STIMULUS OPTIMIZATION IN HARDWARE VERIFICATION USING MACHINE-LEARNING

A Thesis

by

SAUMIL PANKAJBHAI GOGRI

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Jiang Hu
Co-Chair of Committee,	Aakash Tyagi
Committee Member,	Weiping Shi
Head of Department,	Miroslav Begovic

May 2019

Major Subject: Computer Engineering

Copyright 2019 Saumil Pankajbhai Gogri

ABSTRACT

Simulation-based functional verification is a commonly used technique for hardware verification, with the goal of exercising critical scenarios in the design, detecting and fixing bugs, and achieving close to 100% of the coverage targets required for tape-out. As chip complexity continues to grow, functional verification is also becoming a bottleneck for the overall chip design cycle. The primary goal is to shorten the time taken for functional coverage convergence in the volume verification phase, which in return, accelerates the bug detection in the design. In this thesis, I have investigated the application of machine learning towards this objective.

I accessed the machine learning-guided stimulus generation with two approaches: coarse-grained test-level optimization and fine-grained transaction-level optimization. The effectiveness of machine learning was first confirmed on test-level optimization, which rests on achieving full coverage for a certain group of functional coverage metrics in reduced time with a minimal number of simulated tests. It was observed that test-level optimization was limited to some common functional coverage metrics. This was the motivation to explore and implement transaction-level optimization in two novel ways: transaction pruning and directed sequence generation for accelerated functional coverage closure. These techniques were applied on FSM (Finite State Machine) and Non-FSM based coverage metrics and compared the gains using different ML classifiers. Experimental results showed that the fine-grained implementation can potentially reduce the overall CPU time for the verification coverage closure; thus, I propose that complementary application of both the levels of stimulus optimization is the recommended path for efficiency improvements in functional verification coverage convergence.

I dedicate this thesis to my parents, brother and sister-in-law and the almighty God.

ACKNOWLEDGMENTS

It is my honor and privilege to have pursued my graduate studies at Texas A&M University. I am grateful to a lot many people for their efforts during this journey. First and foremost, my deepest gratitude to my advisor, *Dr. Jiang Hu*, for his continuous support and mentoring during my graduate studies. His trust and encouragement helped me to think beyond the normal conventions from time to time, which proved very beneficial in carrying out my research. I would sincerely thank my co-advisor, *Dr. Aakash Tyagi*, for his persistent optimism and motivation, especially at times when I was in doubt or felt lost. He has been a friend, more than a guide, with whom I shared many professional and personal talks. I feel fortunate to receive directions, at par with industry standards, from both of my advisors without which this thesis would be improbable. It was truly an honor to have research advisors and mentors like them.

I would like to thank *Dr. Weiping Shi* for being a part of my thesis committee and providing continuous constructive feedback on my thesis. I am highly indebted to *Prof. Mike Quinn* for introducing “Hardware Verification” to me and imparting all the valuable knowledge along with being constant support throughout my coursework. I am grateful to him for trusting me and providing access to the RTL design for carrying out research. I am very thankful to my fellow colleagues Swati Ramachandran, Fazia Batool and Amrutha Shikaripura, who are also part of this project. Swati helped me strengthen my UVM basics and assisted with the initial testbench modifications. I would refer our collaborative work as “I” while presenting this thesis. I would also like to thank Sheena Goel for being an excellent Teaching Assistant and sharing her incredible knowledge and feedback while I was pursuing my research. I am thankful to Venky, Yuhao Yang, and other designers of quad-core cache design.

I am very grateful to my parents for providing incredible support and motivation throughout my career till date. One person without which I couldn’t excel through my research is my elder brother, *Maulin Gogri*. He has been an amazing troubleshooter to all my questions and worries. His proactiveness and varied knowledge have always motivated me to push my limits. A special

thanks to my sister-in-law, Puja Sangoi, and cousins Ami and Pooja. I am thankful to all the friends in College Station, Eshan, Andrew, Karan, Darpit, Panki, Bansi, Ankur, Rishabh and many more. College Station felt like home because of them.

I extend my thanks to the ECE and CSE department at Texas A&M University for allowing me to be part of them. I am thankful to ECE graduate advisors, Ms. Katharine Bryan and Ms. Vickie Winston in particular, for guiding me through the university formalities and arranging logistics as and when required.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Jiang Hu, and Professor Weiping Shi of the Department of Electrical and Computer Engineering (ECE), and Professor Aakash Tyagi of the Department of Computer Science and Engineering (CSE).

The RTL design and UVM Verification Environment used for this research thesis was provided by Professor Aakash Tyagi and Professor Mike Quinn. Swati Ramachandran and Sheena Goel helped in refining and adding parametric stimulus in the UVM verification environment mentioned in *Appendix A.1*. The Python environment and packages utilized for creating machine learning models were already integrated in ECE servers by the linux team.

All the other work was carried out by the student independently.

Funding Sources

Graduate study was partly supported by a scholarship from the ECE department at Texas A&M University.

NOMENCLATURE

IC	Integrated Circuit
VLSI	Very Large Scale Integration
EDA	Electronic Design Automation
HAS	High-level Architecture Specification
HDL	Hardware Description Language
RTL	Register Transfer Language
SBV	Simulation-Based Verification
FV	Formal Verification
DUT	Design Under Test
UVM	Universal Verification Methodology
ML	Machine Learning
DNN	Deep Neural Network
SVM	Support Vector Machine
RF	Random Forest
RNN	Recurrent Neural Network
LSTM	Long Short Term Memory

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGMENTS iv

CONTRIBUTORS AND FUNDING SOURCES vi

NOMENCLATURE vii

TABLE OF CONTENTS viii

LIST OF FIGURES xi

LIST OF TABLES xiii

1. INTRODUCTION 1

    1.1 Digital IC Design 1
    1.2 Functional Verification 2

2. SIMULATION-BASED VERIFICATION 6

    2.1 Pillars of Verification 6
        2.1.1 Stimulus 6
        2.1.2 Coverage 8
        2.1.3 Checking 9
        2.1.4 Simple Cache and Memory Example 9
    2.2 Verification Cycle 10
    2.3 Verification Environment and UVM 12
        2.3.1 UVM Components 13
        2.3.2 Advantages of using UVM-based Testbench 14
    2.4 Verification Challenges 14

3. MACHINE LEARNING 16

    3.1 Machine Learning Terminologies 16
    3.2 Machine Learning Algorithms 16
    3.3 Supervised Machine Learning Algorithms 18
        3.3.1 Deep Neural Network 18

3.3.2 Random Forest 20
3.3.3 Recurrent Neural Network 21

4. PREVIOUS RELATED WORK 23

5. DESIGN AND EXISTING TESTBENCH 25

5.1 The DUT - Multicore Cache Design 25
5.2 UVM Testbench 26
5.3 Test-Suite 27

6. TEST-LEVEL STIMULUS OPTIMIZATION 29

6.1 Methodology Overview 29
6.2 Test Knobs 30
6.3 Coverage Metrics 31
6.4 Machine Learning Model Training 32
6.5 Coverage Prediction and Test Pruning 33

7. TRANSACTION-LEVEL STIMULUS OPTIMIZATION 36

7.1 Methodology Overview 36
7.2 Per-Transaction Coverage 37
7.3 FSM Transition Coverage 38
7.3.1 Data Extraction and Model Training 39
7.3.2 Coverage Prediction, Transaction Pruning, and Data Modeling 39
7.4 Non-FSM Event Coverage 42
7.4.1 Data Extraction and Model Training 43
7.4.2 Coverage Prediction and Transaction Pruning 44

8. EXPERIMENTAL RESULTS 46

8.1 Test-Level Stimulus Optimization 46
8.2 Transaction-Level Stimulus Optimization 49
8.2.1 FSM Based Coverage Metric 50
8.2.2 Non-FSM Based Coverage Metric 52
8.3 Online Pruning vs Offline Generation 55

9. CONCLUSION 57

REFERENCES 59

APPENDIX A. IMPLEMENTATION CODE 61

A.1 Random Parametric Test Simulation 61
A.2 Coverage Collection per Test 63
A.3 Training Deep Neural Network Model 64
A.4 Training Random-Forest Model 65

A.5 Training LSTM Model 66

LIST OF FIGURES

FIGURE		Page
1.1	IC Design Flow	1
1.2	Reconvergence Model	3
2.1	Three Pillars of Verification	6
2.2	Levels of Stimulus	7
2.3	System-Verilog Coverage	8
2.4	Simple Memory and Direct Mapped Cache Design	10
2.5	Coverage and Bug-rate in the Verification Cycle	11
2.6	Typical UVM Testbench	12
2.7	Challenges Faced in Simulation Based Verification	14
3.1	Flavors of Machine Learning Algorithms	17
3.2	Schematic of a Neuron	19
3.3	Deep Neural Network	20
3.4	Random Forest	20
3.5	LSTM Unit	21
5.1	Multicore MESI Based Cache Design	25
5.2	UVM Verification Testbench	26
6.1	Test-level Stimulus Optimization Methodology	29
6.2	Test-level Stimulus Optimization Flow-diagram	33
7.1	Transaction-level Stimulus Optimization Methodology	36
7.2	FSM Coverage Model	38
7.3	Transaction Attribute Graph	40

7.4 TA Trained Graph ... 41
7.5 TA Predicted Graph ... 42
7.6 Shortest Path Traversal Example ... 43
7.7 Non-FSM Coverage Model ... 43
7.8 Non FSM Coverage: Online Pruning vs Offline Generation ... 45
8.1 Coverage Closure for Cover-Group A with Test-Level Optimization ... 47
8.2 Coverage Closure for Cover-Group B with Test-Level Optimization ... 49
8.3 FSM Transition Coverage Closure using DNN Classifier ... 50
8.4 FSM Transition Coverage Closure using RF Classifier ... 51
8.5 CPU Runtime comparison: RF based FSM Transition Coverage ... 52
8.6 Classification Error with varying w for LSTM and RF Classifier ... 53
8.7 Non-FSM Coverage Closure using LSTM Classifier ... 54
8.8 CPU Runtime comparison: LSTM based Non-FSM Event Coverage ... 55
A.1 Bash Script for Test Simulation ... 61
A.2 Input Matrix - Test Constraints ... 62
A.3 Coverage Report for a Simulated Test ... 63
A.4 Output Matrix - Coverage ... 64
A.5 Instance of DNN model ... 64
A.6 Instance of LSTM model ... 66

LIST OF TABLES

TABLE Page

6.1 Test Knobs ... 31

6.2 Coverage Metrics ... 32

8.1 No. of Tests for Coverage Closure of Cover-Group A ... 48

8.2 Comparison of No. of Tests with and without Machine Learning (RF Model) ... 49

1. INTRODUCTION

1.1 Digital IC Design

Modern day processors, memories and other digital ASICs are a few enormously complex Digital IC designs consisting of several billion transistors manufactured on a few millimeters silicon die. Advances in the VLSI design flow and transistor scaling (currently 7/10nm) have permitted us to bring up such revolutionary ICs. Two key components of any digital IC are the architecture design and the fabrication process. The silicon industry is embarking noteworthy growth in both these facet of digital IC design, putting an added pressure on the engineers to make groundbreaking innovations and build more sophisticated software (EDA) for VLSI design. Based on these components, a typical digital IC design flow is divided into two phases: Front-End and Back-End.

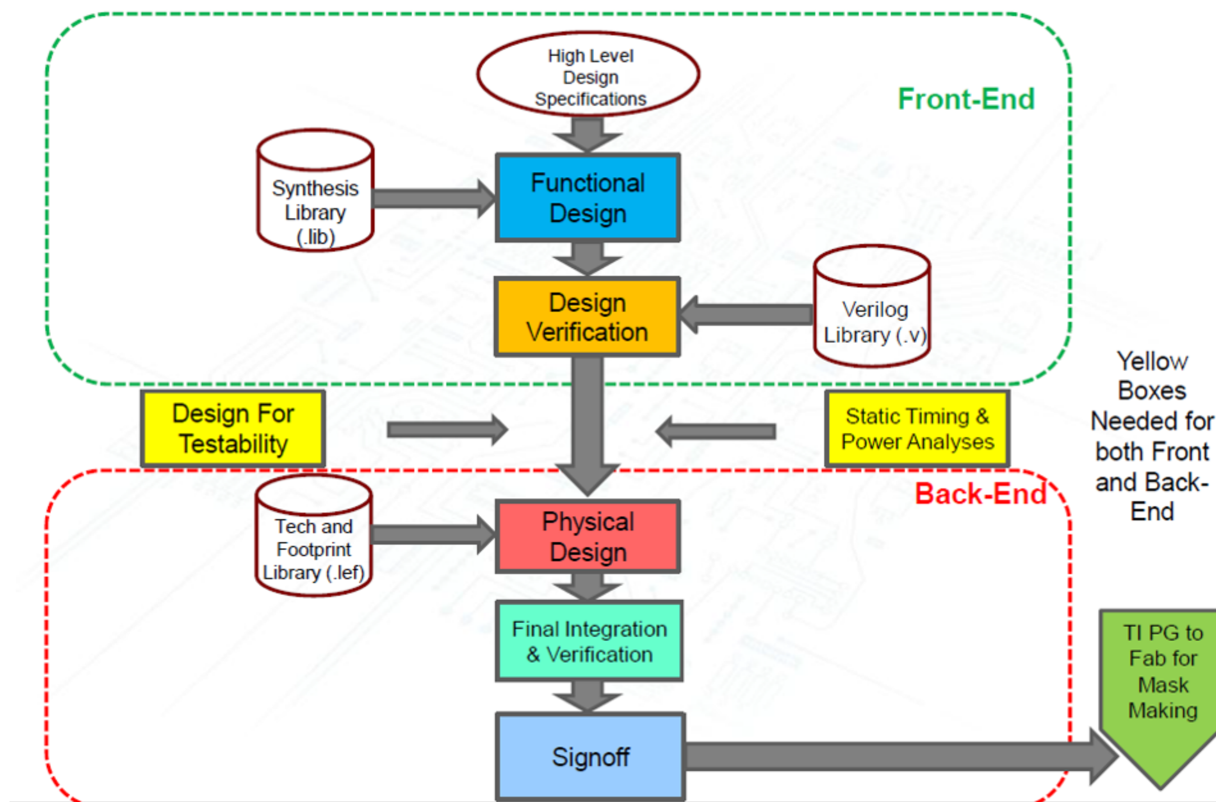


Figure 1.1: IC Design Flow

Digital circuits are designed using the top-down approach as shown in Figure 1.1. The path-breaking team explores and evaluates a product or a product family opportunity. The researchers and the scientists design the product architecture and capture it as an abstract specification document, also called HAS. These specifications are then translated to HDL code by the RTL designers, based on the predetermined micro-architecture. There stands a high probability that a designer misinterprets these specifications or commits a mistake in the implementation, often referred to as a hardware bug. Design verification is a process to detect and fix bugs in the design exhaustively. This is also the least costly method to find/fix the bugs; later down in the IC-design flow, the cost per bug increases exponentially.

The design-RTL is then synthesized into a netlist based on the technology-mapping and available standard cell library for the micro-architecture. The static timing analysis and power analysis is performed to determine the best operating frequency and overall power consumption of the chip. Here on, the transition is made from the front-end to the back-end design phase. Using powerful EDA tools, cell placement and wire routing are performed. Later, the whole design is integrated for the post-layout timing and logic verification. Once the design meets all the sign-off criteria, it is sent to fab for manufacturing. Before shipping the product to customers, post-silicon validation is performed comprehensively.

1.2 Functional Verification

Functional verification, also sometimes referred to as design verification (DV) or logic verification, is a process to demonstrate the functional correctness of a design. Functional verification is a universally acknowledged long pole in hardware design. It is also the costliest investment when it comes to headcount and computing resources. For these reasons, it deserves and often receives the most attention for efficiency improvement measures. The primary goal of functional verification is to ensure that the design implementation matches the specification without the presence of bugs, as shown in Figure 1.2. Thus, it is the responsibility of the verification engineer to flag if the HDL didn't express the correct functionality or if the designer missed considering critical corner-case scenarios.

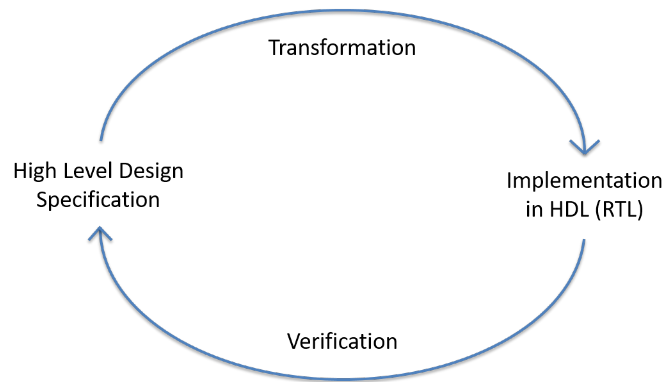


Figure 1.2: Reconvergence Model

Larger functionality integration on a single chip leads to increasing chip complexity. Many of these ICs are used in our daily lives, a few of them are mission/life critical areas such as autonomous cars, airplanes, medical and more. Despite the additional functionality, the customers have the highest quality expectation. Thus design verification proves out to be the most critical stage in any chip design cycle.

There are two main challenges in verifying hundreds of thousands of lines of HDL code.

- Dealing with the enormous design state space
- Detecting incorrect behavior

Typically HDL contains thousands of latches, large arrays (RAM), and combinational logic that control the behavior of a chip. To exhaustively verify that a chip is functionally correct, the verification engineer has to check the enormous state space. For example, to verify a combinatorial circuit with 80 inputs, there are 2^{80} input combinations possible. Thus, verifying the design with each combination is a tedious and time-consuming task. Instead of focusing on each of the possible states of the hardware, verification engineers validate logic at a higher level of abstraction. Traditionally, hardware verification is performed in two major ways to ensure the functional correctness of the design.

1. Simulation-Based Verification
2. Formal Verification

In simulation-based verification, the design is stimulated with a certain set of input vectors to create meaningful scenarios and check the behavior of the design against the given specification. Conceptually, simulating the design for an input vector is verifying a point in the input design space. In a nutshell, simulation-based verification can be seen as verification through input space sampling. Until all the points are sampled, there exists a possibility that a bug escapes verification. Thus, it proves the presence of bugs in the design via simulation and not the absence of it.

The approach in simulation-based verification is to first generate set of input vectors and then check the design behavior with the reference output. This evaluation process is reversed in the formal verification approach. Formal verification is the way to prove or disprove the correctness of a certain specification or property in the design mathematically. Formal verification exhaustively checks for correctness of the property in the design, with no concern of input stimulus. It can be performed in two ways: Equivalence Checking and Model Checking. Although formal verification shows completeness, a problem which simulation-based verification suffers, it uses extensive memory and has long runtimes before reaching a verification decision. Additionally, it is difficult to map complex features of design to formal mathematical property.

A hardware design cycle is managed by balancing the triple constraints- quality, cost, and schedule. Functional verification is the most critical phase, which affects all of these constraints. The end goal of verification is to find the highest quality bugs in the design at the earliest in the development phase and verify the design exhaustively in the least amount of time.

This research is focused on the simulation-based verification approach. Due to its scalability, simulation-based verification is popularly used for functional verification of complex hardware designs with the goal of exercising all critical scenarios in a design and achieving close to 100% coverage targets, required for the design tape-out. As chip complexity continues to grow, simulation-based functional verification is becoming a bottleneck in the overall chip design cycle. In this research, the proposition is to tackle this problem by applying machine learning-guided

stimulus generation that attains verification coverage with a considerable reduction in the number of simulation cycles. In the following chapter, simulation-based verification is discussed further.

2. SIMULATION-BASED VERIFICATION

2.1 Pillars of Verification

Design intent is initially captured at the Register Transfer Level, via a Hardware Description Language like Verilog or VHDL. Simulation-based verification is the most widely accepted technique for verifying RTL designs. It is based on the principle of applying a set of carefully chosen stimulus to exercise a particular area of the design and then checking if the design behaves as expected. The simulation process is continued until all the critical and interesting scenarios in the design space are verified for correct functionality. Here onward, I shall refer to simulation-based verification as verification.

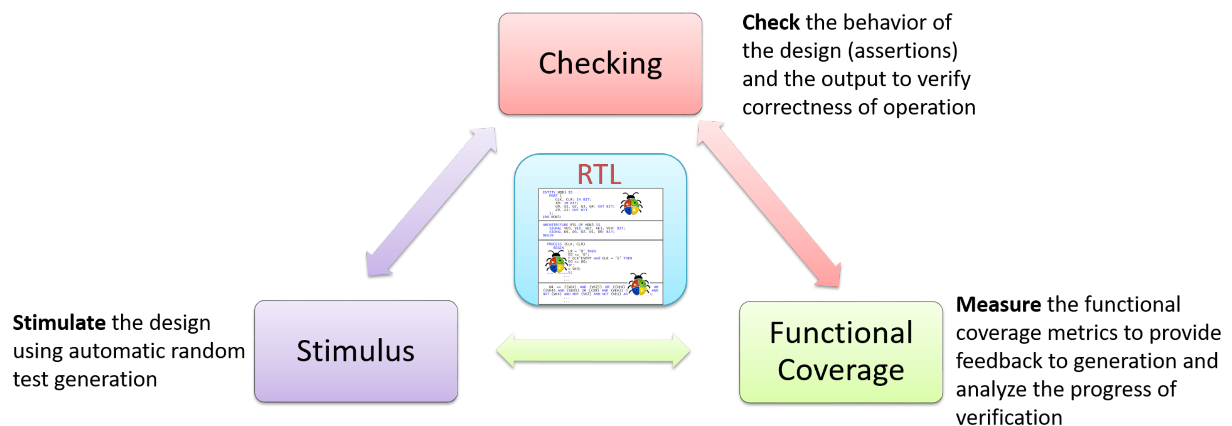


Figure 2.1: Three Pillars of Verification

Stimulus, Checking and Coverage form the three most important pillars of the verification process, as shown in Figure 2.1.

2.1.1 Stimulus

Stimulus is a set of meaningful input vectors applied to the design to verify the implementation (HDL code) against the intent (specification). Stimulus can be categorized into three levels -

transaction, *sequence* and *test* as shown in Figure 2.2. A transaction can be thought of as the fundamental building block of stimulus. For example, a write issued to memory is a transaction with two *attributes* associated with it: address and data. A sequence is a sequential set of transactions that are usually applied to a port of a DUT (Design Under Test). For example, a write followed by a read to the same address will be a sequence of a write and read transactions. Finally, a test is usually a set of multiple sequences applied to different ports of a DUT, and is at the top-most level of the stimulus categories. It is important to note that a DUT can be exercised in multiple fashion with different categories of stimulus. A sequence of transactions may have interdependent effects on the DUT. For example, a read issued after a write to the same address will now return the new data written.

Figure 2.2: Levels of Stimulus

The manual stimulus generation at each level is generally very time-consuming since each of them has to be handcrafted by the verification engineering team. Most of the input patterns to the design are randomly generated along with some constraints. These constraints can be applied to

the stimulus from test-level using test-knobs. In general, a separate test suite is developed for each functionality described in the specification document. On applying a more contrasting stimulus, various DUT functionality will be checked for correctness and heterogeneous design space will be covered.

2.1.2 Coverage

During the design verification effort, coverage metrics are utilized to gauge the progress, assess effectiveness, and help determine when the design is robust enough for tapeout. The coverage results provide the guidance to make critical decisions on the next steps in the verification cycle. In the "coverage-driven verification", a methodology is built around coverage metrics as the primary way to manage verification. Coverage is represented by a set of models, both simple and complex, that capture the design intent. Functional coverage measures the extent to which all the important features and functionality of any design are verified, and thus is of interest in most of the cases. Coverage closure is the point of time at which nearly 100% of the design intent has been verified (or covered).

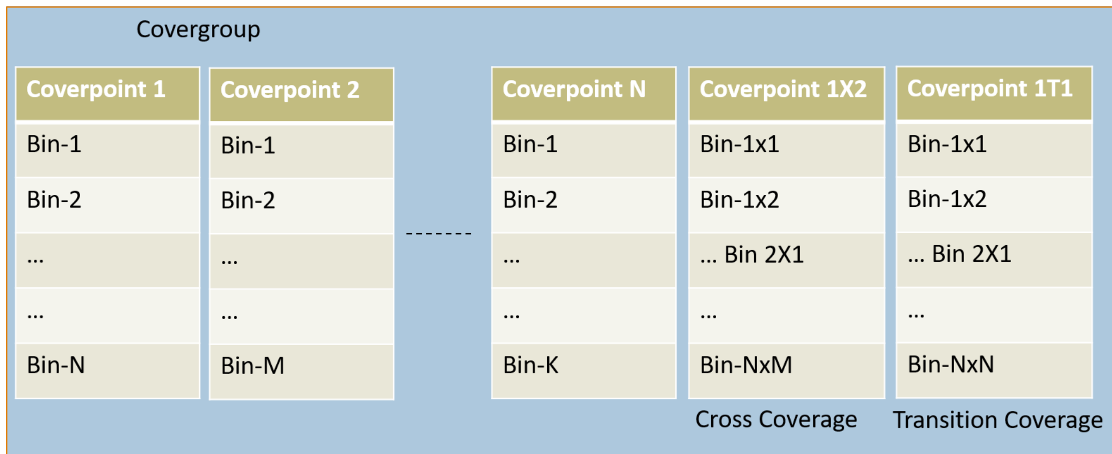


Figure 2.3: System-Verilog Coverage

SystemVerilog is an HDL similar to Verilog but with additional constructs specific to functional verification. One such construct is the implementation of user-defined coverage metrics for

functional coverage. This can be evaluated through a "*cover-point*", also referred to as "*coverage metric*" in this thesis, where each cover-point represents a different feature or signal/variable of the design. All possible legal values of that signal are defined through a set of *bins*. An example is illustrated in Figure 2.3. When any value of a signal/variable is encountered (sampled), that bin is marked as covered. To achieve coverage closure, all the legal bins for all the cover-points must be covered. During volume regression, the coverage bins, which have not been exercised, constitute the coverage holes.

2.1.3 Checking

Checking is a mechanism to monitor the design activity and flag for any erroneous behavior. It is responsible for creating failing conditions, which would guide in finding bugs in the design. An ideal checker would point directly to the bug, making debugging easy. One way to perform checking is to code assertions in the designs. An assertion is an 'if' statement with an error condition that indicates that the condition in the 'if' statement is false. System Verilog provides constructs to write temporal and concurrent assertions. Such checking is done online, while the simulation is running.

Another way for checking is using a scoreboard or reference model. Scoreboard/Reference model is a golden model that maintains the correct functionality of one or more design features from the specification. The scoreboard is provided the same set of input vectors as the DUT; the scoreboard prediction and the DUT output are compared for any inconsistency. Here, the checking is done offline, at the end of the simulation.

2.1.4 Simple Cache and Memory Example

Here, I shall elaborate more on stimulus and coverage metrics with an example of a simple cache and memory design as shown in Figure 2.4. The total memory capacity is 32 bytes with each word size and address size equal to 8 bits and 5 bits respectively. There is a direct-mapped cache for this memory with capacity 4 bytes. Each cache line is mapped to 8 address spaces in memory. There are two basic operations that can be performed on each address, i.e., *read and*

write.

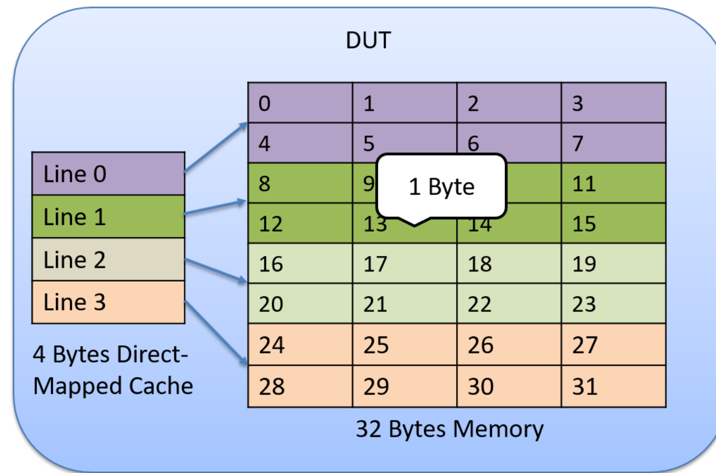


Figure 2.4: Simple Memory and Direct Mapped Cache Design

For simulation-based verification of this design, we provide varied stimulus to the design and define coverage metrics for exhaustive verification effort. Typical transaction attributes would be address, data, and request_type (read or write). A sequence, which is a group of transactions, could be a read sequence-all read transactions, write sequence-all write transactions or random sequence - all transaction attributes take random values. At top level, a test can initiate multiple sequences and thus can be a fully-directed test (pre-decided transactions), random test (all random transactions) or somewhere in between. Various coverage metrics, to gauge the verification process, could be, read on each address, write on each address, cache hit/miss on each address, write followed by a read on each address, and many others.

2.2 Verification Cycle

The verification cycle is divided into two broad stages: directed and volume verification. In the first stage, exact values of the stimulus are known before applying them to the design. This helps in detecting expected bugs and to ensure that the basic functionality of the design is correct. Volume verification, on the other hand, exercises a large combination of random and constrained-random

stimulus to expose corner-case bugs that are harder to detect. The majority of the verification cycle time is spent on volume verification, owing to the huge amount of stimulus getting applied. Coverage is a measure of the success of the volume verification phase. Robustness of functional verification is associated with attainment of near 100% coverage and is one of the critical metrics used in the final tape-out decision.

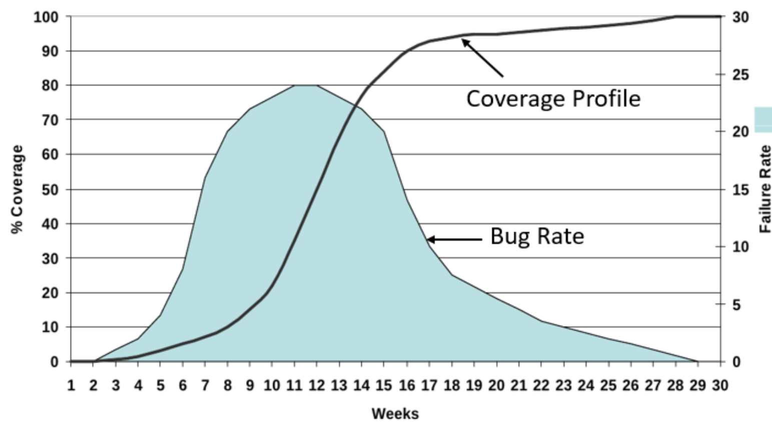


Figure 2.5: Coverage and Bug-rate in the Verification Cycle

Over the verification cycle, coverage typically rises with time up to a point of saturation, where it approaches 100%. On the other hand, the bug rate (number of bugs identified per week) sees a sharp increase initially, and then decreases and eventually approaches close to zero. When the coverage converges to 100% and bug rate drops down to 0, the design becomes ready for tape-out. Figure 2.5 depicts a general trend in coverage closure and bug rate over time. Efforts at improving efficiency in functional verification are aimed at compressing the curve towards the left, thus leading to faster coverage closure, while also detecting bugs sooner. It is worth noting here that coverage aims at verifying every imaginable scenario in the design, which consequently has the potential to uncover more bugs.

2.3 Verification Environment and UVM

A verification environment, also called the testbench, is created around the three pillars of verification. It is the framework to verify the functional correctness of DUT by generating and driving stimulus to design and compare the design output with the expected golden output. A lot of the engineering effort is invested in setting up a verification environment for simulation-based verification; testbench can account for up to 80% of the overall lines of code written in the design process. A typical verification environment is created by assembling several components, each designed to perform a certain operation. Most of the verification testbench are coded in System-Verilog or Specman-e language, as they provide essential constructs to code testbench components.

As creating a verification environment is a time-consuming task, verification reuse improves the productivity and efficiency of the verification process and thus it is highly desired. UVM is a standardized methodology, which is created using System-Verilog to build modular and reusable verification components and testbench. Figure 2.6 displays various verification components in UVM and their interconnections.

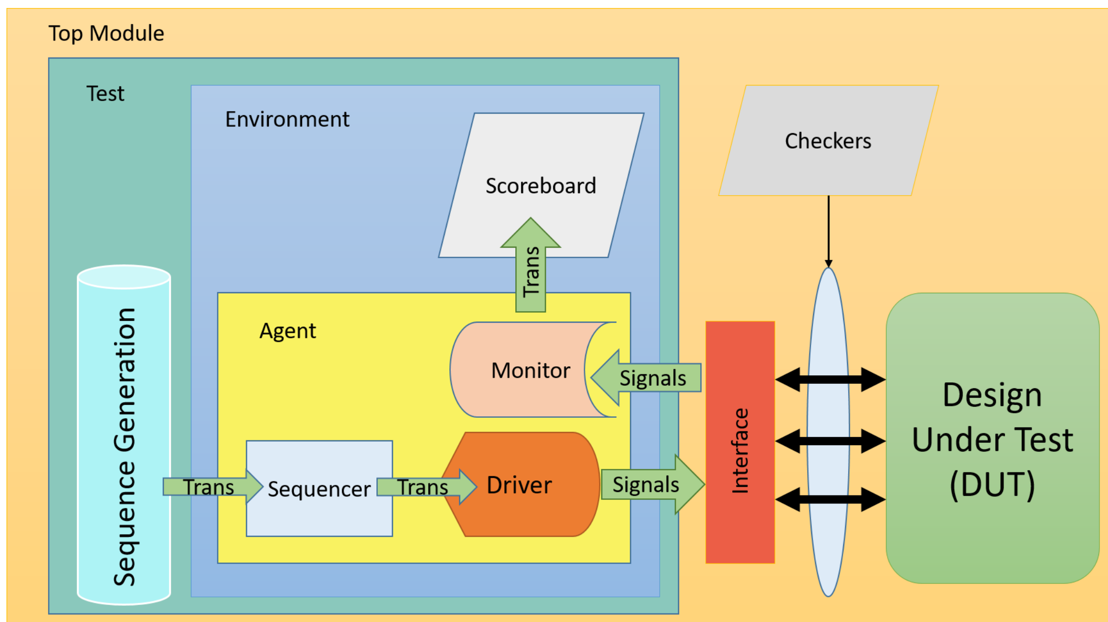


Figure 2.6: Typical UVM Testbench

2.3.1 UVM Components

- **Transactions and Sequences** - A Transaction is a primitive level unit that contains the information to simulate the DUT. It contains the data-members and constraints on them. A Sequence is a collection of transactions. It is responsible for constraint random generation of the transactions by adjusting constraints on its data-members as per the test requirement.

- **Sequencer** - It controls the flow of request and response sequence items between sequences and the driver.

- **Driver** - It requests and receives the stimulus in form of transactions via the sequencer and drivers it to DUT by converting the transaction level information into the pin level through the interface.

- **Monitor** - It observes pin level activity on interface signals and converts into a packet, which is sent to the components such as a scoreboard.

- **Scoreboard** - It receives data items from monitors and compares with expected values. Expected values can be either golden reference values or generated from the reference model.

- **Agent** - It is a container that groups Sequencer, Driver, and Monitor, all specific to an interface or protocol.

- **Environment** - It is a container for grouping higher level components like agents and scoreboard.

- **Test** - It is a program responsible for testbench configuration, testbench component construction and initiating stimulus driving.

- **Interface** - It encapsulates a bundle of wires to enable the communication between stimulus driver in testbench and the DUT. Also, checkers pertaining to interface protocols are coded in it.

- **Top Module** - It connects the DUT and the Testbench via the Interface.

2.3.2 Advantages of using UVM-based Testbench

The UVM architecture has been designed to promote modularity and resuablity of various verification components across different environments. Low-level components as a driver, a monitor or an agent as a whole can be reused for the verification of multiple designs. Also, the whole verification environment can be reused by multiple tests and configured top-down by those tests. UVM verification components can be configured in a very flexible way without modification to their source code. Additionally, UVM provides a straightforward mechanism to replace any component with a new one without making changes in the testbench. Thus, UVM based testbench are widely used in industry because of the these features - Modularity, Scalability, Reusability & Configurability.

2.4 Verification Challenges

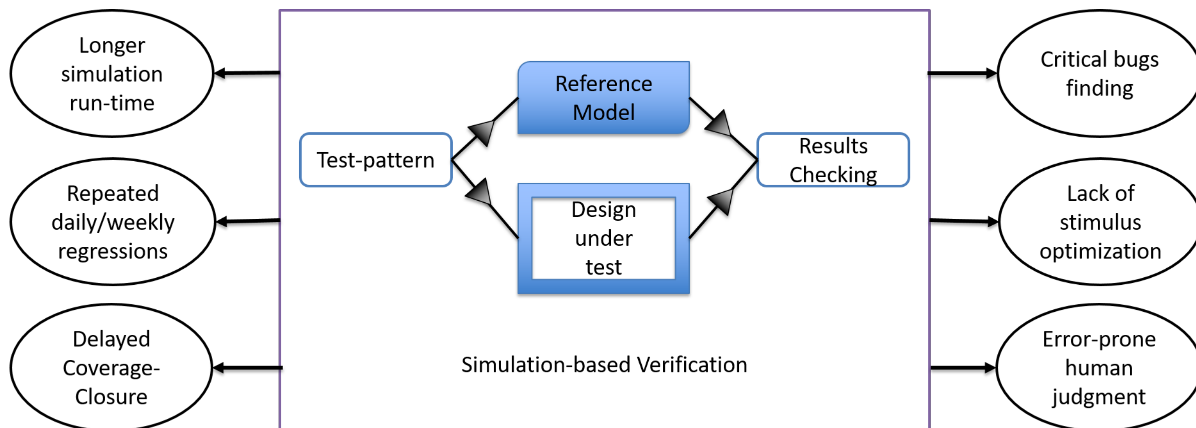


Figure 2.7: Challenges Faced in Simulation Based Verification

Simulation-based verification is most commonly practiced in the industry for verifying complex digital designs. The most critical challenge in verifying such large designs is the longer simulation time. This is complimented by lack of stimulus optimization to remove redundancy in random test vector generation. To counter it, large volume regressions are kicked-off periodically

for the coverage closure. In the whole verification flow, a lot of human decision-making is involved that can be biased, error-prone and cause delays in the project cycle.

3.1 Machine Learning Terminologies

- **Dataset** : Data is the most essential part of Machine Learning. Any Machine Learning system would require the user to either get the data (e.g. from some public resource) or collect it on its own. All the data that is used for either building or testing the ML model is called a dataset. Basically, datasets are divided into three separate groups:

  Training data: Training data is used to train a model. ML model learns to detect patterns from the data and determine which features are most important during prediction.

  Validation data: Validation data is used for tuning model parameters and comparing different models in order to determine the best ones. The validation data should be different from the training data, and should not be used in the training phase. Otherwise, the model would overfit, and make poor predictions for the new (unseen) data.

  Test data : Test data is completely unseen data, used once the final model is trained to simulate the model's behaviour.

- **Input Attribute** : Features extracted from training dataset and used for output prediction

- **Target Label** : Output values/labels to be predicted

- **ML Algorithm** : Program that provides a model for prediction suitable for the training dataset

- **ML Model** : The artifact created by applying the algorithm on training dataset

3.2 Machine Learning Algorithms

Machine Learning, a sub-field of Artificial Intelligence, aims to provide computers or computational machines the ability to learn without being explicitly programmed. The algorithm learns the patterns from the presented problem and makes a smart prediction by constructing a decision

model. There are three broad categories of ML algorithms: Supervised Learning, Unsupervised Learning and Reinforcement Learning algorithms as shown in Figure 3.1. Machine Learning is an advanced step towards exploiting the computational capabilities of computers and can be applied in various applications, especially in the domain where data is available in abundance.

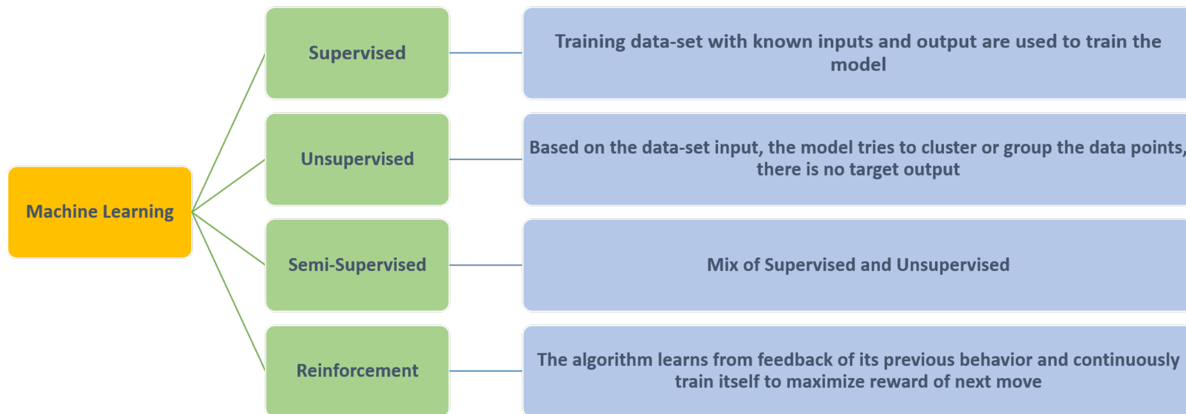


Figure 3.1: Flavors of Machine Learning Algorithms

In Supervised Learning, the model is trained with labeled training dataset to predict the output of unseen example. Here, the learning phase includes the training on known inputs and corresponding correct output or class. It then tries to deduce a function to map inputs to outputs by iteratively making a prediction on the training dataset and correcting it. Once trained, the learning system is then expected to predict the output for unseen examples following the pattern learned on the training set. To sum up, for input variables(x) and the output variable(Y), a Supervised Machine Learning model will create a robust mapping for a new input data(x) such that it can predict the output variable(Y) with high accuracy, i.e., $Y = f(x)$.

Supervised learning problems can be further grouped into classification and regression problems.

- **Classification**: A classification problem is when the output variable is a category, such as "dog", "cat" and "horse"

- **Regression**: A regression problem is when the output variable is a real value, such as "price" or "weight".

In Unsupervised Learning, the system is only given a set of an unlabeled dataset, i.e., data without a class or predicted output assigned to them. The goal is to create a model to structure or distribute the data for better inference. It is called unsupervised learning as there is no correct answer or feedback. The only guidance for learning here is the structure of the received input. A simple example is the clustering problem where data points are grouped based on their inherent behavior. Semi-supervised learning is a mix of both Supervised and Unsupervised learning and applied when from large input data(x) only some of the data is labeled(Y).

There is a third genre of machine learning approach - Reinforcement Learning, where the guidance to learning is provided by reward-based feedback. Here, an agent performs actions in an environment and the agent receives various rewards depending on what state it is in when it performs the action. The overall aim is to predict the best next action to earn the biggest final cumulative reward. Hence, the system is guided towards the correct decision model to be learned without training with an explicit dataset.

Growth of new computing technologies has completely changed the outlook of machine learning application today. We have more sophisticated data preparation capabilities and advance machine learning algorithms which aid in developing scalable smart automation techniques.

3.3 Supervised Machine Learning Algorithms

3.3.1 Deep Neural Network

Deep Neural Network (DNN) is a machine learning technique that is very much inspired by structure and working of our brain. The basic fundamental unit of a neural network is the neuron.

As shown in Figure 3.2, each neuron is fed with a set of inputs, each multiplied with an associated weight. The neuron calculates a function on these weighted inputs. Based on the type of regression: linear or logistic, it calculates the final output value. A sigmoid function returns a value between 0 and 1.

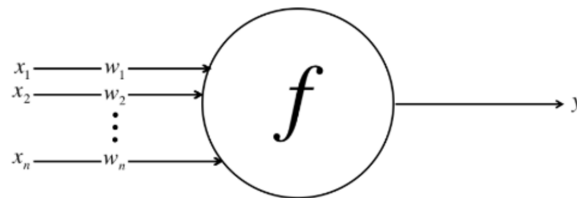


Figure 3.2: Schematic of a Neuron

A node layer is a row of such neurons. Figure 3.3 [1] shows an example of a neural network where each neuron is well connected with neurons in the higher layer. The input layer receives the data and outputs to next hidden layer. Thus, each layer's output becomes the input for the subsequent layer. All the layers between input and output layers are call hidden layers. The number of hidden layers is also the depth of the neural network and if there are more than 3 hidden layers, it is called a deep neural network.

Using the labeled input and output data, a model is trained where essentially, the weights of the neurons are re-adjusted iteratively to create a dependable mapping from input values to the output label(s).

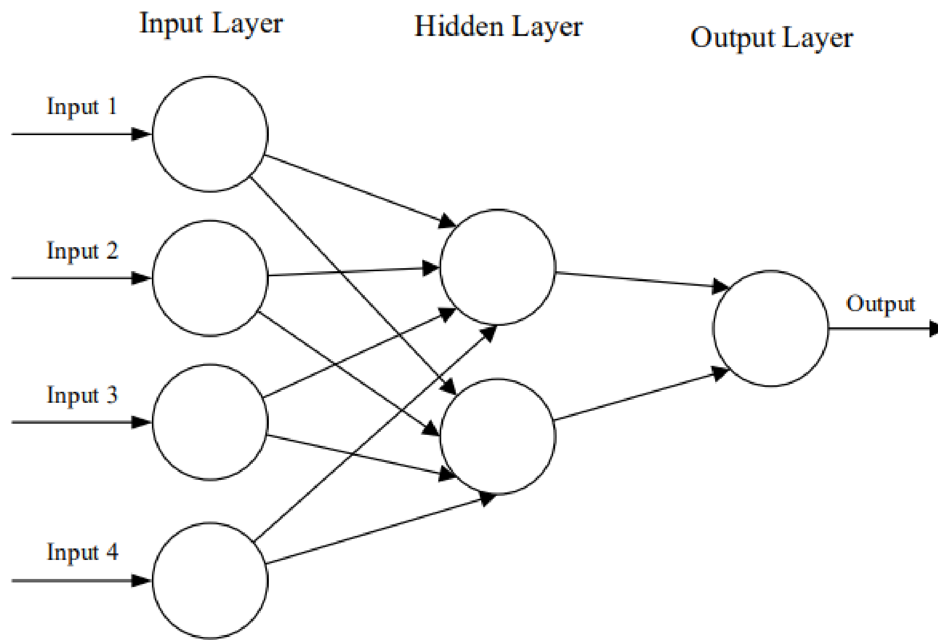


Figure 3.3: Deep Neural Network

3.3.2 Random Forest

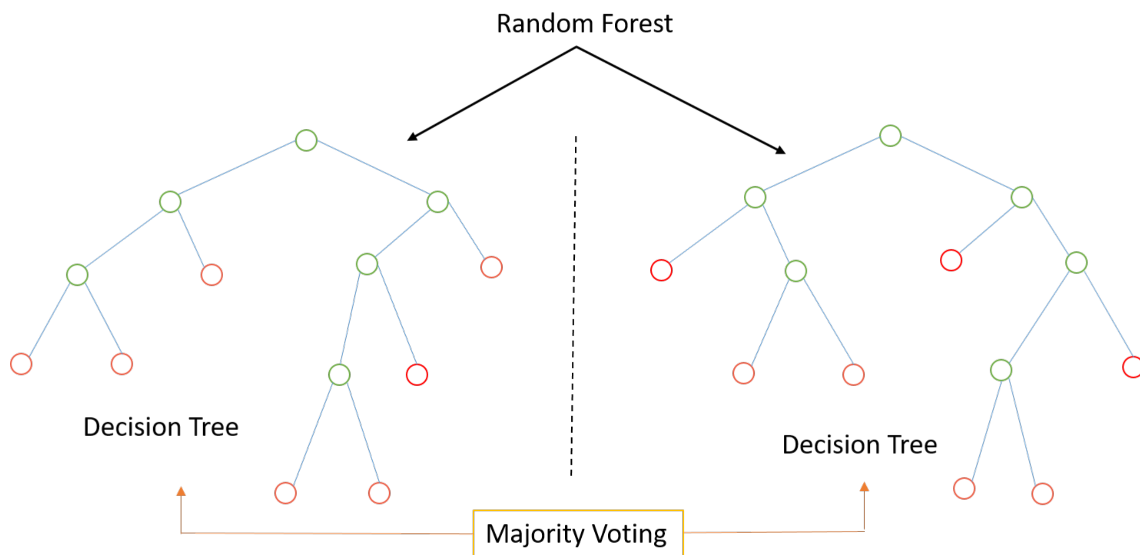


Figure 3.4: Random Forest

The basic building block of Random Forest is a decision tree [2]. The training algorithm seeks the best feature-value pair from the input training data set to create nodes and branches. The value is the decision outcome at each node in the tree. After each split, this task is performed recursively until the maximum depth of the tree is reached or an optimal tree is found. At every branch or node, a conditional statement based on a fixed threshold in a specific variable classifies the data point, therefore splitting the data. For prediction, a new input data-point starts in the root node (top of the tree) and moves along the branches until it reaches a leaf node (decision) and no further branching is possible.

A Random Forest consists of multiple instances of the decision trees. Each decision tree in the forest considers a random subset of features when forming decisions and accesses only a random set of the training data points. This increases the diversity in the forest and thus overall predictions are more robust. The final prediction of the random forest is done by either taking average or majority votes of all individual decision trees.

3.3.3 Recurrent Neural Network

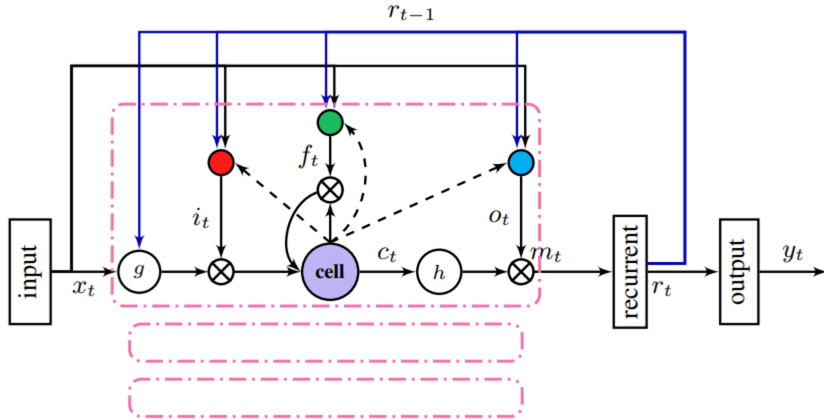


Figure 3.5: LSTM Unit

RNN (Recurrent Neural Network) is a deep learning model, widely used for training time series data and voice recognition. Unlike feedforward networks like DNN, RNNs have an internal state

memory element for processing sequence of inputs. LSTM (Long Short Term Memory) Network is an RNN architecture and its building block LSTM unit shown in Figure 3.5 [3].

The LSTM unit's repeating module has a very simple structure, such as a single tanh layer. Instead of having a single neural network layer, LSTM has four interacting nodes in a very special way. LSTM regulates the addition or removal of information to its state with a structure called gates. Gates are designed to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation. The gate output value controls the flow of data through each component; a value zero means "let through nothing" while a value one would mean "let through everything". There are three such gates in the LSTM unit: an input gate, an output gate and a forget gate.

4. PREVIOUS RELATED WORK

Recent years have seen an uptick of effort in the exploration of ML in functional verification to varying degrees. Stan Sokorac [4] proposed a methodology where toggle coverage metrics and ML algorithms are utilized to create tests with higher potential of exposing previously uncovered regions in the design. To maximize the probability of capturing non-trivial bugs, toggle pair coverage metric is considered to form clusters of the test group. Unfortunately, toggle coverage is not a very effective metric to capture design functionality. In another work [5], clustering and neural network techniques are implemented to reduce the time taken to collect and evaluate coverage. It recommends collecting the coverage only on a subset of structural coverage metric and predicts coverage for the rest of the design. However, this work does not take stimulus optimization into consideration and also isn't focused on reducing the number of simulation cycles for coverage closure.

Coverage-Directed Test Generation (CDTG) is a popular SBV methodology designed to close the loop between coverage analysis and test generation with the help of ML or other techniques. A review of various CDTG techniques is provided in [6]. This closure is achieved by learning from the simulated tests and the achieved coverage, the cause and effect relationships between tests, or test generation constraints, and the resulting coverage. These relationships are then utilized to construct new tests or to create new constraints for a test generator in such a way that coverage closure is achieved faster and more reliably.

The authors in [7, 8] proposed stimulus optimization through test pruning using support vector analysis (SVA). In [7], a graph-based kernal function *k()* is defined that measures the similarity between a pair of tests. Only selective tests are simulated based on relative similarity measures and clustering. Contrarily, [8] recommends using coverage-based kernel function to estimate the coverage of un-simulated tests. Only tests hitting the uncovered area are then simulated. This idea is very similar to the test-level optimization discussed in *section* 6. Both these researches were carried out on a confined logic of a commercial processor.

A Bayesian network based CDTG model is proposed by Shah and Avi [9]. Their methodology provides directives to test generation mechanism for expeditious depth and breath design coverage. A neural network-based CDTG method is introduced in [10]. Here an ANN (Artificial Neural Network) module is introduced into the CDTG loop with the goal of increasing the probability of selecting high coverage test cases and assertions coverage is the main metric of coverage.

All the previous work is focused on test level optimization. Our work, on the other hand, proposes transaction level optimization, which is at a lower abstraction level and hence more fine-grained. The experiments carried out and results obtained as part of this research presents such finer optimization proves more effective in many scenarios.

5.1 The DUT - Multicore Cache Design

The research is targeted at developing smart stimulus optimization mechanism in the functional verification environment with the DUT comprising of L1 and L2 cache systems. The design contains 4 processor stubs for each L1 cache, a bus system to carry out transactions among L1 caches and also between L1 & L2, an arbiter to determine bus access, and a stub for main memory. DUT implements functional aspects of L1 cache coherency protocols. The basic block representation of the system is as shown in Figure 5.1. The design is very modular with the option available to set parameters for the number of cores, address width, data width, cache line size, and cache size.

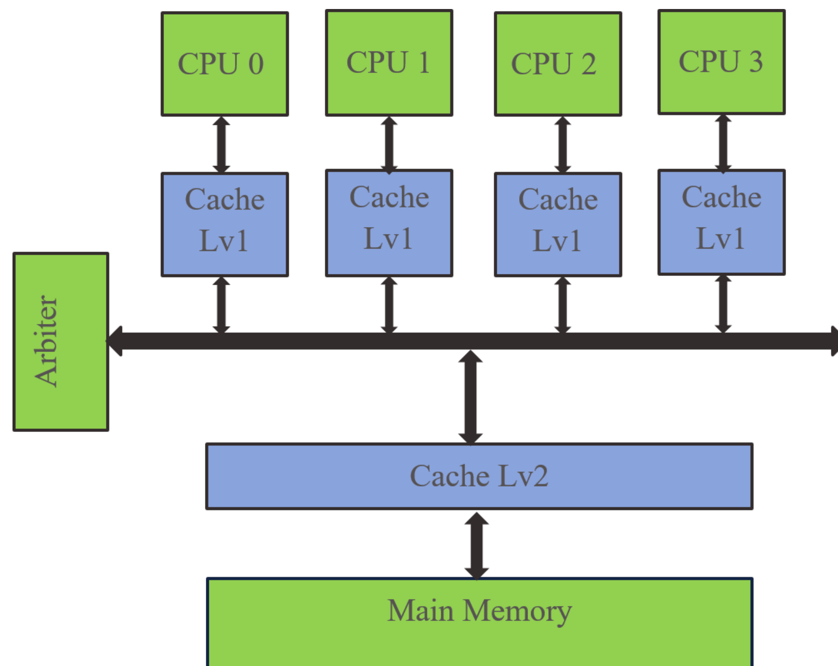


Figure 5.1: Multicore MESI Based Cache Design

The key design specifications of the DUT are the following:

1. 32 bit 4 core processor system

2. L1 cache for each processor with Shared L2 memory

3. Communication between L1 & L2 and among L1s happens through system bus and the grant of the bus is decided by an Arbiter

4. 4-way associativity in L1 and 8-way associativity in L2

5. MESI based coherency protocol in L1

6. Pseudo LRU replacement policy

7. Data and instruction caches are separated in L1, L2 is a unified cache

5.2 UVM Testbench

As mentioned in *section* 2.3, UVM is a commonly used methodology to develop a simulation-based verification environment for any design. For this research, we modified the currently existing UVM based testbench for the DUT mentioned in the previous section. Figure 5.2 represents the overall structure of this testbench.

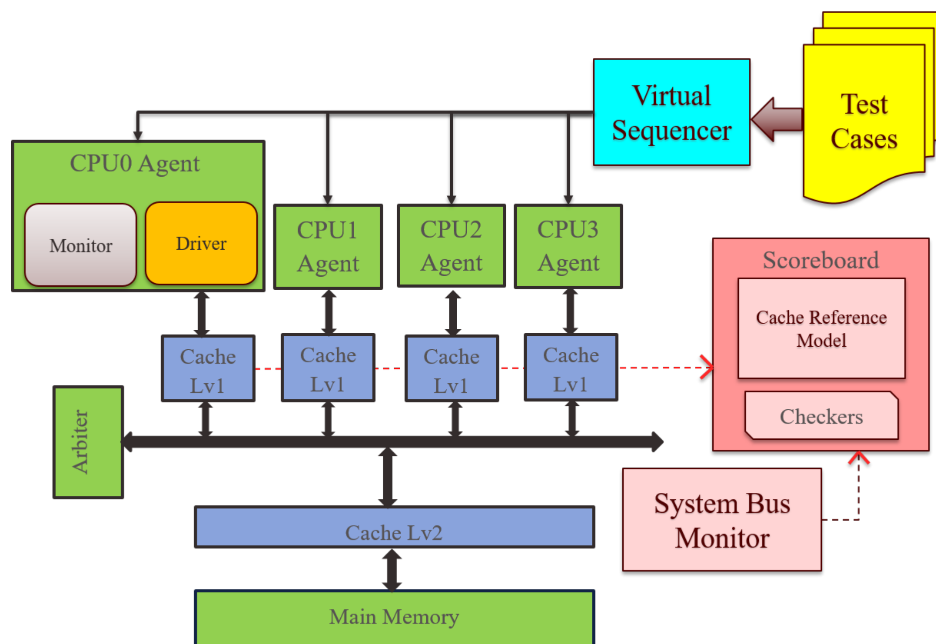


Figure 5.2: UVM Verification Testbench

The UVM components in the testbench include CPU agent, virtual sequencer, test cases, scoreboard, and the system bus monitor. The ***CPU agent*** mimics the role of a CPU core and communicates with L1 cache in the design. The ***Driver*** receives the transaction from the sequencer and drives packet information to DUT using the interface signals. The ***Sequencer*** receives transactions from the testcases (stimulus generator) via virtual sequencer and forwards it to driver sequentially. The ***Monitor*** passively observes the activity on the CPU-L1 interface, packages the information into a monitor packet and is sent to the scoreboard for high-level checks. Coverage collection is also realized in this component. The ***System Bus Monitor*** is another passive device that records the activity on system bus interface and bundles it in SBUS packet. It is also sent to the scoreboard for checking. Coverage is logged in this module as well.

The ***Scoreboard*** is primarily responsible for checking the DUT functionality by ensuring the correctness in the DUT output behavior. It compares the incoming/outgoing transaction values with the golden expected values and flags any mismatch. It contains a cache reference model and high-level checkers. The baseline here is the reference mimics the correct functionality of the DUT. It is coded using the associative array structures for an efficient implementation in SystemVerilog. The ***Virtual Sequencer*** enables us to have fine-grained temporal control of the transactions on each of the CPU agents. We can send transactions in parallel to each of the CPU agents. It receives a sequence from the test class and forwards transactions to the agents as specified in the virtual sequence.

The DUT in the testbench environment is bug-free, i.e., functionally correct.

5.3 Test-Suite

The present test suite contains tests to verify the cache system at the microarchitecture level. The microarchitecture verification refers to check the implementation of the processor/SOC. This includes the verification of subsystems for pipelining, in-order or out of order execution, branch prediction, caches and coherency, system bus and many more. The tests comprise of various sequences of transactions as explained in *section* 2.1.1.

On contrary, the verification at processor/SOC level is done using tests that are assembly pro-

grams or traces of huge programs comprising of various instructions. These tests are coded to check the architecture correctness and system performance analysis. A simple example of architectural functionality is the instruction set architecture. In this study, as we are considering a sub-system of the processor, only micro-architecture level verification is considered.

The existing test suite consists of a total of 23 random and directed tests targeted to cover various design functionalities. Several coverage metrics are defined in the testbench pertaining to the design features and specification. Beyond 99% coverage is obtained when running all the tests with multiple random seeds as a part of dynamic regression.

A fully random test is the most simple and trivial test to code that requires very little knowledge about the design functionality. As the transactions are generated in the most random fashion, there is a higher chance of repetitive patterns in the stimulus. Therefore, random regressions suffer from poor performance in terms of covering newer design space after a few tests simulation. This issue is complemented by adding constraints in the test template to create directed-random and directed tests to target the uncovered design scenarios. This reverse engineering exercise requires extensive knowledge of the design where a hand-coded test template is simulated to maneuver the design to explore new space. Also, generating a specific sequence of transactions for coverage closure is the last resort and least preferred in the whole verification process.

Alternatively, here I explore the use of machine learning in the grand scheme of functional verification where only selective tests and transactions are simulated from the random stimulus generation templates. This requires very little modification to the existing verification environment; a Python-based machine learning model is created for training and prediction surrounding it.

6. TEST-LEVEL STIMULUS OPTIMIZATION

This research is mainly undertaken with an outlook of exploring different avenues where Machine Learning can be applied for stimulus optimization in the functional verification. The first-half of this work focuses on coarse-grained test-level stimulus optimization technique. The goal here was to predict the performance of a test in terms of its coverage in the design without simulating the test.

6.1 Methodology Overview

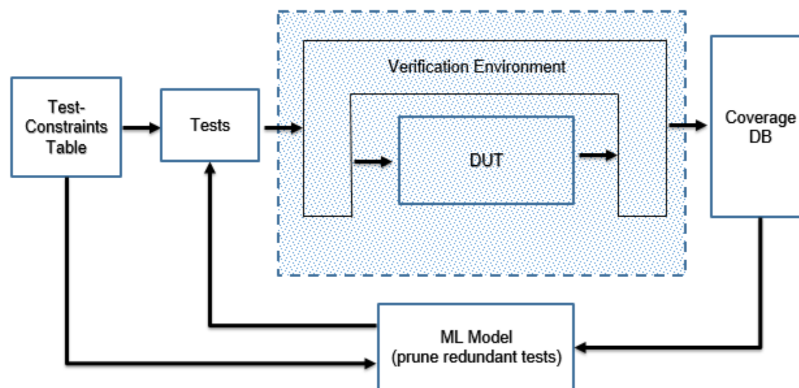


Figure 6.1: Test-level Stimulus Optimization Methodology

Test-level stimulus can be controlled through knobs (or test constraints), which can help steer the stimulus towards a particular design functionality. Here, I introduce test-level optimization to predict functional coverage based on these knob values. This technique is applied to prune redundant tests and direct the stimulus towards uncovered areas of the design. A machine learning-guided test level optimization flow is illustrated in Figure 6.1. The flow begins with the simulation of random tests, where the test knobs are fully randomized. The simulation data in the form of coverage database and log files serve as training data for the Machine Learning (ML) model. The input features to the ML model are extracted from the test knobs and the prediction output specifies

whether certain bins are covered by these test knob settings. Once the ML model is sufficiently trained, it is used to predict the coverage for a set of random input knob values. The decision for pruning is then made accordingly. The model is re-trained in batches with simulation data from the newly simulated tests. This is done until coverage closure is attained. These steps are further described in the following sections.

6.2 Test Knobs

A fully random test performs random sequence generation where transaction attributes are assigned completely random values by random number generation mechanism in the environment. These values are not purely random but based on pseudo-random number generation algorithm, which takes in a parameter (the initial value) called a seed value. Most of the times, based on the design specification, the transaction attributes can only be assigned a value within a specific range and cannot take a complete random number. To avoid the illegal transaction generation, constraints are applied on these attributes that are defined as random variables in the code. The constraint solver starts with a random seed and creates a test that satisfies the constraints. Then it takes another seed and does the same again for another test. The testbench provides us the freedom to specify the stimulus generation constraints at the transaction, sequence, and test-level with slight modifications in the environment.

The CPU-L1 transaction packet has 4 attributes- request_type, access_cache_type, data and address. The ***request_type*** attribute specifies whether the incoming transaction is of type read or write whereas ***access_cache_type*** describes the target I-Cache or D-Cache. Attributes ***address*** and ***data*** would contain the target address and the data of the transaction, which are constraint within a specified range in the specification. A sample system-verilog constraint is shown below.

Specification : Any address less than 32'h4000_000 is for instruction cache and the rest is for the data cache.

```
constraint c_address_type { address['TAG_MSB_LV1:(‘TAG_MSB_LV1-1)] == 2'b0 ->
access_cache_type == ICACHE_ACC; address[‘TAG_MSB_LV1:
(‘TAG_MSB_LV1-1)] != 2'b0 -> access_cache_type == DCACHE_ACC; }
```

Test Parameter	No. of bits	Description
No. of Transactions	1	Controls number of transactions in a sequence
Fix Address	1	1-all transaction has same target address, 0-otherwise
Is Parallel/Sequential	1	1-all sequences are executed in parallel, 0- in sequential
Seed	1	Seed for randomization, between 50-100
Core Selection	4	1-execute a seqeuence, 0-otherwise; 1 bit per core
I-Cache or D-Cache	8	10- icache, 01-dcache; 2 bits per core
Read or Write	8	10- read, 01-write; 2 bits per core
Total	24	

Table 6.1: Test Knobs

A sequence is a collection of transactions, thus all the transaction attribute constraints can be overwritten at the sequence-level. Another parameter available at sequence level is ***no. of transactions*** in a given sequence. Using a virtual sequence in a test, multiple sequences can be kicked off on separate CPU agents in parallel (using fork..join construct) or sequentially. Thus, there are a total of 24 test parameters, also called test knobs, as shown in Table 6.1, to control the random behaviour of a given test. These parameters are passed as switches in the command-line while kicking off the simulations. More details on how to simulate the parametric test can found in *Appendix A.1*

6.3 Coverage Metrics

Coverage is the most important metric to measure the quality of stimulus and progress of the verification cycle. Coverage-driven verification is performed where the overall coverage in the design is the measure to sign-off for the verification phase. The existing testbench had certain coverage metrics defined based on the specification and features of the design. Coverage metrics are user-defined and are based on the human judgment of the extent to design is verified to prove the absence of bugs in the design. Few more metrics were added to make the verification effort more rigorous and prove that stimulus optimization technique helps to reach the target goal in few simulation cycles.

The coverage metrics, usually called coverpoints, are based out of the design specification.

Coverage Metric	No. of Bins	Description
address_X_req_type	448	address cross request type
data_bin_cov_out	256	data value range is divided in 256 bins
fsm_cov_out	76	19 per core (proc_cur+snoop_cur +proc_tran+snoop_tran)
proc_X_req_cov_out	16	processor cross request type
proc_X_snoop_cov_out	16	processor cross snoop state
snoop_request_bin_cov_out	15	snoop request state
design_mesi_fsm_transition	143	transition of design mesi state
cache_hit	768	cache hits on each address for each processor
Total	1738	

Table 6.2: Coverage Metrics

Each feature/event is captured in one or more coverpoints. Cross and transition coverages are defined to increase the verification effort. Transition coverages are very useful for covering DUT state machines. Cross coverage is the cross product of two or more previously declared coverpoints, i.e., overlap of two or more events, thus defining rare and critical corner scenarios in the design. Table 6.2 shows all the coverage metrics used to attain the coverage closure in the context of this research.

Coverage collection for each simulated test and further data processing is explained in *Appendix A.2*

6.4 Machine Learning Model Training

For test level optimization, a random test has 24 test knobs (integers) and these are translated to 24 input features to the ML model. Using Python based randomization script, a database for random testcases is created and stored in form of a table. Each entry in this table has 24 values, each corresponding to a test knob, and can be translated for test simulation. For the coverage prediction, eight coverage metrics in the design are considered with a total of 1738 bins. Each metric has an individual ML-model for prediction, as the coverage for any two different metrics is mutually exclusive. The initial ML model training was performed with the data-set collected from the first 50 randomly simulated tests, achieving the training error $< 10\%$.

The previous works on test-level stimulus optimization are mostly based on SVM (Support Vector Machine) and neural network. In this work, random forest (RF) [11] is also investigated, which is a decision tree-based machine learning approach and shows pretty good results. Two types of ML classification errors are considered to avoid the over-fitting on training data-set, which are validation set and 5-fold cross validation. A lower validation-set error conveys model perform well for data-set not used in training, while 5-fold CV will make sure the model is not biased towards the training data-set. The training to validation dataset split in input dataset is 80-20. On simulating more new tests, the input and coverage data is appended to the existing dataset and the model is re-trained periodically. Figure 6.2 represent the model training and validation process.

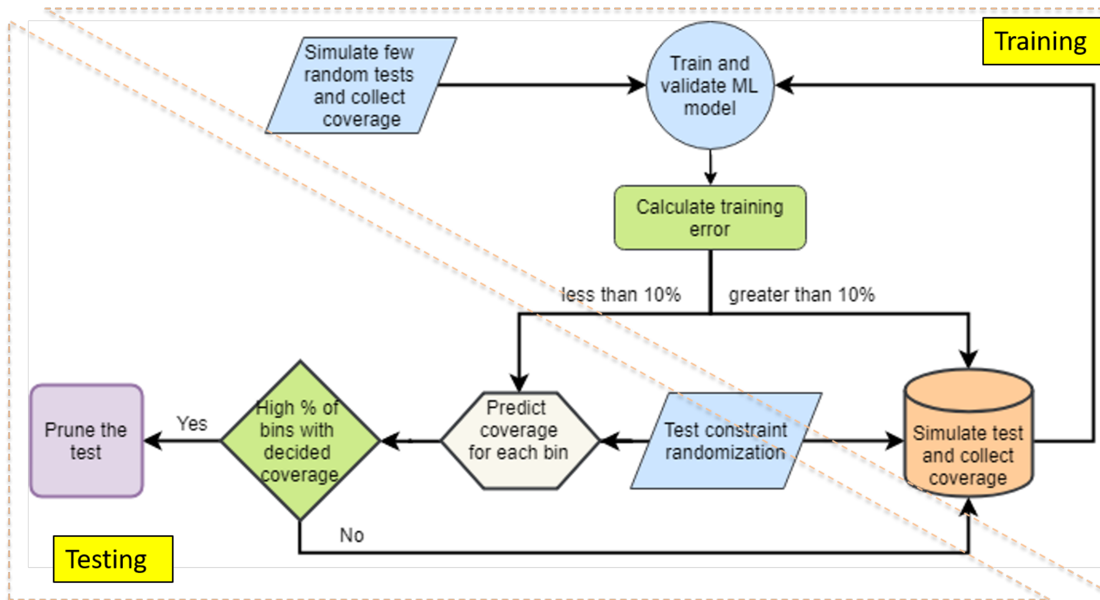


Figure 6.2: Test-level Stimulus Optimization Flow-diagram

6.5 Coverage Prediction and Test Pruning

Once the model is well-trained, i.e., the validation error is less than 10%, the next step is to predict the coverage on test parameters of new incoming test. One noteworthy improvement in this work is the ternary classification on coverage of a bin instead of binary classification in previous

works. I take the classification output of the ML model as the probability of a bin being covered. If the probability is greater (smaller) than a higher (lower) threshold $\alpha \in (0, 1)$ ($\beta \in (0, \alpha)$), the corresponding test is classified as (not) covering the bin. Otherwise, the probability is between α and β, and the coverage is undecided. If a test leads to a high percentage of bins with decided coverage, either "yes" or "no", it is (not) performed for simulation when it would (not) cover bins that have not been covered yet. This is depicted in Figure 6.2.

When the value of α is set close to 1 (pessimistic) with moderate training data, a high percentage coverage predictions were undecided. The model requires intensive training and more training data for the coverage prediction to lie in the interval $(\alpha, 1)$. Both these propositions hurt our approach as prior will lead to model overfitting and later requires more simulation, which defeats the overall purpose. Similarly setting the value of α further to 1 (optimistic) will lead to prediction in decided bin, i.e., "yes" even though there is a higher chance that the test may actually not hit that coverage bin. To determine a balanced value for α, a set of training experiments were performed with varying value of alpha and the training error is determined using a validation set with 80-20 (training and validation) input data breakage. Similar experiments were performed to determine a fair value of β. Going further, α is set to 0.9 and β is set to 0.1.

If the ML prediction on a test can decide the coverage for over 90% of bins, then this test is simulated or pruned out depending on its improvement to the overall coverage. Otherwise, if ML finds it difficult to predict the coverage then the test is sent for simulation. After the initial training, the ML model re-training is performed according to another 50 newly simulated tests. In this methodology, the two major timing overheads, apart from simulation time, are training and prediction time. As training time is much greater than prediction time, the ML model is re-trained after simulating sufficient new tests. The whole framework is coded in Python. The coverage per test is extracted from the coverage reports and unified into a database using Python scripts. More details can be found in Appendix A.

Overall, only the concept of test pruning based on coverage prediction coincides with the approach proposed by LC.Wang [8]. The implementation is altogether new and different from any

of the previous work undertaken in this domain. Based on the ternary classification, i.e., covered, uncovered, and undecided, for a group of bins in a coverpoint, the decision to prune a test or not is taken. More number of undecided bins suggest a disparate test than the ones simulated earlier, thus pushed through simulation. Also, the model training and calculation of training error is very unique.

7. TRANSACTION-LEVEL STIMULUS OPTIMIZATION

The test-level constraints impart low controllability on the transaction generation in any sequence. Using the 24 test knobs described in the previous section, one can control over the cores, on which a sequence of transactions is executed in a parallel or sequential manner. Moreover, I can set the number of transactions in any sequence and their request and access type. Finer control over the attributes like address, data, and the order of the transactions are difficult at test-level. Moreover, certain events in the design are triggered only when certain type of transactions are exercised with a specific order in the design. Thus, the prediction of these event occurrences in the design using test-level features proved very difficult and requires feature extraction at fine-grained transaction-level. The work shown in this section is exclusive to all the efforts undertaken previously.

7.1 Methodology Overview

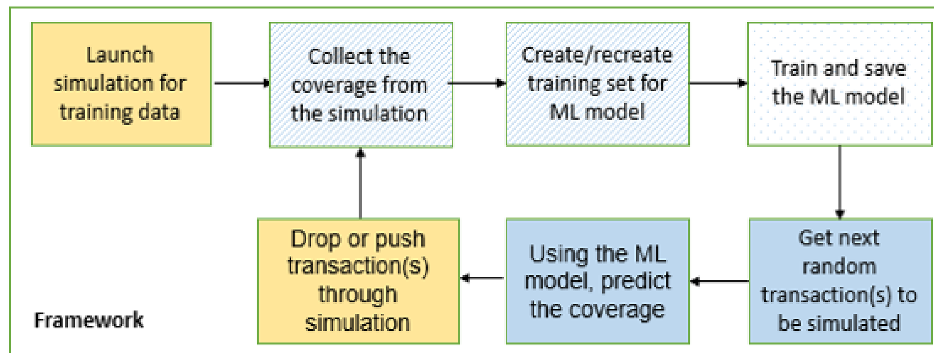


Figure 7.1: Transaction-level Stimulus Optimization Methodology

Empirically, it takes about 20% of overall simulation time to reach 80% design functional coverage and then extensive efforts are put in generating directed-random stimulus to cover the rest of the design. The aim of this experiment was to observe and record the transactional-level activity

during the random simulations for easy to achieve initial coverage and utilize this information to accelerate the rest of the verification activity. Transaction-level stimulus optimization is proposed that can recognize the impact of individual transactions on coverage and thereby determine transactions for the speedup of coverage closure. The general flow is depicted in Figure 7.1. Random simulations are initiated at the beginning to collect the training data for learning. The scoreboard in the existing environment is modified to record and display the per transaction activity in the simulation log. The training set is created by processing the simulation logs from initial random regressions.

The ML model is trained to classify/predict the coverage for a transaction or set of transactions. The initial regression proceeds until the model attain the training error below a certain threshold value. The trained model is saved and used for further stimulus optimization. Two optimization templates are devised as follows.

1. **Online transaction pruning**: This is an online approach where transactions are processed in the middle of simulating a sequence. If some transactions are classified by the ML model as not helpful on improving coverage, they are pruned out without being simulated.

2. **Offline directed sequence generation**: Based on ML predictions, a sequence composed of transactions improving the overall coverage is generated offline for simulation. This is similar to writing a directed test to target all the coverage holes, but instead of writing the test manually, the ML algorithm generates such directed sequence test.

These two templates are customized for two different types of coverage metrics for DUTs with and without FSMs mentioned in *section* 7.3 and 7.4 respectively.

7.2 Per-Transaction Coverage

The literature survey undertaken for this research depicts a heavy amount of efforts has been put in stimulus optimization was by closing the feedback loop from coverage to testcase generation. Thus, I was motivated to explore the scope of optimization at a finer level, a territory unexplored. The primary challenge faced was coverage collection for each transaction. The currently existing

industry tools provide us with the total coverage per test at the end of the simulation with no information about individual transaction's impact on the design state. Thus, it is difficult to create a labeled dataset of transaction activity using these coverage logs.

Scoreboard/Reference-model is a component of the testbench that contains checkers and verifies the functionality of a design. It receives the transaction information repacked in form of the monitor packet via TLM (Transaction Level Modeling) analysis port. The scoreboard/reference-model implements a correct transaction-level model of the DUT and thus an ideal setting to extract the transaction-level activity. The verification environment has a fully functional scoreboard logic having the accurate behavior of L1 and L2 cache systems as per the design specification. I modified the scoreboard logic to capture FSM and Non-FSM based per transaction coverage and display the relevant information.

7.3 FSM Transition Coverage

In a design with FSMs, one popular metric is the coverage of state transitions. A state transition depends on the current state and the new transaction, which can be characterized by its attributes. The general idea is shown in Figure 7.2. The ML model takes transaction attributes and current state as its input features and outputs the prediction of the next state. By examining the current state and the predicted next state, one can tell if transition coverage is improved. Initially, pilot random tests are performed and the results are applied to train the ML model. Once the model is well trained, it can be employed for either transaction pruning or directed sequence generation.

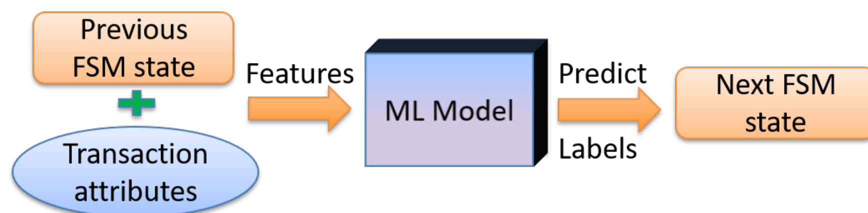


Figure 7.2: FSM Coverage Model

7.3.1 Data Extraction and Model Training

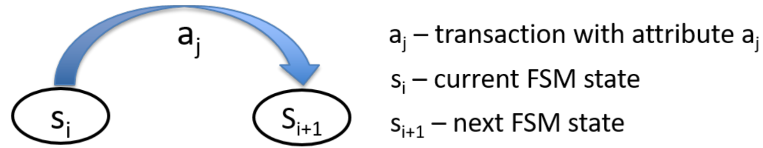
The FSM transition coverage metric evaluated in this study is MESI design state-transitions. In this design, cache coherency is implemented using MESI protocol, i.e., a particular entry in each L1 cache should be either in Modified, Exclusive, Shared or Invalid state. Overall, MESI state for cache entry is defined as combined MESI state for the corresponding cache entry in all of the four cores, e.g., $\{I, M, I, I\}$, represents that the cache entry in core 1 is in Modified state and Invalid for other cores.

To extract the design MESI state information, a new function is added in the scoreboard logic - "get_mesi_state". This function takes two input arguments: monitor packet and the CPU core. From the packet address field, tag and index values are extracted. Using these values the current MESI state for that cache line is extracted corresponding to all four cores. The design MESI state is recorded before and after each packet activity. The previous MESI state and packet attributes form the inputs and next MESI state become the output label. Based on the MESI protocol, not all states and transitions are valid. For example, $\{M, M, I, I\}$ is not a valid design MESI state for a cache line. Thus, the coverage metric consists of a total of 143 valid state transitions.

An overall state is represented by 16-bit binary label, 4 for each core. Hence, there are 22 input features bits to the ML model and 16 output labels to classify. The 22 input feature bits include 16-previous state, 4-core and 2-request type bits. The transaction address (can be included) and data attributes are not included. Different supervised machine learning algorithms are used to train the model using above dataset to compare its accuracy.

7.3.2 Coverage Prediction, Transaction Pruning, and Data Modeling

The coverage prediction and closure is further obtained by two strategies. In online transaction pruning, the application of the ML model is relatively straightforward. Before transactions of a sequence is fed to simulation, they are examined by the ML model to tell if they help improve transition coverage. Only those helpful transactions are fed to simulation while the others are pruned out.



Training Phase	Prediction Phase	Coverage Closure
Edge (s_i; s_{i+1}) with condition a_j is added to the TA graph	Predict transitions for state-TA (s_i; a_j) pairs that have not been simulated yet	On this predicted TA graph, apply weights on uncovered edges and visit them by applying Dijkstra's shortest path algorithm

Figure 7.3: Transaction Attribute Graph

The application of ML for offline directed sequence generation is more sophisticated. I construct a TA (Transaction Attribute) transition graph to help here as shown in Figure 7.3. It is similar to state transition diagrams for FSMs, except that its edges represent transitions associated with transaction attributes instead of FSM input (or transactions themselves). Usually, the FSM state space in a DUT is quite large and the dependence of transitions on transaction attributes is not known *a priori*. Therefore, the TA transition graph has to be constructed during transaction simulations. Suppose current state is s_i, by simulating a transaction with attribute $\mathbf{a}_j$, the next state s_{i+1} is reached. Then, an edge (s_i, s_{i+1}) with condition $\mathbf{a}_j$ is found and added to the TA transition graph. Figure 7.4 shows a TA graph on the training data set with 17 design states and 78 state-transitions.

After the ML model is trained, it can be used to predict transitions for state-TA $(s_i, \mathbf{a}_j)$ pairs that have not yet been simulated. Figure 7.5 shows a TA graph obtained after transaction prediction with 18 design states and 106 state-transitions. All the coverage holes are registered in a list with information of the required design FSM state (source node) and type of incoming transaction (transaction attributes) for each coverage holes in the design. Thus, this list contains all the undecided and unvisited edges from the predicted graph. On such predicted TA transition graph, large (small) weights are assigned to edges for transitions that have (not) been covered. Then, Dijkstra's shortest path algorithm is performed on the weighted predicted TA transition graph to reach from

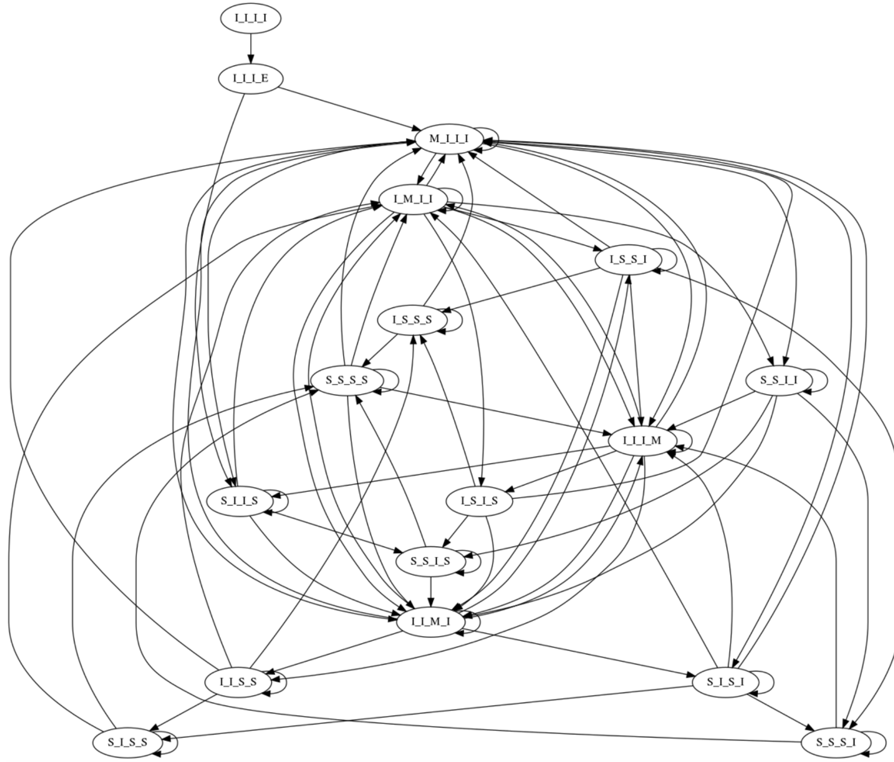


Figure 7.4: TA Trained Graph

current state to required state for exercise next coverage hole from the list. Normally, such paths contain many lower weight edges associated with transitions that have not been covered, hence filling the coverage on the move. Then, the attributes on these edges are utilized to find transactions that form a sequence for simulation.

The weights can be assigned in two manners. First way is to associate each edge with a probability value $p\ \epsilon(0,1)$ where a lower value would mean the model is indecisive and thus needs to be addressed for coverage closure with more importance. All the visited and unvisited edges are assigned weights 1 and 0 respectively. Thus, during the graph traversal, the algorithm will pick path with more weights which corresponds to the coverage holes. But in a larger context,all mispredicted and unvisited edges are categorized with weight 0 and rest with 1 as the aim is to fill all the holes and reach coverage closure.

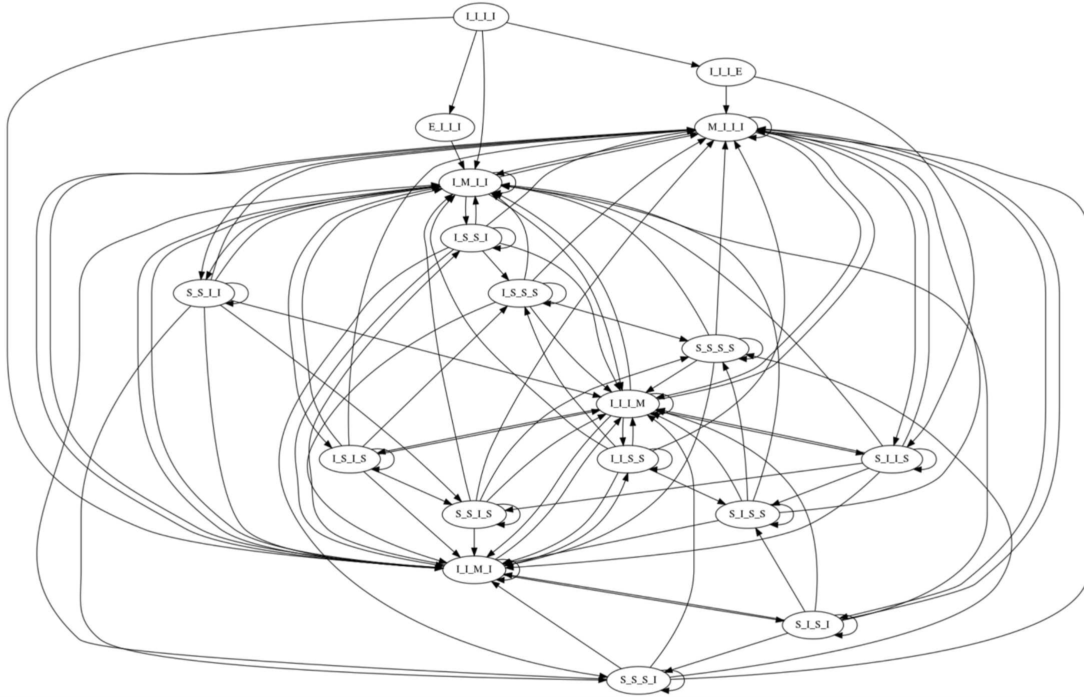


Figure 7.5: TA Predicted Graph

An example is shown in Figure 7.6 where currently the design is in INIT state. To reach the source node (S) of the next coverage hole, i.e., the source of the directed edge to be visited, there are two possible paths: I-A-B-S or I-C-D-E-S. If weights are assigned based on prediction confidence of model, the recommended path with lowest sum of weights is I-A-B-S with total 2.3 against 2.8 for I-C-D-E-S. Contrarily, assigning same weight to all coverage holes will result in I-C-D-E-S as the recommended path. Such graph traversal technique is employed till the coverage closure.

7.4 Non-FSM Event Coverage

In certain designs without FSM, it is of interest to verify the DUT under some events, e.g., cache hit. The occurrence of such events often depends on a sequence of transactions in a specific order. It is very difficult, if not impossible, for test level constraints or knobs to deterministically generate such ordered sequences. Therefore, I suggest to make use of machine learning for transaction level control on such sequences. As the coverage metric is non-FSM, there are no circuit states for tracking the history dependence in transactions. Hence, the history dependence needs to be

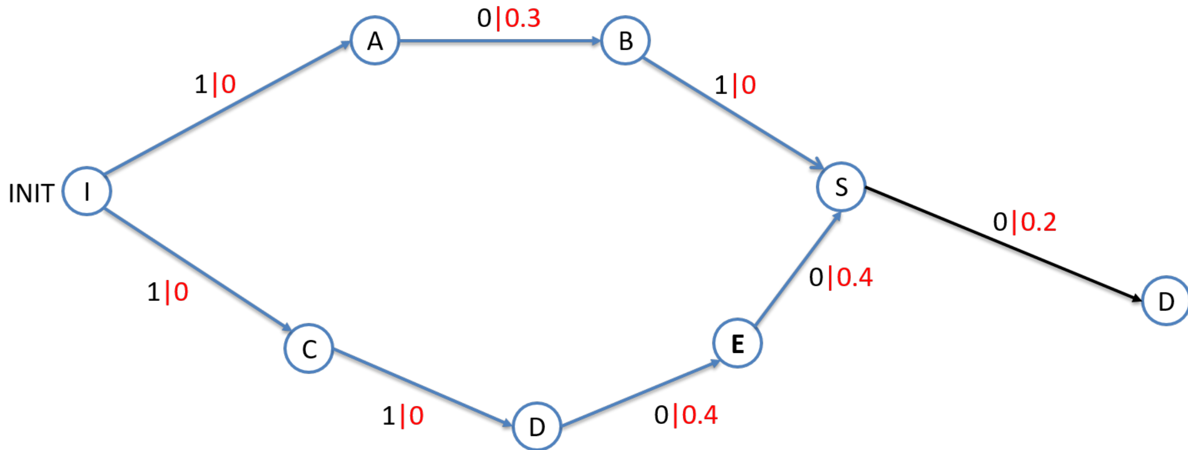


Figure 7.6: Shortest Path Traversal Example

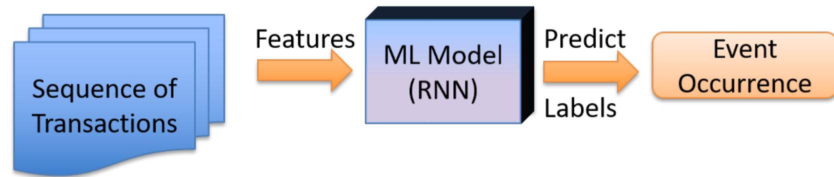


Figure 7.7: Non-FSM Coverage Model

explicitly captured by the ML model. In this approach, the LSTM (Long Short-Term Memory) [12] model is adopted, which is a popular recurrent neural network (RNN) model famous for handling time series and history dependence as shown in Figure 7.7. This methodology can be applied for sequence classification problem in hardware verification where task is to predict category of a sequence of inputs which span over time or space.

7.4.1 Data Extraction and Model Training

For transaction level optimization on non-FSM event coverage, the events of interests are L1 cache hits on each address bin in each core. A cache hit occurs when an address line requested by a transaction is present in L1 cache of the core. The event occurrence depends on the last access of the same address and type of transactions simulated in the past. The cache hit coverage metric consists of a total of 768 bins, i.e., each core of the four cores has 192 address space.

For a given test, the information of the sequence of transactions and functional coverage metric for each transaction is extracted using the simulation log. The information on the cache hit/miss per transaction is fetched by revising the scoreboard logic. During the cache update for each transaction, a flag "miss" is set to 1 in case there is no cache hit condition, which is checked by the function "get_way_hit" in the scoreboard. The data-points such as CPU core, index, and cache hit/miss for each transaction are printed in the simulation log. The input features for the LSTM are the attributes of an ordered sequence of w transactions and the output predicts if the event of interest is covered or not. The input features for the LSTM are the attributes of an ordered sequence of w transactions and the output predicts if the event of interest is covered or not. For each transaction, 3 main attributes are considered and their values are made one-hot coded, which forms the input feature set 22-bits wide per transaction: 4-core, 2-request type, and 16 address. The output label is one bit, 1-cache hit and 0-cache miss.

Intuitively, the w value controls the depth of simulated transactions history considered for making the prediction on the coverage metric. In the results section, the impact of varying w on training accuracy is shown. The live coverage database, created using Python, stores the current coverage and is updated on every new coverage bin hit. The training error is calculated similarly by taking the mean of the validation set error and 5-fold cross validation (CV) error. The initial random regression is paused when the classification error drops below a threshold value (10%).

7.4.2 Coverage Prediction and Transaction Pruning

Similar strategies are applied for filling the Non-FSM event coverage holes in the design. Figure 7.8 shows the step involved in each strategies.

For online transaction pruning, the next w transactions to be simulated according to a test are examined by the LSTM model. In the driver code, the next w transactions are requested from sequencer but not simulated yet. The attribute values for these w transactions are passed for coverage prediction using a Python script. If the LSTM predicts that they will not trigger the events that have not been covered, then the first transaction is pruned out without being simulated and new transaction is requested from sequencer and appended at the end of sequence for the

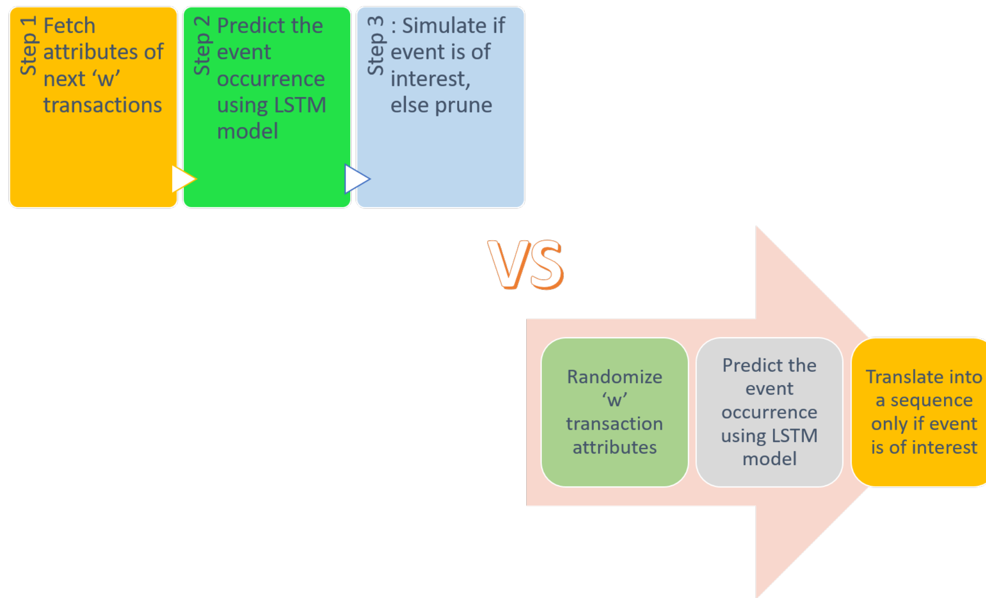


Figure 7.8: Non FSM Coverage: Online Pruning vs Offline Generation

coverage prediction. In other case, if the coverage prediction targets a coverage hole in the design, these w transaction are simulated in the same order. The LSTM model is re-trained periodically as more sequence of transactions are simulated.

In offline directed sequence generation, the constraints and knobs for a sequence are iteratively tuned till it would cover events of interests according to the LSTM model, and then the sequence is sent for simulation.

8. EXPERIMENTAL RESULTS

8.1 Test-Level Stimulus Optimization

In test-level stimulus optimization, the coverage for a new test is predicted using the coverage information collected from previously simulated tests. Then, using the coverage forecast, a decision is taken whether to stimulate the test or to prune it. The testcases which are not rewarding are barred from simulation, thus saving on total simulation time. This optimization technique comes with a cost of added machine learning model training and coverage prediction time. In this thesis, I have categories the results into two savings: simulation time and CPU-time. The simulation time is the sum total of the number of clock cycles each test ran for in a regression until the coverage closure is achieved. The information on the number of clock cycles for which the test is simulated is extracted from the simulation log. The CPU-time is the actual wall-clock time required for the whole regression to run with stimulus optimization until the coverage closure.

For the evaluation of this research, it is assumed that the stimulus/test generation time is small as compared to its simulation time. The input constraint matrix generation is performed by only once using Python script as shown in *Appendix A.1*. For each test, the sequence generation and driving of the transactions through the interface is done instantaneously by the testbench. The majority of time is taken for simulation of the transaction through DUT. Thus, this assumption is valid for the moderate size design used in this research and the skew between stimulus generation and simulation exponentially rises with increasing scope of design.

Three popular supervised machine learning classification algorithms are picked for comparison: deep neural network (DNN), Support Vector Machine (SVM), and Random Forest (RF). A DNN is an artificial neural network (ANN) with multiple layers of neurons between the input and output layers and finds the correct mathematical manipulation to turn the input into the output, whether it be a linear relationship or a non-linear relationship by re-accessing the weight on each neuron during the learning phase. SVM learns the correlation between input and output using

kernel functions. RF classifies the output by constructing a multitude of binary decision trees at training time and outputting the class by mean prediction of the individual trees. The training time, accuracy and memory footprint for all the three algorithms varies. The model parameters are fine-tuned for each algorithm to gain the best performance in terms of coverage prediction accuracy.

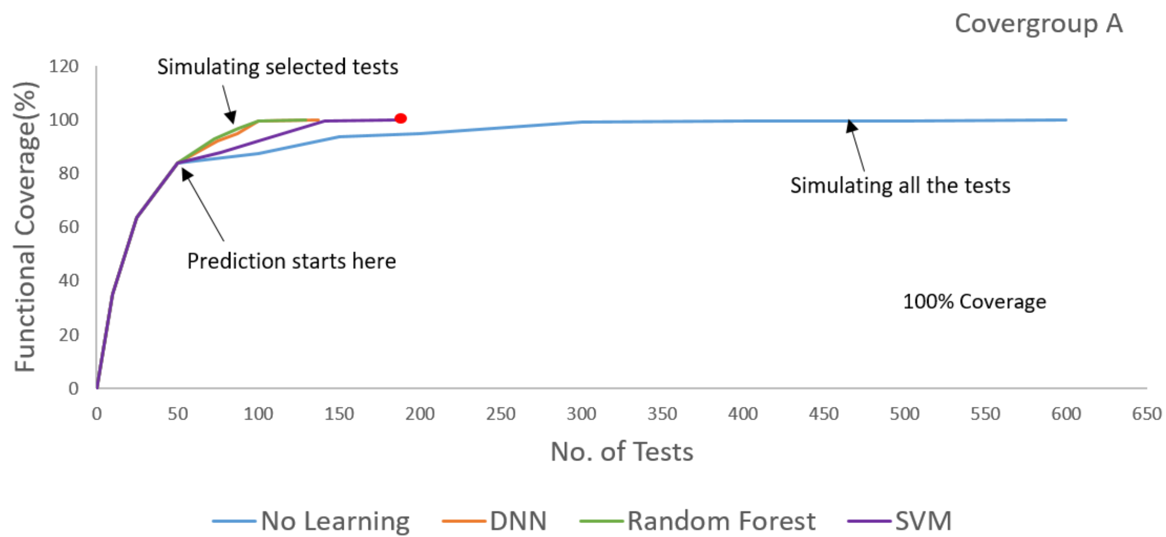


Figure 8.1: Coverage Closure for Cover-Group A with Test-Level Optimization

The eight metrics discussed in the *section* 6.3 were divided into two cover-groups based correlation with test knobs: cover-group A consisting of six coverage metrics (827 bins) and cover-group B containing two metrics (911 bins). The results for cover-group A are plotted in Figure 8.1. Without using machine learning, it took a total of 598 tests, which were simulated as part of random regression, to obtain 100% coverage for cover-group A as shown by the curve labeled "no learning" in blue. Although, it is quite evident that the 90% coverage mark is reached with 157 tests only. Thus, there is a high amount of redundancy in coverage hits once the coverage crosses the 80-90% mark when the curve begins to saturate.

On the other hand, the proposed stimulus optimization technique can help prune out the redun-

dant tests and only stimulate tests that target achieving coverage further. The training classification error for cover-group A was observed below 10% with simulation data from the initial 50 test, which shows a high correlation with test knobs. Once the machine learning model is ready for accurate coverage prediction, later only selected tests are simulated as shown in the plot. The figure depicts that coverage beyond the prediction mark is achieved with approximately at the same rate. The numbers of tests simulated for reaching 100% coverage for cover-group A by different methods are listed in Table 8.1.

ML Algorithm	No. of Tests
No Learning	587
DNN	137 (77% $\downarrow$)
SVM	185 (68.5% $\downarrow$)
RF	129 (78% $\downarrow$)

Table 8.1: No. of Tests for Coverage Closure of Cover-Group A

These results show $68-78\%$ reduction on the number of tests with RF producing the best result. RF classifier has better coverage prediction accuracy and overall coverage closure requires simulation of only 129 tests which is a mighty 78% reduction in the number of simulated tests. Table 8.2 shows the comparison of no. of tests simulated to achieve 100% coverage per each coverpoint. "snoop_request_bin_cov_out_mat" is the most difficult to exercise coverpoint in the design. DNN performs with a close prediction accuracy and requires a total of 137 tests for coverage closure. SVM has a higher misprediction rate and thus requires more number of tests to simulate for 100% coverage.

The results on cover-group B in Figure 8.2 show little benefit of using machine learning. I observed that the ML training for cover-group B is difficult to converge. Even after intensive training, due to high classification error on training data-set, the ML models still have a hard time in deciding coverage. This tells that cover-group B has a low correlation with test knobs for coverage prediction. The results reveal limitation of test level optimization and confirms the need

Coverpoints	No. of Tests to reach 100% coverage (w/o learning)	Random Tests to reach 100% coverage (with learning, RF model)
address_X_req_type	200	125
data_bin_cov_out_mat	100	72
fsm_cov_out_mat	100	78
proc_X_req_cov_out_mat	100	87
proc_X_snoop_cov_out_mat	100	71
snoop_request_bin_cov_out_mat	600	129

Table 8.2: Comparison of No. of Tests with and without Machine Learning (RF Model)

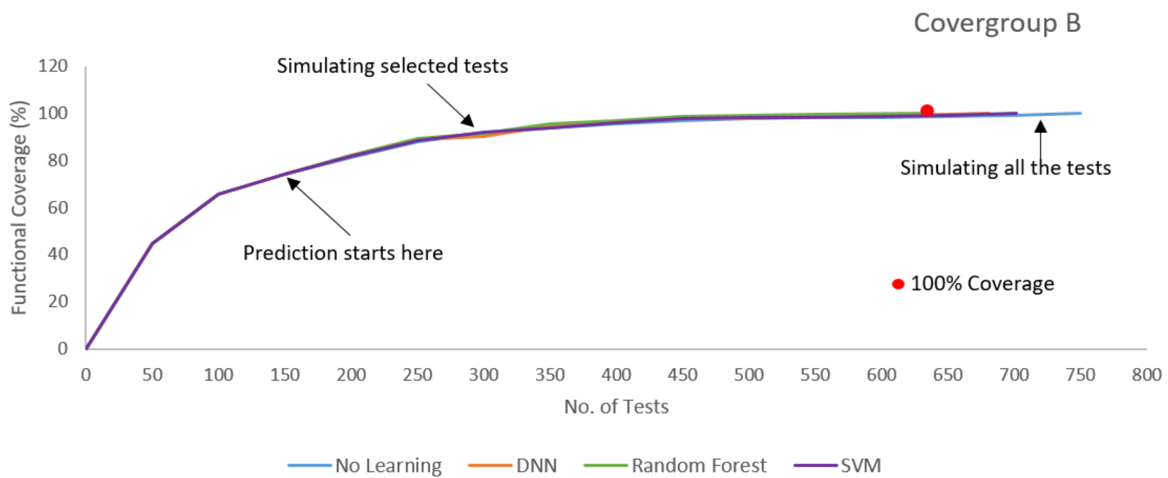


Figure 8.2: Coverage Closure for Cover-Group B with Test-Level Optimization

for transaction level optimization for coverage metrics in cover-group B.

8.2 Transaction-Level Stimulus Optimization

Transaction-level stimulus optimization is applied in two variants for two different coverage metrics in cover-group B shown earlier, which showed no significant improvement with test-level stimulus optimization.

8.2.1 FSM Based Coverage Metric

The FSM based transaction-level optimization is applied for MESI transition coverage. The next MESI state depends on the previous MESI state and the incoming transaction attributes. Thus, using the 22 input features the machine learning model was trained till a classification error got below the threshold value (10%). Two supervised ML algorithms are picked for performance comparison: DNN and RF. The model parameters were fine-tuned to obtain better prediction accuracy.

In fully random regression run, random tests were simulated till all 143 state transitions got covered to obtain the baseline result. The total simulation time required for coverage closure is the accumulated sum of number of clock cycles for which each test was simulated. Next, the similar coverage closure activity was driven using the two flavors of stimulus optimization: transaction pruning and sequence generation.

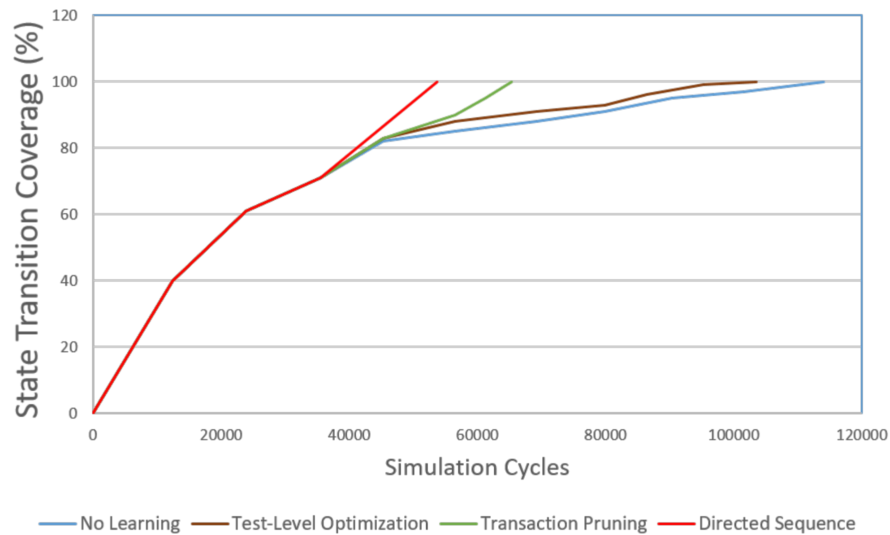


Figure 8.3: FSM Transition Coverage Closure using DNN Classifier

Figure 8.3 plot the simulation cycles required for coverage closure with fully random, test level optimization, transaction pruning and directed sequence generation using DNN classifier. With fully random regression, 80% coverage is reached in about 50K simulation cycles while it takes

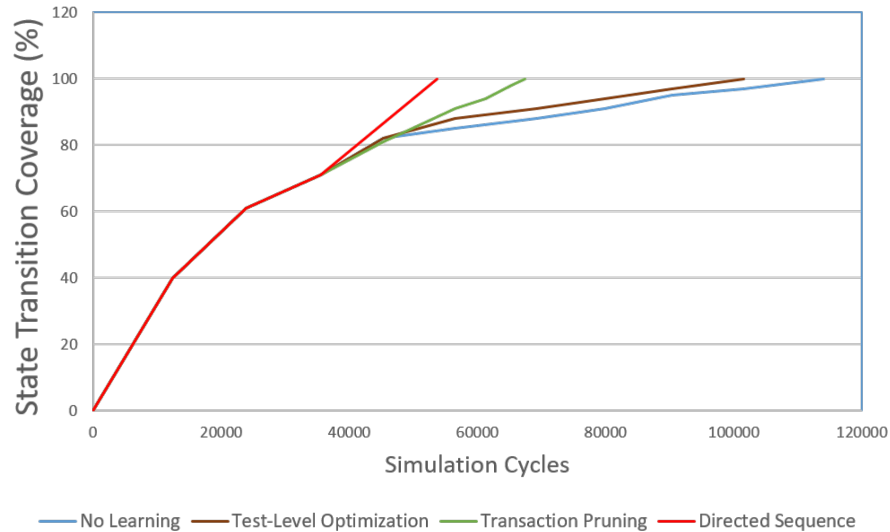


Figure 8.4: FSM Transition Coverage Closure using RF Classifier

an additional 65K cycles to reach 100% coverage. This implies that the coverage hits without ML have high redundancy. One can also see that the help from test level optimization is quite small. The fully random tests regression and test level optimization shows almost the same saturation behavior where rate of covering new MESI state transition diminishes as the regression progresses. In contrast, transaction level optimization achieves much more significant simulation cycle reduction. The transaction pruning technique and directed sequence generation technique can reduce simulation cycles by about 48% and 55%, respectively. On the predicted graph, applying shortest path strategy as described in the methodology is the more directed way to fill coverage holes and thus proves better results. However, applying weights on uncovered edges as a probability p or singular value *1* resulted in almost same number of transactions simulated for coverage closure; hence I adopted the later technique. Also, from Figure 8.4 it is evident that the reductions from DNN and RF are about the same.

The reduction in simulation cycles comes with a cost of the time taken by the ML model training and prediction. The data pre-processing consumes negligible time and hence can be ignored. I observed that the training time of random forest model is usually less than a half of DNN. Therefore, I advocate a random forest-based approach. The total CPU time and breakdown for the MESI

design are shown in Figure 8.5. One can see that the overall training and prediction time are much smaller than simulation time. The results show reductions in total CPU time of about 57% and 69% with transaction pruning and directed sequence generation, respectively.

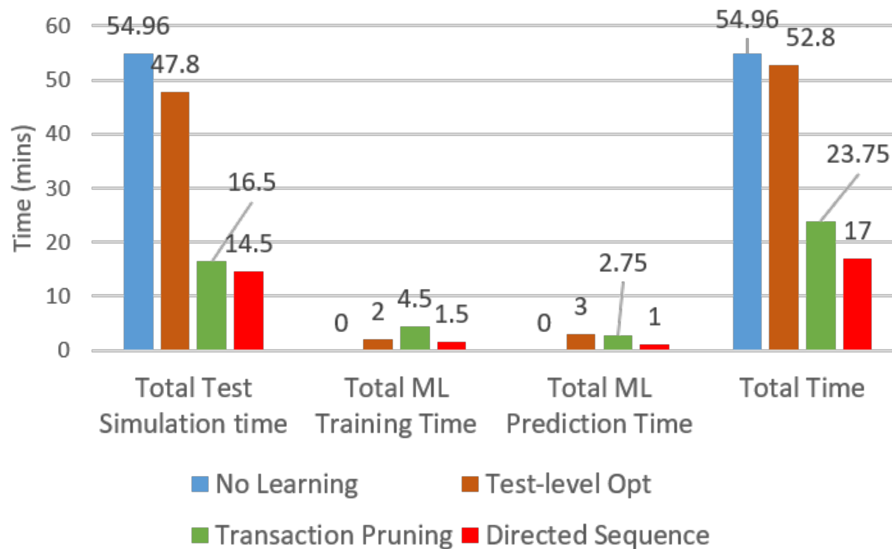


Figure 8.5: CPU Runtime comparison: RF based FSM Transition Coverage

8.2.2 Non-FSM Based Coverage Metric

In Non-FSM based coverage, the event of interest in the design depends on an order of group of transactions. The features for machine learning model are extracted from attributes of *w*-consecutive transactions as mentioned in the *section* 7.4. The RNN based machine learning algorithm applied for training such series data is the LSTM. Besides LSTM, Random Forest classifier is also implemented for comparison. The transaction info and transaction level coverage data are extracted from simulation logs and further processed using Python scripts for training.

Figure 8.6 shows the classification error on the training set by varying the length of transaction window w for feature extraction. Small window size *w* results in inaccurate prediction and too large w values cause over-fitting, thus higher error. Apparently, the LSTM-model has a higher accuracy compared to Random Forest and thus, LSTM with *w*=20 is only used for non-FSM event coverage

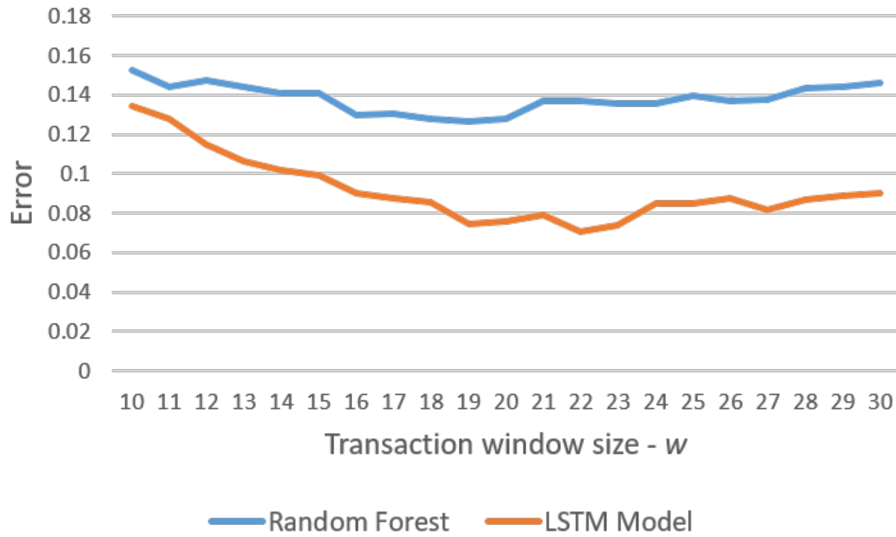


Figure 8.6: Classification Error with varying w for LSTM and RF Classifier

where the classification error is about 8%. In a separate experiment,the transaction attributes of all *w* transaction were collated as input attributes to train the DNN model. The model was unable to perform feature extraction and thus the training error never got below 40%. The primary reason for this nonperformance was that all the transactions were given equal value but that should not be the case.

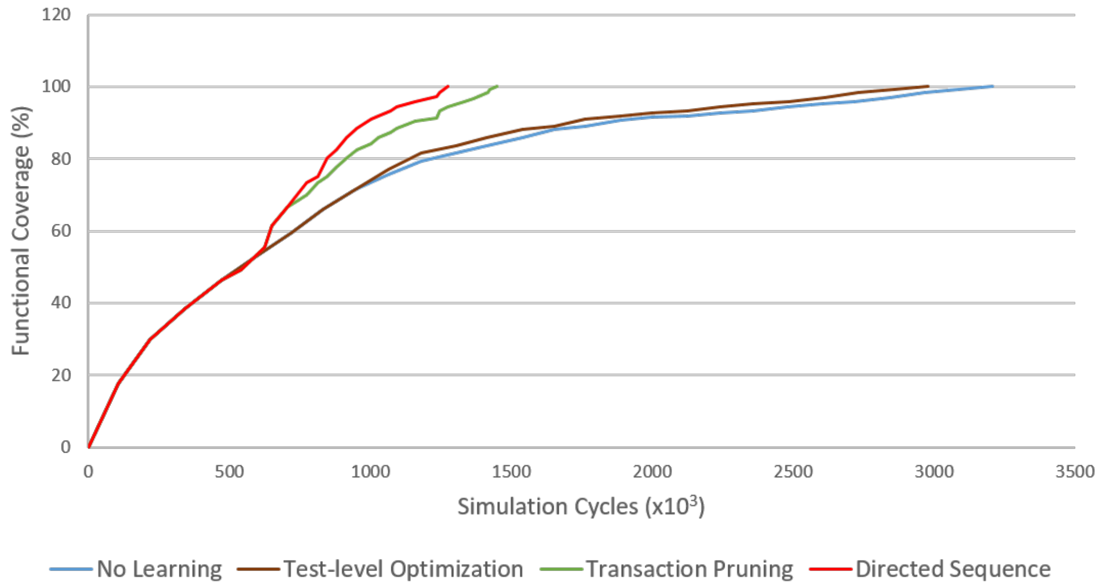


Figure 8.7: Non-FSM Coverage Closure using LSTM Classifier

Figure 8.7 shows the results achieved by applying LSTM-based transaction level optimization on non-FSM event coverage, also the part of cover-group B. The random regression and test level optimization take approximately same number of simulation cycles to reach 100% coverage, thus it is incompetent. Using the trained LSTM model, the coverage prediction is performed over a sequence of transactions for which least classification error was obtained, i.e.,w=20. The LSTM model training error dropped below 10% when total coverage was 60%. The remaining 40% coverage was achieved with transaction pruning and directed sequence generation, which resulted in reduction of 55% and 61% of simulation cycles, respectively, compared to random regression.

Figure 8.8 shows the total CPU runtime and its breakdown for evaluating the LSTM-based transaction level optimziation on non-FSM event coverage. Again, the LSTM training and prediction time are much less than simulation time. Compared to the random regression without machine learning, coverage closure with transaction pruning and directed sequence generation can reduce 65% and 72% of the total CPU runtime, respectively.

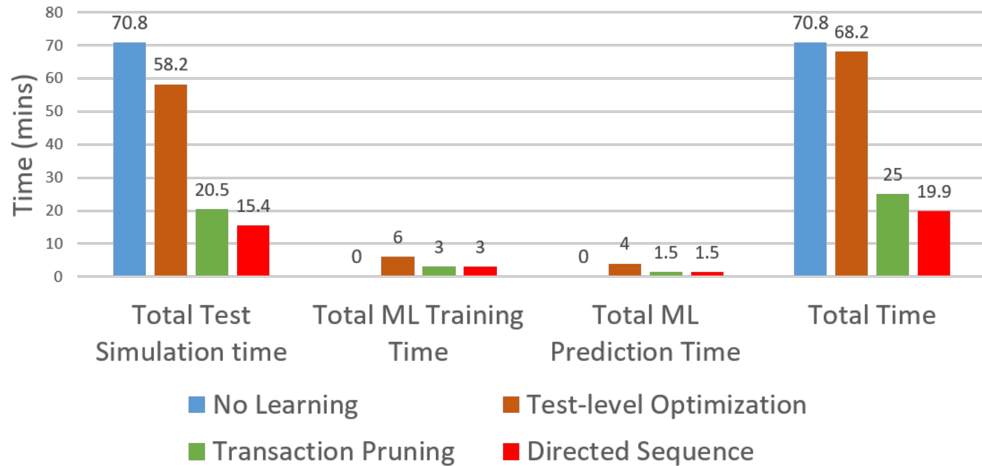


Figure 8.8: CPU Runtime comparison: LSTM based Non-FSM Event Coverage

8.3 Online Pruning vs Offline Generation

As discussed earlier, the coverage closure, i.e., hitting the coverage holes in the design, is conducted by two complementary approaches: online transaction(s) pruning and offline sequence generation. The former approach is easy to integrate with the testbench as the stimulus generation is done internally and only the decision about the effectiveness is performed using prediction scripts integrated into the driver. Although, this method is not the most optimized as there is low controllability in the transaction generation process. For example, to make a prediction on transaction-window size 20, information of 20 transaction attributes are fetched and used to predict the coverage. If the predicted coverage is non-useful then all these transactions are dropped and next 20 transactions are fetched. There is no control to permute only specific set of transactions in the window. Thus, the coverage closure obtained with online pruning is farther than the ideal transactions generation method to hit only the coverage holes.

On the other hand, offline sequence generation includes some more sophisticated methods like the generation of TA-graph. Such methodology requires more engineering efforts than just modifying the driver logic. For instance, for the FSM-based coverage metric, the information of TA-graph is utilized to sweep the uncovered transition in the predicted graph with least edge-

path traveled. If such a path exists in the graph, then generating and simulating the translated transactions sequence is the most optimal way to hit all the coverage holes in the design. Also for event-based coverage metric, transaction attributes can be re-arranged in Python to generate combinations for accelerated coverage closure. Hence, with offline sequence generation method, it is likely to reach coverage closure sooner but the trade-off is the time and labor to develop the methodology.

Figures 8.3, 8.4, and 8.7 show very marginal improvement for the transaction pruning over the directed sequence generation in terms simulation time-reduction when compared to the total simulation time when no learning is applied. Also, the total CPU time for both these methodologies is very comparable. Ergo, online pruning gives a remarkable run-time reduction with very few modifications to the framework while the coverage closure can be further compressed left with a more intelligent methodology like offline sequence generation at the cost of more engineering investment.

9. CONCLUSION

Machine learning is aptly suited for efficiency improvement in the area of simulation-based functional verification. The primary aim of this research is to exploit machine-learning capabilities to bring in more automation in test generation, obtain quicker coverage closure, and at the same time gain reduction of engineering effort as well as simulation time and test size. This attempts to accelerate bug detection in the design and reduce the overall time for verification in the chip design cycle. The factors that decide the feasibility for introducing such a learning-based stimulus optimization approach in practical use are the extent to which knowledge about DUT is required and how technically demanding it would be to construct and finetune the machine-learning based framework around the existing verification testbench.

Recently published work shows growing interest in the industry and academia in this area. In this research, I explored and suggest stimulus optimization as a favored avenue for the application of machine learning to accelerate coverage convergence. Simulation of hand-coded directed tests is the usual norm for covering complex and hard to hit design space which is a time extensive task and requires a thorough understanding of the design. Moreover, it is very difficult to generate a test with a sequence of custom-made transactions aimed to hit coverage holes. Thus, I explored machine learning based coarse- and fine-grained stimulus refinement via test level and transaction level exercise, respectively as an alternative approach for coverage closure.

The design used to conduct this research is quad-core mesi-based cache system. I modified the testbench to record transaction-level activity in logs and modified the driver for pruning decision and the rest of the framework is coded in Python. Thus, a feedback system from coverage to stimulus generation is implemented in the existing testbench with a minimum modification. This optimization strategy is completely automated, scalable to bigger design and requires little to moderate knowledge about the design.

Total eight functional coverage metrics were considered for coverage closure which covers important features in the design. Test-level optimization proved effective for six coverage metrics

with a 78% reduction in the number of simulated tests. However, test-level optimization failed to provide significant gains for the other two cases. I demonstrate the effectiveness of transaction-level optimization on the remaining coverage metrics and obtained coverage closure by two unique methods: online transaction pruning and offline sequence generation, resulting in a notable cutback of total simulation time by about 48% and 55% respectively. The overall methodology proved effective with around 70% reduction in total CPU time required for verification coverage closure. This work leads us to conclude that the complementary application of both of these optimization techniques is the recommended path for efficiency improvements in functional verification coverage convergence.

REFERENCES

[1] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[2] L. Breiman, "Random forest," *Statistics Department, University of California Berkeley*, January 2001.

[3] Hasim, Andrew, and Francoise, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," *Google USA*.

[4] S. Sokorac, "Optimizing random test constraints using machine learning algorithms," *Design Verification Conference*, 2017.

[5] R. Roy, C. Duvedi, S. Godil, and M. Williams, "Deep predictive coverage collection," *Design Verification Conference*, 2018.

[6] C. Ioannides and K. Eder, "Coverage-directed test generation automated by machine learning - a review," *ACM Transactions on Design Automation of Electronics and Systems*, vol. 17-1, p. 17, 2012.

[7] O. Guzeya, L.-C. Wang, J. R. Levitt, H. Foster, J. Bhadra, X. Feng, and M. S. Abadir, "Increasing the efficiency of simulation-based functional verification through unsupervised support vector analysis," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactionson*, vol. 29, pp. 138–148, January 2010.

[8] W. Chen, N. Sumikawa, L.-C. Wang, J. Bhadra, X. Feng, and M. S. Abadir, "Novel test detection to improve simulation efficiency - a commercial experiment," *Proceedings of the International Conference on Computer-Aided Design*, vol. 29, pp. 101–108, 2012.

[9] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," *Design Automation Conference*, pp. 286–291, 2003.

[10] F. Wang, H. Zhu, P. Popli, Y. Xiao, P. Bodgan, and S. Nazarian, “Accelerating coverage directed test generation for functional verification: a neural network-based framework,” *Great Lake Symposium on VLSI*, pp. 207–212, 2018.

[11] A. Liaw and M. Wiener, “Classification and regression by random forest,” *R News*, vol. 2/3, pp. 18–22, December 2002.

[12] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, pp. 1735–1780, 1997.

[13] Chollet, François, *et al.*, “Keras,” *https://keras.io*, 2015.

APPENDIX A

IMPLEMENTATION CODE

A.1 Random Parametric Test Simulation

Figure A.1 represents the bash script that is sourced to simulate a random test. The existing test generation code in the environment was modified such that all the possible randomization parameters/constraints at test level can be passed as input arguments. These are also referred to as "Test Knobs". There are 24 such input parameters that forms input features for test-level optimization mentioned in *chapter 6*.

```
declare -a arr=("simple_param_test")
source /home/grads/s/saumilgogri/ml_verif/ml_new/design_update-master/setup.bash

irun -f /home/grads/s/saumilgogri/ml_verif/ml_new/design_update-master/project/sim/cmd_line_comp_elab.f

#REQ_TYPE: 0--> READ 1-->WRITE
#CACHE_TYPE: 0--> ICACHE 1-->DACHE

for i in "${arr[@]}"
do
  irun -f /home/grads/s/saumilgogri/ml_verif/ml_new/design_update-master/project/sim/sim_cmd_line.f -access +rwc +UVM_TESTNAME=$i -svseed
 38 -covtest "$i"_98 +NUM_TRANS=34 +IS_PARALLEL=0 +FIX_ADD=1 \
 +SEL_CORE[0]=1 +READ_REQ[0]=1 +WRITE_REQ[0]=1 +DCACHE[0]=1 +ICACHE[0]=0 \
 +SEL_CORE[1]=1 +READ_REQ[1]=0 +WRITE_REQ[1]=0 +DCACHE[1]=1 +ICACHE[1]=0 \
 +SEL_CORE[2]=1 +READ_REQ[2]=0 +WRITE_REQ[2]=1 +DCACHE[2]=1 +ICACHE[2]=0 \
 +SEL_CORE[3]=1 +READ_REQ[3]=1 +WRITE_REQ[3]=0 +DCACHE[3]=0 +ICACHE[3]=1 \
 +UVM_VERBOSITY=UVM_FULL
 imc -exec /home/grads/s/saumilgogri/ml_verif/ml_new/design_update-master/project/sim/param_tescases/gen_cov_98.txt
  mv /home/grads/s/saumilgogri/ml_verif/ml_new/design_update-master/project/sim/irun.log /home/grads/s/saumilgogri/ml_verif/ml_new/design
_update-master/project/sim/logs/"$i"_98.log
done
```

Figure A.1: Bash Script for Test Simulation

The setup.bash sets all the required Unix environment variables required to invoke Cadence Incisive package for design simulation. First step is to compile and elaborate the design. In this step all the testbench files are compiled and all UVM components are built in top-down approach and then they are connected in bottom-up fashion. Now, the testbench is ready for simulation.

Simulation is kicked off with all the test constraints declared as input arguments. For example, '+NUM_TRAN=8' means constraint NUM_TRAN is assigned thes value 8. Each test can kick off at most 4 sequences sequentially or in parallel, based on values of constraint SEL_CORE[0-3] and

IS_PARALLEL. Constraints REQ_TYPE and CACHE_TYPE controls the nature of transactions in each of these sequences.

I created Python-based random script that creates an input matrix, as shown in Figure A.2, with multiple rows and 24 columns. Each column corresponds to a test constraint and each row in that matrix forms set of input constraints for a test simulation. This matrix forms the input matrix in test-level stimulus optimization technique.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	Seed	#Transaction	Fix Add	Parallel Seq	CORE[0]	READ_REQ[0]	WRITE_REQ[0]	DCACHE[0]	ICACHE[0]	CORE[1]	READ_REQ[1]	WRITE_REQ[1]	DCACHE[1]	ICACHE[1]	CORE[2]	READ_REQ[2]	WRITE_REQ[2]	DCACHE[2]
2	71	10	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	1
3	25	5	1	0	0	1	0	0	1	0	0	0	1	0	1	1	0	0
4	96	10	0	0	1	1	0	0	1	0	1	0	0	1	0	1	0	1
5	10	6	1	0	0	1	0	0	1	0	1	0	0	1	1	1	0	0
6	84	5	1	0	0	0	0	1	0	0	0	1	1	0	0	1	0	0
7	56	7	0	1	0	1	0	0	1	1	1	0	1	0	1	0	1	1
8	5	5	1	0	1	1	0	0	1	0	1	0	0	1	1	0	0	1
9	58	5	0	0	1	1	0	1	0	0	1	0	1	0	1	0	0	1
10	69	10	0	0	0	0	0	1	0	1	0	1	1	0	0	1	0	0
11	32	8	1	0	0	1	1	1	0	0	1	0	1	0	0	1	0	0
12	54	5	1	0	1	1	0	0	1	1	0	1	1	0	1	1	0	1
13	62	6	0	1	1	1	0	1	0	1	1	1	1	0	1	0	0	1
14	66	7	0	1	0	0	1	1	0	0	0	0	1	0	1	1	0	0
15	1	7	0	0	0	1	0	0	1	1	1	0	0	1	0	0	0	1
16	57	6	0	0	1	1	0	0	1	1	1	0	0	1	0	0	0	1
17	67	9	0	0	1	0	1	1	0	1	1	1	1	0	1	0	1	1
18	49	10	1	0	1	1	0	0	1	0	1	0	0	1	1	1	1	1
19	75	9	1	0	1	1	0	0	1	1	0	1	1	0	1	1	1	1
20	72	6	0	0	0	1	0	0	1	0	1	0	0	1	0	1	0	1
21	57	8	0	1	0	1	0	0	1	0	1	0	0	1	1	1	0	0
22	81	8	0	1	0	1	0	0	1	0	1	0	0	1	1	1	1	1
23	57	8	0	0	1	1	0	0	1	0	1	1	1	0	0	1	0	0
24	70	9	0	1	1	0	0	1	0	0	1	0	0	1	1	1	0	1
25	14	8	0	1	0	0	0	1	0	1	1	0	0	1	0	0	0	1
26	33	5	1	0	1	1	0	0	1	0	0	0	1	0	1	0	0	1
27	38	5	1	0	0	1	0	0	1	1	1	0	0	1	1	1	0	1
28	100	6	1	0	0	1	1	1	0	1	1	0	0	1	0	1	0	1
29	8	9	1	0	0	0	1	1	0	1	1	0	0	1	0	1	0	0
30	45	8	1	0	1	1	0	0	1	0	1	1	1	0	1	1	0	0
31	92	10	1	0	1	1	1	1	0	1	0	1	1	0	0	1	0	1
32	61	8	1	0	0	1	0	0	1	0	1	1	1	0	0	1	0	0
33	76	10	0	1	0	0	0	1	0	1	0	1	1	0	1	1	0	0
34	74	7	1	0	0	1	0	0	1	1	0	1	1	0	0	1	0	0
35	60	8	0	1	0	1	0	0	1	1	1	0	1	0	1	0	1	1
36	16	8	0	1	0	1	1	1	0	0	1	0	0	1	0	1	0	0
37	21	5	1	0	0	0	1	1	0	1	0	0	1	0	0	0	0	1

Figure A.2: Input Matrix - Test Constraints

A.2 Coverage Collection per Test

Incisive Metric Center is the coverage collection and analysis tool from Cadence. The imc command in the bash file records the coverage of the test and dumps it in as coverage report (.txt format) with the file name specified in the command. This report contains the coverage information for each bin in every coverpoints and covergroups. Figure A.3 shows a snippet of the coverage report. For each coverpoint, its coverage is capture in range of 0-100% and for each bin the coverage is either 0 or 100%.

```
Name                            Average, Covered Grade         Line  Source Code
----------------------------------------------------------------------------------------------------
system_bus_monitor_c::cover_sbus_packet 18.13%, 8.91% (32/359)        18 (system_bus_monitor_c.sv) covergroup cover_sbus_packet;
|--REQUEST_TYPE                 25.00% (1/4)                   21 (system_bus_monitor_c.sv) REQUEST_TYPE: coverpoint  s_packet.bus_req_type;
| |--auto[BUS_RD]               0.00% (0/1)                    21 (system_bus_monitor_c.sv) REQUEST_TYPE: coverpoint  s_packet.bus_req_type;
| |--auto[BUS_RDX]              0.00% (0/1)                    21 (system_bus_monitor_c.sv) REQUEST_TYPE: coverpoint  s_packet.bus_req_type;
| |--auto[INVALIDATE]           0.00% (0/1)                    21 (system_bus_monitor_c.sv) REQUEST_TYPE: coverpoint  s_packet.bus_req_type;
| |--auto[ICACHE_RD]            100.00% (5/1)                  21 (system_bus_monitor_c.sv) REQUEST_TYPE: coverpoint  s_packet.bus_req_type;
|--REQUEST_PROCESSOR            25.00% (1/4)                   22 (system_bus_monitor_c.sv) REQUEST_PROCESSOR: coverpoint s_packet.bus_req_proc_num;
| |--auto[REQ_PROC0]            100.00% (5/1)                  22 (system_bus_monitor_c.sv) REQUEST_PROCESSOR: coverpoint s_packet.bus_req_proc_num;
| |--auto[REQ_PROC1]            0.00% (0/1)                    22 (system_bus_monitor_c.sv) REQUEST_PROCESSOR: coverpoint s_packet.bus_req_proc_num;
| |--auto[REQ_PROC2]            0.00% (0/1)                    22 (system_bus_monitor_c.sv) REQUEST_PROCESSOR: coverpoint s_packet.bus_req_proc_num;
| |--auto[REQ_PROC3]            0.00% (0/1)                    22 (system_bus_monitor_c.sv) REQUEST_PROCESSOR: coverpoint s_packet.bus_req_proc_num;
|--REQUEST_ADDRESS              20.00% (4/20)                  23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[0:11]                 100.00% (2/1)                  23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[12:23]                0.00% (0/1)                    23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[24:35]                100.00% (1/1)                  23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[36:47]                100.00% (1/1)                  23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[48:59]                100.00% (1/1)                  23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[60:71]                0.00% (0/1)                    23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[72:83]                0.00% (0/1)                    23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[84:95]                0.00% (0/1)                    23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[96:107]               0.00% (0/1)                    23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[108:119]              0.00% (0/1)                    23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[120:131]              0.00% (0/1)                    23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[132:143]              0.00% (0/1)                    23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[144:155]              0.00% (0/1)                    23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[156:167]              0.00% (0/1)                    23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[168:179]              0.00% (0/1)                    23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[180:191]              0.00% (0/1)                    23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[192:203]              0.00% (0/1)                    23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[204:215]              0.00% (0/1)                    23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[216:227]              0.00% (0/1)                    23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
| |--auto[228:255]              0.00% (0/1)                    23 (system_bus_monitor_c.sv) REQUEST_ADDRESS: coverpoint s_packet.req_address{
|--SNOOP_REQUEST                6.67% (1/15)                   26 (system_bus_monitor_c.sv) SNOOP_REQUEST: coverpoint s_packet.bus_req_snoop{
| |--auto[0]                    100.00% (5/1)                  26 (system_bus_monitor_c.sv) SNOOP_REQUEST: coverpoint s_packet.bus_req_snoop{
| |--auto[1]                    0.00% (0/1)                    26 (system_bus_monitor_c.sv) SNOOP_REQUEST: coverpoint s_packet.bus_req_snoop{
| |--auto[2]                    0.00% (0/1)                    26 (system_bus_monitor_c.sv) SNOOP_REQUEST: coverpoint s_packet.bus_req_snoop{
| |--auto[3]                    0.00% (0/1)                    26 (system_bus_monitor_c.sv) SNOOP_REQUEST: coverpoint s_packet.bus_req_snoop{
| |--auto[4]                    0.00% (0/1)                    26 (system_bus_monitor_c.sv) SNOOP_REQUEST: coverpoint s_packet.bus_req_snoop{
| |--auto[5]                    0.00% (0/1)                    26 (system_bus_monitor_c.sv) SNOOP_REQUEST: coverpoint s_packet.bus_req_snoop{
| |--auto[6]                    0.00% (0/1)                    26 (system_bus_monitor_c.sv) SNOOP_REQUEST: coverpoint s_packet.bus_req_snoop{
| |--auto[7]                    0.00% (0/1)                    26 (system_bus_monitor_c.sv) SNOOP_REQUEST: coverpoint s_packet.bus_req_snoop{
| |--auto[8]                    0.00% (0/1)                    26 (system_bus_monitor_c.sv) SNOOP_REQUEST: coverpoint s_packet.bus_req_snoop{
```

Figure A.3: Coverage Report for a Simulated Test

Each coverage bin is the output label for coverage prediction. For each test and each coverpoint, I store the coverage as an array of bits, where each bit corresponds to a bin in that coverpoint and a value 0 or 1 is assigned based on the values in the corresponding coverage report. This array becomes the output matrix for coverage prediction and stimulus optimization as show in Figure A.4. Each row entry is the per bin coverage for a particular test.

Also, the coverage reports from all the simulated tests are processed by a Python script to

	A	B	C	D	E	F	G	H	I	J	K	L	
1	REQUEST_TYPE	REQUEST_TYPE	REQUEST_TYPE	REQUEST_TYPE	REQUEST_PROCESSOR	REQUEST_PROCESSOR	REQUEST_PROCESSOR	REQUEST_PROCESSOR	REQUEST_ADDRESS	REQUEST_ADDRESS	REQUEST_ADDRESS	REQUEST_ADDRESS	R
2	1	1	0	1	1	0	1	0	1	1	0	1	
3	1	0	0	1	0	0	1	1	0	0	0	1	
4	0	0	0	1	1	0	0	0	1	1	0	1	
5	0	0	0	1	0	0	1	0	0	0	1	0	
6	0	0	0	0	0	0	0	0	0	0	0	0	
7	1	1	0	0	0	1	1	1	0	0	0	0	
8	1	1	0	1	1	0	1	1	0	0	0	0	
9	1	0	0	1	1	0	1	1	1	1	1	0	
10	0	1	0	1	0	1	0	1	1	1	1	1	
11	1	0	0	0	0	0	0	1	0	0	0	0	
12	1	1	0	1	1	1	1	0	1	0	0	0	
13	1	1	0	1	1	1	1	1	1	1	0	1	
14	0	0	0	1	0	0	1	0	0	1	1	1	
15	0	0	0	1	0	1	0	0	0	1	1	1	
16	0	0	0	1	1	1	0	0	1	1	1	1	
17	0	1	0	0	1	1	1	0	0	0	0	0	
18	1	1	0	1	1	0	1	1	0	0	0	0	
19	0	1	0	1	1	1	1	0	0	1	0	0	
20	0	0	0	0	0	0	0	0	0	0	0	0	
21	0	0	0	1	0	0	1	0	0	1	1	0	
22	0	1	0	1	0	0	1	1	1	0	1	0	
23	0	0	0	1	1	0	0	0	0	0	1	0	
24	1	0	0	0	1	0	1	0	0	0	0	0	
25	0	0	0	1	0	1	0	0	1	1	1	0	
26	1	0	0	1	1	0	1	1	0	0	0	0	
27	1	0	0	1	0	1	1	0	0	0	0	1	
28	0	1	0	1	0	1	0	1	0	0	1	0	
29	0	0	0	1	0	1	0	0	0	0	0	0	
30	0	0	0	1	1	0	1	1	0	0	1	0	
31	0	1	0	0	1	1	0	0	0	0	0	0	
32	0	1	0	0	0	0	0	1	0	0	0	0	
33	0	1	0	1	0	1	1	0	1	1	1	0	
34	1	1	0	0	0	1	0	1	0	0	0	0	
35	1	1	0	0	0	1	1	0	0	0	0	0	
36	1	0	0	0	0	0	0	1	0	0	0	0	
37	1	1	0	0	0	1	0	1	0	0	0	0	
38	0	0	0	1	1	1	0	0	1	1	1	1	

Figure A.4: Output Matrix - Coverage

evaluate the current overall design coverage.

A.3 Training Deep Neural Network Model

A Deep Neural Network is series of layers of well connected neurons used to supervised machine learning. The weights on each neurons are adjusted as we train the model using the labeled training data-set for accurate prediction. I have used 'Keras' [13] library to create such instance of deep neural network model. From Keras, I imported modules such as Sequential, Dense, Dropout, Activation, and SDG. In Figure A.5 I have shown how to instantiate and train a DNN model.

```
model = Sequential()
model.add(Dense(80, activation='relu', input_dim=X_train.shape[1]))
model.add(Dropout(0.2))
model.add(Dense(160, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(80, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(Y_train.shape[1], activation='sigmoid'))

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X_train, Y_train,epochs=100,batch_size=int(X_train.shape[0]/4))
```

Figure A.5: Instance of DNN model

X_train and Y_train are the training data-set where each row in X_train consists of all input attributes and the same row in Y_train has the values to the output labels. The first layer of DNN is the input layer and the size is determined by the number of the inputs attributes. Next, I define two hidden layers with 160 and 80 neurons respectively, which are followed by the output layer. The size of the output layer is decided by the total number of output labels to be predicted. I have assigned a dropout value to each layer to avoid over-fitting i.e to reduce the model bias towards the training data. ‘Relu’ is a linear function, used for activation in each layer except the output layer. The model for logistic regression, so the prediction value needs to be between 0 and 1, and thus ‘sigmoid’ function is used for the output activation.

Once the model architecture is ready, we compile it and set the ’binary cross-entropy’ as the loss function. This is used for back-propagation i.e. adjusting weights on the neurons. Then I train the model in many epochs. In each epoch, a batch (a subset of the training set) of given batch-size is created for training. I train the model in batches to avoid over-fitting. During training, the model prints the loss function error after each epoch. The value of epoch is decided to have the model training error just below a threshold value. Once the model is trained, it ready for the prediction on input data-set with unknown outputs.

A.4 Training Random-Forest Model

Random forest is a disjoint set of multiple decision trees used for supervised machine learning application. To create the model, I have used the ‘RandomForestClassifier’ module from the powerful ‘sklearn’ library. Below is how I created a Random Forest model instance and trained it using labeled dataset.

$model = RandomForestClassifier(n_estimators = 1000,$

$criterion = `entropy', random_state = 42)$

$model.fit(X_train, Y_train)$

The value of n_estimators decides the number of decision trees in the forest. Other arguments are kept as default. From experiments I noticed that having a fewer number of decision trees will produce an inaccurate model while more trees increase model bias towards the training data-

set. Thus, its value was set using trial and error to have least training error calculated using the validation set and 5-fold cross-validation.

A.5 Training LSTM Model

LSTM model is used to predictions on the series data. LSTM has an inbuilt memory, which stores information from previous input data. Here the input matrix is 3-dimensional where the depth is the transaction window size w. I used keras library to build the LSTM network as shown in Figure A.6.

```
model = Sequential()
model.add(LSTM(10*L, input_shape=(L, 8)))
#model.add(LSTM(100, input_shape=(L, 2), dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1,activation='sigmoid'))
#model.add(LSTM(1, return_sequences=True))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
model.fit(x_train_seq, y_train_seq,epochs=10,batch_size=50)

# serialize model to JSON
model_json = model.to_json()
with open("lstm_cache_predict_model.json", "w") as json_file:
  json_file.write(model_json)
#serialize weights to HDF5
model.save_weights("lstm_cache_predict_model.h5")
print("Saved model to disk")


# load json and create model
json_file = open('/home/grads/s/saumilgogri/ml_verif/ml_new/design_update-master/project/cache_hit_miss_pred/lstm_cache_predict_model.j
son', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("/home/grads/s/saumilgogri/ml_verif/ml_new/design_update-master/project/cache_hit_miss_pred/lstm_cache_predic
t_model.h5")
print("Loaded model from disk")
```

Figure A.6: Instance of LSTM model

The trained model is then saved into JSON format. The weights corresponding to each LSTM nodes is saved in a separate file with extension h5. The saved model is then loaded for prediction.