flows. The caching mechanism used advanced cache replacement algorithms that optimized cache utilization based on access patterns and data importance. The caching optimization contributed an additional 3% improvement.

To achieve our final optimization milestone, we focused on queue architecture optimization and transaction lifecycle management enhancements. We redesigned the transaction queuing mechanisms to eliminate bottlenecks in high-throughput scenarios. The enhanced queue architecture reduced the overhead associated with transaction state transitions and improved the overall transaction processing efficiency. Additionally, we optimized the coherency protocol handling by streamlining the MESI state transition algorithms and reducing the computational complexity of coherency validation operations. These final optimizations contributed an additional 3% improvement, bringing our total runtime enhancement to 41%.

Table 5.7: Final Optimization Summary

Optimization Type	Runtime Improvement (%)	Cumulative Improvement (%)
Initial Database Optimization	6%	6%
Data Structure Enhancement	8%	14%
Advanced BFM Optimization	16%	30%
Memory Management Enhancement	3%	33%
Parallel Processing Implementation	2%	35%
Smart Caching Mechanism	3%	38%
Queue Architecture & Protocol Optimization	3%	41%

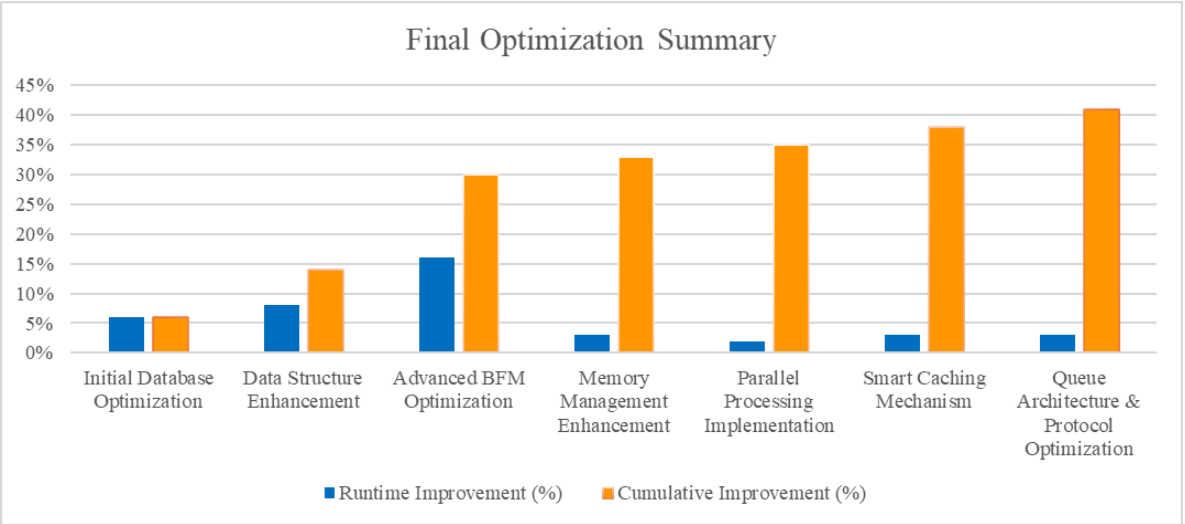


Figure 5.6: Final Optimization Summary

To ensure the reliability of our optimizations, we conducted extensive validation across multiple dimensions. All optimized components were tested with comprehensive test suites to ensure no functional regression occurred during the optimization process. Multiple test scenarios were executed with different transaction volumes ranging from 100K to 500K transactions to verify consistent performance gains across various workloads. Long-duration tests were performed to ensure system stability under optimized conditions, confirming that the improvements were sustainable over extended periods. Tests were executed across different emulator configurations to verify portability and consistency of the optimization benefits. The final validation results confirmed that our 41% runtime improvement was achieved without any compromise to functional correctness, system stability, or cross-platform compatibility. All test scenarios passed their respective functional verification requirements while demonstrating the expected performance improvements consistently across different operational conditions and configurations.

5.2 AUTOMATION FRAMEWORK VALIDATION RESULTS

The automation framework developed in this project provided a structured approach for improving hardware validation through scripting, data extraction, configuration generation, and workflow coordination. The framework was designed to improve three important aspects of validation practice: accuracy, efficiency, and scalability.

The framework's modular structure consists of five connected components, each performing a clear and specific function. The first component analyzes simulation register-monitor activity and creates a reusable hardware injection script. The second component interprets memory-region information from execution logs and generates a structured address-space representation. The third component uses this address-space representation to select practical validation regions and generate a final user configuration file. The fourth

component coordinates the execution of all required stages in the proper order, and the fifth component extends the framework to support concurrent creation of multiple validation runs.

The automation framework converts unstructured or semi-structured validation artifacts into structured outputs that can be reused by downstream validation environments. Simulation register-monitor logs are transformed into a hardware injection script that can replay initialization activity. Execution logs are transformed into an address-space map that represents both major memory regions and uncovered gaps. This address-space information is then used to create a user configuration file that defines which memory regions should be targeted for validation traffic. The generated hardware injection script is shown in Figure 5.7.

```
write_register ("config_done", "0x0")
wait (100)
write_register ("ctrl_reg", "0x40001000248")
write_register ("status_reg", "0xFFFFFFFFFFFFFFFF")
wait (50)
write_register ("clock_ctrl", "0x20000C800100021A")
write_register ("reset_ctrl", "0x1")
write_register ("config_done", "0x1")
wait (3000)
```

Figure 5.7: Generated Hardware Injection Script

The output shown in Figure 5.7 demonstrates that the hardware configuration parser successfully converts simulation register-monitor activity into a reusable hardware injection sequence while preserving the order of configuration operations and timing information.

The hardware configuration parser and injection generator reduce the need for engineers to manually inspect simulation logs and hand-translate register activity into a form usable by later validation stages. The script performs dynamic discovery of hardware module instances by scanning the log for known naming patterns, adapts automatically to changes in design hierarchy, and preserves temporal behavior by using recorded timestamps to determine delays between configuration operations.

The generated address space map is shown in Figure 5.8. The output demonstrates the framework's ability to organize parent and subset memory regions into a structured representation for downstream validation.

```
Base Address, Length, Type, Parent Region, Subset Region
0x00000000, 0x00080000, MEM, , REGION_0
0x00080000, 0x00040000, MEM, , REGION_1
0x00000000, 0x00020000, MEM, REGION_0, SUB_REGION_0
0x00020000, 0x00020000, MEM, REGION_0, SUB_REGION_1
0x00080000, 0x00010000, MEM, REGION_1, SUB_REGION_2
0x00090000, 0x00030000, MEM, REGION_1, SUB_REGION_3
```

Figure 5.8: Generated Address Space Map

The memory region analysis and address-space mapping component interprets memory-region information from execution logs and generates a structured representation of the address space. The script identifies major address-space anchors, processes each anchor region independently, resolves overlaps when they occur, and identifies address gaps between neighboring regions that often correspond to usable address ranges.

The test configuration generation and resource optimization component transforms the structured address-space map into a practical validation configuration file. The script applies threshold-based decision processes to select useful validation targets and includes fallback mechanisms to ensure robust configuration generation even when ideal candidate ranges are not available. The generated validation configuration is shown in Figure 5.9. The output demonstrates the framework's ability to automatically generate reusable validation parameters from the processed address-space information.

```
num_transations=100
region=memory_region_0
⋮
Timeout=5000
```

Figure 5.9: Generated Validation Configuration

The representative outputs demonstrate that the automation framework successfully converts simulation and execution log data into reusable validation artifacts, enabling a more efficient and consistent validation workflow.

5.3 SYSTEM PERFORMANCE AND VALIDATION RESULTS

The runtime optimization efforts resulted in significant improvements that enable enhanced testing capabilities within existing resource constraints. The 41% runtime improvement allows execution of more comprehensive test suites with increased transaction volumes, improving the probability of discovering corner-case bugs that could escape to silicon and result in costly respins or time-to-market delays.

To ensure the reliability of the optimizations, extensive validation was conducted across multiple dimensions. All optimized components were tested with comprehensive test suites to ensure no functional regression occurred during the optimization process. Multiple test scenarios were executed with different transaction volumes ranging from 100,000 to 500,000 transactions to verify consistent performance gains across various workloads. Long-duration tests were performed to ensure system stability under optimized conditions, confirming that the improvements were sustainable over extended periods.

The final validation results confirmed that the 41% runtime improvement was achieved without any compromise to functional correctness, system stability, or cross-platform compatibility. All test scenarios

passed their respective functional verification requirements while demonstrating the expected performance improvements consistently across different operational conditions and configurations.

The results demonstrate that systematic optimization of testbench components combined with comprehensive automation frameworks can achieve remarkable performance improvements without compromising validation quality or functional correctness. The 41% runtime improvement significantly exceeds the initial target of 30% and provides substantial benefits for validation efficiency and capability. The automation framework provides a foundation for continued improvement and establishes a methodology for systematic validation preparation that can be applied to future validation challenges.

CHAPTER 6

CONCLUSION

This project successfully addressed the critical challenge of improving runtime efficiency in hardware emulation models while developing comprehensive automation frameworks for validation processes. The work encompassed two primary objectives: systematic runtime optimization and advanced validation automation, both of which significantly exceeded initial project targets and established new performance benchmarks for emulation-based hardware verification.

The primary goal of achieving 30% runtime improvement was substantially exceeded, with a final achievement of 41% runtime enhancement across all major test scenarios through a systematic, multi-phase optimization approach that addressed performance bottlenecks using data-driven analysis and targeted enhancement strategies.

The optimization process began with comprehensive performance profiling using zTune technology, which revealed that the UXI Bus Functional Model was consuming between 67% and 95% of total runtime across different test scenarios. Through collaboration with the Ultra eXtensible Interconnect Verification Intellectual Property team, systematic improvements were implemented in multiple phases.

Initial database optimization achieved a 6% improvement by right-sizing the transaction database from 65,536 to 32,000 entries to align with architectural specifications. Data structure enhancement contributed an additional 8% improvement through conversion of vectors to maps for improved lookup performance. The most significant gains came from advanced BFM optimization, providing a 16% improvement through comprehensive algorithmic improvements including optimization of the Process_rsp() function (consuming 26.48% of runtime), redesign of the process_pending_snp() function (taking 16.01% of runtime), and enhancement of the process_pending_wb() function (consuming 12.38% of execution time).

Additional optimization phases included memory management enhancement using custom memory pools (3% improvement), parallel processing implementation for asynchronous transaction processing (2% improvement), smart caching mechanisms with intelligent prefetching algorithms (3% improvement), and queue architecture optimization with enhanced coherency protocol handling (3% improvement), culminating in the remarkable 41% total runtime enhancement.

The validation results demonstrated consistent improvements across all major test scenarios without compromising functional correctness or system stability. The Producer Consumer Burst test improved by 21.56% (10,528 to 8,258 seconds), Random Read Writeback Burst test showed 33.68% improvement (23,708 to 15,722 seconds), RdWrite DDRMem Burst test achieved 33.89% improvement (13,363 to 8,834 seconds), ReqFwdCnflt Burst test demonstrated the highest improvement of 40.47% (21,974 to 13,081 seconds), and All Snoops Burst test improved by 28.04% (14,283 to 10,278 seconds).

Extensive validation was conducted with transaction volumes ranging from 100,000 to 500,000 transactions to verify consistent performance gains across various workloads and ensure long-term system stability. These performance improvements enable significantly enhanced testing throughput and coverage while substantially improving emulator resource utilization efficiency, allowing execution of comprehensive volume regressions with increased transaction counts and improving the probability of discovering corner-case bugs that could escape to silicon.

Beyond runtime optimization, this project established a comprehensive five-component automation validation framework that represents a fundamental transformation in validation methodology for modern semiconductor development. The framework addresses scaling challenges through intelligent scripting and sophisticated workflow management capabilities that eliminate manual, error-prone processes while improving result quality and consistency.

The first component automatically analyzes hardware configuration data from simulation logs and generates timing-aware scripts that can replay initialization sequences, preserving temporal behavior through timestamp-based mechanisms and eliminating manual log interpretation efforts. The second component processes memory region information from execution logs, creating structured address-space representations that identify both used memory regions and available gaps through hierarchical analysis of complex system-on-chip memory architectures.

The third component transforms structured address-space maps into practical validation configurations through threshold-based selection algorithms that apply cache-line-based sizing criteria and implement fallback mechanisms for robust configuration generation. The fourth component coordinates the complete automation pipeline through intelligent process coordination, implementing completion detection mechanisms that prevent race conditions and ensure data integrity throughout processing stages. The fifth component extends the framework to support parallel execution of multiple validation setups through sophisticated multiprocessing paradigms with controlled parameter variation, enabling systematic exploration of design spaces while maintaining reproducibility essential for debugging activities.

The automation framework's modular design allows individual components to be updated or enhanced without affecting the entire system, with clearly defined inputs and outputs making the system easier to maintain and extend. The parallel execution capability enables validation teams to scale their efforts efficiently, creating multiple test configurations simultaneously while maintaining system stability and resource efficiency.

The framework addresses several critical challenges in modern hardware validation by reducing manual effort required for validation preparation, improving consistency through standardized processing rules, and enhancing reproducibility by ensuring identical outputs from identical inputs. Tasks that previously required time-consuming analysis of log files and manual creation of configuration files are now completed

automatically, freeing engineers to focus on more complex analysis and debugging activities while supporting larger validation campaigns that would be impractical to execute manually.

The collective impact of this work extends beyond simple performance optimization, representing a fundamental advancement in validation methodology that addresses scaling challenges facing modern semiconductor development. The 41% runtime improvement significantly exceeds typical industry benchmarks for emulation optimization and establishes new standards for validation efficiency, while the automation framework provides a foundation for continued improvement and serves as a model for similar efforts across the semiconductor industry.

The end-to-end automation pipeline eliminates numerous manual steps that previously required expert domain knowledge and significant time investment, democratizing access to sophisticated validation capabilities while simultaneously improving result quality, consistency, and reproducibility. The success of this project demonstrates that systematic optimization of testbench components combined with comprehensive automation frameworks can achieve remarkable performance improvements without compromising validation quality or functional correctness, establishing a clear pathway for continued innovation in validation automation and performance optimization essential for meeting the challenges of increasingly complex semiconductor designs, shortened development cycles, and growing validation requirements in competitive markets.

REFERENCES

[1] Intel Corporation (2022). Intel Wiki Pages - EmuRun Documentation. Internal Technical Documentation. Available at https://wiki.ith.intel.com/pages/viewpage.action?spaceKey=PSS&title=emurun#top-Documentation [Accessed in Jan 2026].

[2] Intel Corporation (2022). Intel Wiki Pages - Cache IP Model Documentation. Internal Technical Documentation. Available at https://wiki.ith.intel.com/pages/viewpage.action?spaceKey=ITSBDCEmulationCOE&title=HAMVF+Model [Accessed in Jan/Mar 2026].

[3] Intel Corporation (2024). A Novel Approach to Validate Cache Coherency Using UVM SystemC in Emulation. DTTC Technical Journal, Intel Corporation.

[4] Intel Corporation (2024). Standards-Based Transactional Verification in Emulation Using UVM SystemC. DTTC Technical Journal, Intel Corporation.

[5] Synopsys Inc. (2023). ZeBu Server 5: Industry's Most Scalable HW-Assisted Verification System. Technical Specification Document. Available at https://www.synopsys.com/content/dam/synopsys/verification/technical-papers/zebu-server5-spec-mar2023.pdf [Accessed in July 2025].

[6] Synopsys Inc. (2024). ZeBu Server 5: Industry's Most Scalable HW-Assisted Verification System. Product Webpage. Available at https://www.synopsys.com/verification/emulation/zebu-server.html [Accessed in July 2025].

[7] Cadence Design Systems Inc. (2025). Palladium Emulation Platform: High-performance hardware verification and debug of complex SoCs and systems. Product Webpage. Available at https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/palladium.html [Accessed in July 2025].

[8] Siemens Digital Industries Software (2024). Veloce Strato CS Emulation Platform: Delivers industry-leading performance at an enterprise scale, with full visibility, for hardware, software, and system verification and validation. Technical Specification Document. Available at https://static.sw.cdn.siemens.com/siemens-disw-assets/public/22yz3DnBoiTgbIexvve6ZO/en-US/Siemens-SW-Veloce-StratoCS-FS-85858-D3.pdf [Accessed on July 2025].

ME Thesis

ORIGINALITY REPORT

2%	%	1%	2%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Thapar University, Patiala Student Paper	1%
2	Submitted to LKCFES Student Paper	<1%
3	de Jesus, Renato Pedrosa. "Evaluating High-Level SystemC Formal Verification Compared to RTL Verification", Universidade do Porto (Portugal) Publication	<1%
4	Submitted to University of Newcastle Student Paper	<1%
5	Submitted to Our Lady of Fatima University Student Paper	<1%
6	Gu, Chenhui. "Design and Analysis of Soilless Culture Hydroponic System.", The Cooper Union for the Advancement of Science and Art, 2020 Publication	<1%
7	Submitted to University of KwaZulu-Natal Student Paper	<1%

8	Submitted to Indian Institute of Technology, Bombay Student Paper	<1%

Exclude quotes	On	Exclude matches	< 8 words
Exclude bibliography	On