Enhancement of SBST Techniques for Detection of Processor Faults

Thesis submitted in partial fulfilment of the requirements for the award of the degree of

Doctor of Philosophy

in

Computer Science and Engineering

Submitted by

Vasudevan.M.S

Under the guidance of

Dr. Santosh Biswas and Dr. Aryabartta Sahu

Department of Computer Science and Engineering

Indian Institute of Technology Guwahati

June, 2020

Abstract

At-speed testing of processors is extremely difficult with any external testing technique, therefore, Software-based self-testing (SBST) is introduced for efficient at-speed testing of processors. Evolutionary approaches are used for the automated synthesis of SBST codes. However, SBST development has been exceedingly difficult due to the sophisticated circuits of the modern processor hardware. In addition, the momentary nature of faults necessitates a careful and extensive test application. This thesis comprises four major contributions which address the challenges in SBST code synthesis, application, and optimization phases.

In the first contribution, we present a greedy cover-based strategy for automated SBST code synthesis, where the instruction sequences that detect the freshly identified faults are preserved throughout the evolutionary process to identify the hard-to-test faults of the processor. This preservation of test programs that detect hard-to-test faults in the evolutionary process increases the fault coverage. Also, a selection probability is estimated from the testability properties of the processor components and assigned to every instruction to accelerate the test synthesis. In addition, we have used high-level behavioral fault models for modeling processor hardware faults without using gate-level details of the processor. In this contribution, we synthesize high-quality SBST programs with 96.32% fault coverage for a MIPS processor and 95.8% fault coverage for a Leon3 processor with the detection of 40% of the hard faults.

However, the test synthesis time required for automated SBST synthesis is high for the existing evolutionary approaches. So, an advanced SBST technique, termed as Rapid SBST (RSBST), is proposed in the second contribution that reduces the overall test synthesis time by reusing the simulation responses of existing test programs of identical observability. The fault diagnostic characteristics of test programs are reused for the test quality evaluation if these test programs produce similar values in the observable locations

of the processor. We exploit this reusability to enhance the speed of the test synthesis process. In a nutshell, this contribution develops a faster technique for synthesizing high-quality SBST programs. This strategy develops test solutions with 96.1% fault coverage for the MIPS processor in 90 hours and test solutions with 95.5% coverage for the Leon3 processor in 98 hours. To achieve this, we have exploited the reusability of 82.1% of test solutions for the MIPS processor and the reusability of 80.8% of test solutions for the Leon3 processor during the evolutionary test synthesis.

In the third contribution, the test codes are optimized with the help of enhanced assembly code compaction techniques. The tradeoff between test compaction and computational effort required for the test compaction is dealt with two compression stages. In the first stage, the test program is preprocessed using a novel instruction removal technique that makes use of data dependence graphs to identify and eliminate independent and redundant instruction groups. In the second stage, an instruction restoration technique delivers a high compaction rate with the help of low-cost, high-level logic simulations for test quality evaluation. In this contribution, SBST programs are efficiently compacted after test synthesis phase by identifying and removing 19% of the redundant instructions of the SBST program consuming 72.24% of the computational cost of instruction-by-instruction redundancy check.

In the online test application mode, SBST schemes provide high fault coverage but incur long detection latencies in case of intermittent faults, due to large size and longer execution time of the test codes. In the last contribution, a fragmented SBST method is developed which develops a reliable set of SBST code fragments of minimal fault detection latency to detect the intermittent faults and enhance the reliability of the system. Also, these code fragments suffer inconsiderable overall fault coverage drop, compared to the coverage of the complete SBST test code. In this contribution, test programs with a better trade-off between execution time and fault coverage are selected during the test synthesis phase and are applied for the online testing of the processor. In our experiment, a group of 20 smaller test programs of 80% fault coverage and adequate overall coverage of 96% is observed to have maximum reliability to replace the optimal test program with 96.3% for the online testing of MIPS processor for our input data set.

Declaration

I certify that:

a. The work contained in this thesis is original and has been done by me under the guidance of my supervisors.

b. The work has not been submitted to any other Institute for any degree or diploma.

c. I have followed the guidelines provided by the Institute in preparing the thesis.

d. I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

e. Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Vasudevan.M.S

Certificate

This is to certify that this thesis entitled, **"Enhancement of SBST Techniques for Detection of Processor Faults"**, being submitted by **Vasudevan.M.S**, to the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, for partial fulfillment of the award of the degree of Doctor of Philosophy, is a bonafide work carried out by him under our supervision and guidance. The thesis, in our opinion, is worthy of consideration for award of the degree of Doctor of Philosophy in accordance with the regulation of the institute. To the best of our knowledge, it has not been submitted elsewhere for the award of the degree.

....

Dr. Santosh Biswas
Associate Professor
Department of Computer Science and Engineering
IIT Guwahati

....

Dr. Aryabartta Sahu
Associate Professor
Department of Computer Science and Engineering
IIT Guwahati

Dedicated to

my wife and my mother

Acknowledgments

I wish to express my deepest gratitude to my supervisors, Prof. Santosh Biswas and Prof. Aryabartta Sahu for their valuable guidance, inspiration, and advice. I feel very privileged to have had the opportunity to learn from, and work with them. Their constant guidance and support not only paved the way for my development as a research scholar but also changed my personality, ability, and nature in many ways. I have been fortunate to have such advisors who gave me the freedom to explore on my own and at the same time the guidance to recover when my steps faltered. Besides my advisors, I would like to thank the rest of my thesis committee members: Prof. J. K. Deka, Prof. H. K. Kapoor, and Prof. Arnab Sarkar for their insightful comments and encouragement. Their comments and suggestions helped me to widen my research from various perspectives.

I would like to express my heartful gratitude to the administration of IIT Guwahati and all faculty and staff of Dept. of Computer Science and Engineering for extending their co-operation in terms of technical and official support for the successful completion of my research work.

I am thankful to my friends Jiss J Nallikuzhy, Hrishikeshan, Vivek Lukose, Vijith, Sandeep, Haris, Sonu, Anoop, Sajith, Arun Mathew, Vishnu, Piyoosh, Dileep, Uma Narayan, Malu, Thomas, Mathew, Fahad, Suresh Babu, Ranjith, Anuj Budhkar, Vikavi, Arjun P, Naveen, Abhijith K V, Sooraj Chacko, Jith, Freddy, Hijas, Tom, Jorge, George Moses, Sudhi, Rezeem, Ansel Jose, Vivek Francis, Kiran Mukhathala, Priyadarshan, Sreejith, Akhil GV, Nikhil, Salama, Suvin, Christy, Abhjith, Faizal, Kiran, Nikhil V, Akhil, Shahabaz, Subhash, Manu, Rafi, Rishi shreedhar, Gokul, Aneez, Vyzakh, Burhan, Ashmil, Ranchal, Amarjith, Aswani, Arun, Albert, Adarsh, Riya, Gadha, Sreejith Muarlidharan, Sachin, Thahir, Adil, Hrishi, Shibili, Merlin, Caraline for sharing beautiful moments during my life in IIT Guwahati. I am grateful to all friends in the department especially Achyut mani tripathi, Pradeep sharma, Panthadeep, Dhantu, Nayantara, Ashish Mishra, Rakesh pandey, Aakansha for their unconditional help and support. You made my life at IIT Guwahati a memorable experience.

I am also grateful to the professors of malayali community Dr. A. Rajesh, Dr. Benny George K, Dr. Ganesh Natarajan, Dr. Ravi K, Dr. Vibin Ramakrishnan, Dr. Suresh Kartha, Dr. Tony Jacob, Dr. John Jose, Dr. John Thomas and their families. I cherished my moments with the football and swimming clubs of IIT Guwahati.

Most importantly, I thank for the love and patience of my family. I want to thank my parents, in-laws, brother Padmanabhan, sister-in-law Parvathy, brother-in-law Aswin for being a constant source of love, concern, support, and strength all these years.

Last but not least, to Aswathy, whose seamless encouragement, cheerful presence, and absolute love made my days bright and jubilant.

Contents

List of Figures

List of Algorithms

List of Tables

List of Acronyms

IC	*Integrated Circuit*
SoC	*System-on-Chip*
ATE	*Automatic Test Equipment*
CUT	*Circuit Under Test*
ATPG	*Automatic Test Pattern Generation*
DFT	*Design For Testability*
BIST	*Built-in Self-test*
EDA	*Electronic Design Automation*
SBST	*Software-based Self-testing*
ISA	*Instruction Set Architecture*
RTL	*Register Transfer Level*
ALU	*Arithmetic Logic Unit*
MUT	*Module Under Test*
CPU	*Central Processing Unit*
HDL	*Hardware Description Language*
VHDL	*Very-High-Speed-IC HDL*
DAG	*Directed Acyclic Graph*

GA	*Genetic Algorithm*
ES	*Evolutionary Strategies*
LFSR	*Linear-feedback Shift Register*
PC	*Program Counter*
RF	*Register File*
FC	*Fault Coverage*
OBS	*Observability*
TP	*Test Program*
FTP	*Fragment of Test Program*
DM	*Deadline Monotonic*
FDL	*Fault Detection Latency*

List of Symbols

μ	Size of initial population in a generation
λ	Number of newly created individuals in a generation
τ	Tournament size
P_i^j	j^{th} individual test solution of i^{th} generation
FC_i^j	Set of covered faults by P_i^j
FN_i^j	Set of newly covered faults by P_i^j
FCC_i	Set of all covered faults until the generation i.
t	Instruction template
t_i	Instance of instruction template t
c	Component of processor
O_i	i^{th} output value of a processor module
I_i	i^{th} input value of a processor module
Z	A genetic operator
Φ_Z	The activation probability of Z
Θ_Z	The number of activations of Z
Ψ_Z	The number of successful activations of Z
α	The coefficient of activation probability

ρ	Mutation strength for a generation
H_Ω	number of successful mutations for last Ω generations
EM	The number of consecutive elementary modifications for a mutation
m_i	A memory update after the execution of P_i^j
M_i^j	Set of all memory updates after the execution of P_i^j
p	Number of memory updates
R_i^j	Register contents after the execution of P_i^j
q	Number of register locations
O_i^j	Primary outputs after the execution of P_i^j
r	Number of primary outputs
OBS_i^j	Overall observability $\{M_i^j, R_i^j, O_i^j\}$ of P_i^j
P	A test program solution
P'	The test program after compacting P
G	A data dependency graph corresponding to the instructions of P
G_i	A connected component of G
u	Number of instructions in P
v	Number of connected components in G
FC_{G_i}	Set of faults covered by instructions of G_i
σ	Number of instructions in P after first stage of compaction
δ	Number of blocks in P
B_i	A block of instructions of P
FR_i	Set of reliant faults of B_i
I_{B_i}	Initial state of B_i

s	A random instruction
CRT	Ratio between execution times of optimized and original test codes
CRS	Ratio between sizes of optimized and original test codes
$cost$	Computation cost for compaction
τ_i	i^{th} task
Γ	$\Gamma = \{\tau_1, \tau_2, \tau_3, \ldots, \tau_n\}$; Set of n periodic real-time mission tasks
T_i	Periodic length of task τ_i
E_i	Execution time of task τ_i
D_i	Relative deadline of task τ_i (with respect to its arrival)
U_i	Utilization factor of task τ_i
U	Overall utilization of n tasks
$Rel(t)$	Reliability of system during a time interval $[0, t]$
β	Failure rate of system
k	Maximum number of faults in each task instance
C_s	Time consumed for saving a checkpoint
C_r	Time consumed for retrieving a checkpoint
$WCRT_i$	The worst-case response time of task τ_i
H	Instance of task τ_i
τ_t	A self-test task
S	Instance of self-test task τ_t
τ_{t_j}	j^{th} self-test sub-task
w_j	Execution window of τ_{t_j}
r_j	Pseudo-release time of τ_{t_j}
d_j	Pseudo-deadline of τ_{t_j}

Chapter 1

Introduction

1.1 Complexity of Modern Digital Circuits

The integrated circuits (ICs) manufactured today consists of millions of logic gates and memory cells. Very deep sub-micron technologies are used to implement these ICs [6]. System-on-Chip (SoC) design techniques [7,8] integrates every computer component and associated electronic systems into a single chip which encompasses many pre-designed and reusable units, called Intellectual Property (IP) cores. Today it is possible to integrate more than one billion transistors onto a single chip. This has enabled the implementation of complex functionalities in handheld gadgets and personal computers. However, handling such complexity is intensively burdensome. *As the electronic chip technology is fast-growing and complex, the quality and reliability of the manufactured chips have become hugely challenging.*

The challenges of handling this complexity are mostly related to the design and testing of the digital components of these chips [9]. Test cost holds a significant part in the total development cost of modern complex chips. Several well-researched disciplines have been employed for the efficient testing of large and complex chips. These include the utilization of various abstraction levels of circuit implementation, test methodologies for appropriate architectures, different test optimization methods, testing techniques with high reliability, etc. This research contributes mainly to the areas of high-quality testing methods and their optimization.

A post-silicon validation process [10] aims to validate the manufactured chips stage-by-stage so that no bugs may be left undetected. *Manufacturing testing* [11] and *on-line testing* [12] are two such stages of testing after the chip fabrication. Manufacturing test

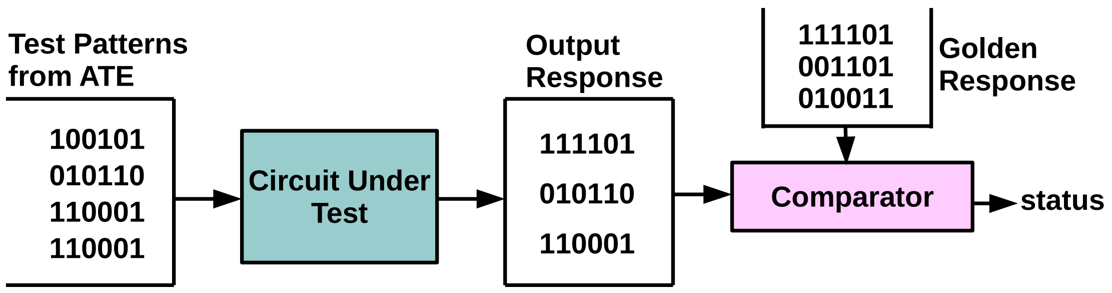


Figure 1.1: *Digital Testing Principle*

is performed after the chip is manufactured. If any fault is identified anywhere in the circuit after the manufacturing process, the whole chip is discarded and thrown away. Even after the chip is released into the market, the testing process has to be continued to make sure that the chip performs properly during its normal operation for safety-critical and industrial applications. This is called online testing or in-field testing.

However, the existing post-silicon validation techniques cannot cope with the complexity of modern ICs. Manufacturing testing of complex chips uses automatic test equipment (ATE) [9, 11, 13], which externally generates the test inputs, applies them on the circuit under test (CUT) and evaluates the test responses. For a high-frequency operating chip, a high-frequency test capability of ATE is required and the size of the physical memory of ATE should be large enough to store a large number of test patterns and responses. So, the external testing method which makes use of a high-frequency test capability of external ATE has become more expensive and less feasible because of the high test data volume and the longer test application time.

1.2 Digital Testing Principles

In traditional digital testing, ATE generates the binary patterns as input and applies those patterns to the CUT, as shown in Fig. 1.1. Following the application of test patterns, the output responses are collected and compared with a golden response, which is the expected output response. If the collected output is matching with the expected response, the final status of the comparison is recorded as true and the chip is declared to be non-faulty. Otherwise, the final status of the comparison is recorded as false and the chip is declared to be faulty. If the chip is found to be faulty, it is discarded and thrown away.

In the *digital testing* of ICs, *processor testing* has higher significance as compared with the testing of other SoC components. If any processor functionality is damaged, every active application will be affected by that fault. But if other components, such as memory, has some defect on any of its cell, it may not affect the execution of all programs since every program may not use that specific memory cell. So, processor core defects are more important and crucial and this factor motivates us to focus on the domain of test generation for the processor cores.

1.3 Processor Testing and its Challenges

Effective post-silicon validation has been highly challenging for complex embedded processors. Several structural and functional testing approaches, such as automatic test pattern generation (ATPG), design for testability (DFT), built-in self-test (BIST), etc., have been diversely used in the processor testing domain based on the characteristics of the processor to be tested. Fundamentally, post-silicon validation methods apply either 1) structural testing or 2) functional testing techniques.

Structural test development methods cultivate the test patterns using the gate-level descriptions of the processor. Automatic test pattern generation (ATPG) techniques [14,15] exploit the structural details of the processor to apply high-coverage test patterns using an ATE. In structural testing, a comprehensive fault model like gate-level stuck-at fault model is used for synthesizing a smaller set of high-coverage input test patterns. Although test execution is shorter, the development cost of structural test patterns is higher as compared with the high-level functional test synthesis.

Functional testing approaches attempt to validate the correctness of every functionality of the CUT. In the case of processors, functional testing refers to the development of test patterns that checks only the functions of the processor instruction set architecture (ISA). In most of the traditional functional testing approaches, pseudorandom test patterns, which require long test sequences, were used as operands leading to excessively large test application time. However, the development cost of functional testing is lesser.

In *deterministic* test generation methods [16], pre-developed test patterns that functionally test the processor components are cultivated using ATPG tools. Scan-based *design for testability (DFT)* approaches [17, 18] reduce the test generation complexity of ATPG methods by accessing the internal memory elements of the processor circuit.

Structural testing methods could use electronic design automation (EDA) tool for generating ATPG test sequences with the help of structured DFT techniques like scan chain method.

As purely random test pattern generation is burdensome, *pseudorandom* test methods [19, 20] are used to generate repeated test input sequence that yields an acceptable fault coverage only with the help of a large number of test patterns. To validate smaller processor circuits, test engineer could manually develop effective test patterns [21, 22]. But in the case of complex circuits, automated methods like *formal verification*, *evolutionary approaches*, etc, are popularly used for test generation. In *formal methods* [23–25], test solutions are generated using powerful formal tools, such as bounded model checking (BMC) or satisfiability-based (SAT) methods whereas *evolutionary technique* [26, 27] makes use of genetic operators to generate diversified test routines with high coverage.

Identification of all physical faults (delay faults, transition faults, etc.) has been ever-challenging because the test patterns must be applied at the operational frequencies of processors, which are extremely high. This at-speed testing feature is very difficult to achieve with external tester technologies as the ATE frequencies could not reach up to the processor frequencies [28]. Later, the concept of *self-testing* [16, 22, 29–32] was introduced which reduces yield loss with the help of actual at-speed testing while the overall test cost of the processor is lesser. The concept of processor self-testing will be discussed in the next section.

1.4 External Testing to Self-testing: A Paradigm Shift

Self-testing could be *hardware-based self-testing* [29, 30] or *software-based self-testing (SBST)* [16,22,31,32]. In hardware-based self-testing, also termed as *built-in self-testing (BIST)*, a dedicated hardware module is attached to the processor for testing. This module generates the test patterns and applies them to the module under test (MUT). Eventually, the responses are collected and delivered to another circuit, which does the response analysis. Deterministic testing or pseudorandom testing could be used for the test generation in BIST approach. However, the on-chip test generation is easier with the help of pseudorandom testing as it produces the test patterns using smaller additional

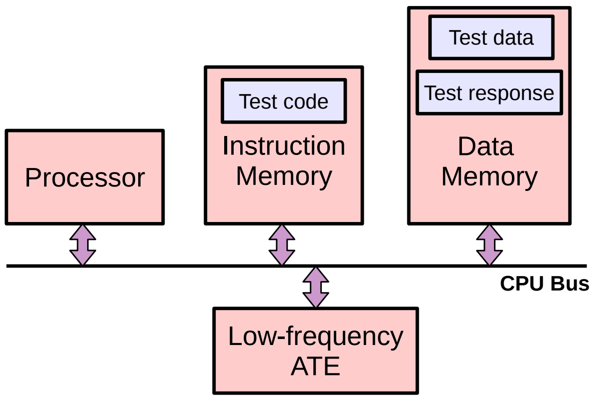


Figure 1.2: *SBST Procedure*

hardware circuits.

Apart from the at-speed testing feature, the transfer of the processor testing approach from external ATE-based approach to an internal BIST mechanism provided significant advantages. One of them is the reduction of test cost earned by the use of self-test methodologies for processor testing. Self-testing reduces yield loss with the help of actual at-speed testing while the overall test cost of the processor is lesser. Also, the use of self-testing drives down the design cycle and therefore, a better time-to-market is achieved. The Intellectual Property (IP) protection is also improved when compared with that of the scan-based external testing techniques. An apparent drawback of hardware-based self-testing approach is the hardware overhead spent for the additional testing circuit. Also, during the hardware-based self-testing, power consumption is more than that of the normal operational mode of the chip. To solve this, SBST methodologies [16, 22, 31, 32] were introduced.

SBST methodologies have cultivated software-based test codes that were applied on the processors as test routines. These test codes are sequences of instructions with selected operands that could validate the processor functionality. The SBST approaches are non-intrusive because the chip design or hardware does not necessitate any modification for testing. These light-weight test codes are uploaded into the memory locations and the responses are downloaded and compared for the fault identification. Furthermore, SBST does not require any extra hardware which leads to a reduced test cost and zero chip area penalty [33]. For these reasons, SBST is exceedingly used for embedded processor testing.

The advantages of SBST [28] are:

- *At-speed testing*: The test program application is performed at the speed of the actual frequency of the processor. Therefore, all physical faults can be detected and the test quality is improved.

- *Non-intrusive*: SBST does not add any extra hardware or DFT modification overhead to the existing circuitry. It executes normally like all other programs and consumes the same average power during the testing phase.

1.4.1 SBST of Processors

SBST procedure for a processor is as illustrated in Fig. 1.2. The self-test program, which is a sequence of assembly code instructions, is generated using a test pattern synthesis technique [28] and is downloaded into the instruction memory of the chip for testing. The self-test data and its corresponding responses are stored in the data memory. The test program is executed to generate a test response, which is compacted and stored in the data memory. The response collection and response compaction of SBST require negligible hardware intrusion. Now, the response is analyzed to produce a pass or fail indication, based on which we proceed for further actions on the system. In SBST, the self-test programs are downloaded to the tester memory from a low-cost, low-speed ATE and the test responses uploaded back to the ATE for the response analysis [34].

Recently, several advanced manual SBST approaches [33, 35–37] have been introduced for SBST code generation. To automate SBST synthesis, test engineers exploited hierarchical structural testing methods like formal verification [25, 38, 39] which discovers input sequences that violate user specifications for the fault detection of each processor module. Also, functional feedback-based methods like evolutionary approaches are prevalently used in the domain of SBST automation. As structural methods are computationally prohibitive for SBST automation, genetic algorithm-based evolutionary strategies for SBST automation are used.

1.4.1.1 Phases of SBST Procedure

The overall SBST procedure, as shown in Fig. 1.3, includes the following five phases: A) *Information extraction*, B) *Processor component classification and test prioritization*, C)

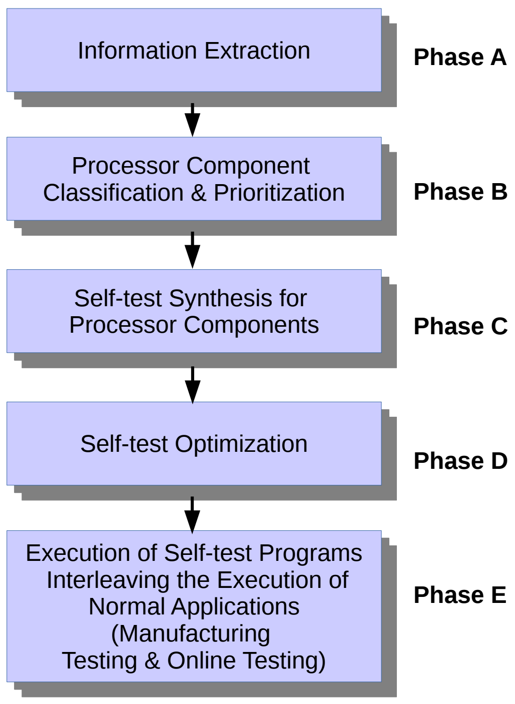

Information Extraction
Phase A
Processor Component Classification & Prioritization
Phase B
Self-test Synthesis for Processor Components
Phase C
Self-test Optimization
Phase D
Execution of Self-test Programs Interleaving the Execution of Normal Applications (Manufacturing Testing & Online Testing)
Phase E

Figure 1.3: *Phases of SBST Procedure*

Test program synthesis, D) *Test program optimization*, and E) *Test program application*. In phase A, the ISA information and Register Transfer Level (RTL) information of the processor are used to identify the components of the processor and the operations of the components, etc. With the help of the information from phase A, the processor components are classified in phase B as functional, control, and hidden components.

Functional components could be either computational functional modules, such as Arithmetic Logic Unit (ALU), adder, multiplier, etc. or storage functional modules, such as accumulator, register file, etc. Major control components, such as control unit, generate the control signals for the functional components of the processor, thereby controlling the data flow and instruction flow. The hidden components, such as pipelining, increase the throughput of instruction execution but are functionally invisible.

These components are prioritized based on accessibility and testability to enhance the test development phase (Phase C). The computational functional components have higher testability than any other components because their operations are directly associated with instruction execution, i.e., they are functionally more visible. So, these components are assigned a higher priority for test development. In phase C, self-test codes are synthesized for each component as a module under test (MUT), based on the priority. The self-test synthesis is initially conducted for high-priority components because adequate fault coverage has to be achieved as quickly as possible.

For the self-test synthesis in phase C, we gather the ISA information and component information from phase A and the test priority information of components from phase B. However, it is apparent that the test software is an additional process that competes with user processes for system resources, such as central processing unit (CPU) cycles and memory. Therefore, on-line test program execution is considered to be an overhead to overall system performance regarding memory area, power consumption and execution cycles. Also, large size and longer execution time of SBST programs for complex processors make manufacturing test difficult. So, in phase D, the self-test codes (synthesized in phase C) are subjected to an optimization procedure in terms of memory footprint, execution time, power consumption, test size, test execution time, etc. In the last phase (phase E), we apply the optimized self-test codes from phase D during the processor manufacturing stage and/or operational stage. The optimized test code with the highest fault coverage is used to test every instance of the same processor chip.

1.5 Principal Scheme of SBST Automation

The principal scheme for automatic self-test synthesis, where an evolutionary core develops a test solution of optimum fault coverage given a processor model, fault models, an instruction library, and an external fault simulator, is shown in Fig. 1.4. Here, an individual test solution refers to a self-test code, which is a valid sequence of instructions.

The processor model describes the RTL model of the processor to be tested in any hardware description language (HDL) like Very-High-Speed-IC HDL (VHDL) [40], either in synthesizable or simulatable form. Here, the RTL model is subjected to module partitioning which is realized by breaking down the RTL design into several functional units and testing them separately. So, each processor module corresponds to a single hardware block and therefore, there are as many modules as the number of valid digital blocks in the processor model.

The instruction library is created using the ISA information with every instruction of the ISA having their entries in it. Each entry is called a macro, which is an instruction with randomly selected operands. In this approach, an individual solution, which is a test program, is generated using the evolutionary process and is represented by a directed

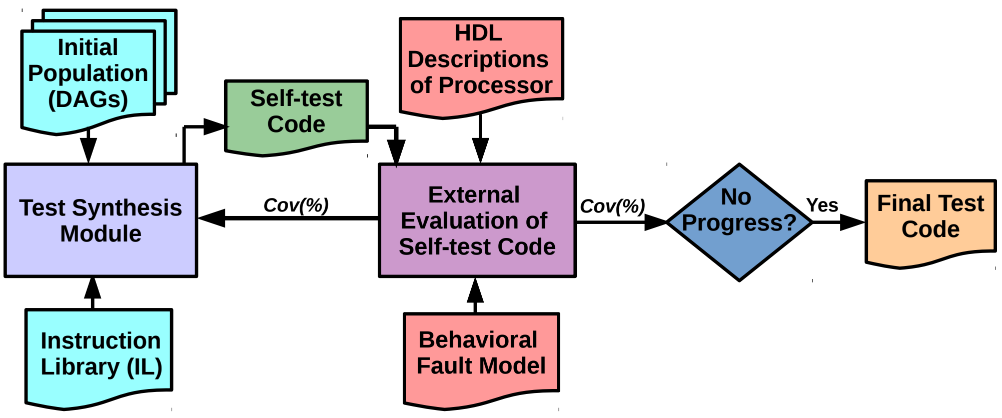


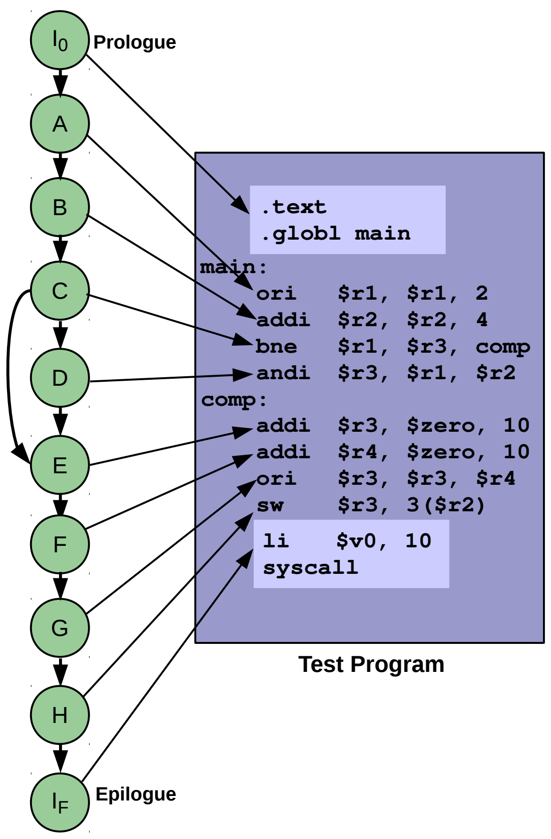
Figure 1.4: *Principal SBST Automation Scheme*

acyclic graph (DAG). In the initial population of test solutions, a group of empty DAGs is collected. The evolutionary core synthesizes the fragments of assembly code (macros) taken from the instruction library to generate a new population of DAGs.

An external evaluator measures the quality of these self-test program solutions with the help of several faulty processor models. The contents of the observable locations of the processor, such as memory locations, register locations, and primary outputs, are extracted as test responses for fault evaluation.

With the help of these necessary inputs, the evolutionary core develops a genetic algorithm (GA) based test code synthesis. In this method, a parent population of test program solutions is modified in each generation using mutation, crossover, and selection operators. Mutation operator explores the search space for diverse test solutions and crossover operator mixes the genetic data of two parent solutions to develop the offspring solutions. The selection operator selects the fittest of them in order to generate the next generation of population. This process goes on until there is no progress in the generations of test solutions.

MicroGP (μGP) [41], i.e., genetic programming for microprocessors, is an evolutionary approach generally designed for automatically developing assembly code programs for microprocessors. In [2], G. Squillero exploits μGP for the fine-tuned automated synthesis of self-test assembly programs for processors. Unlike the earlier evolutionary approaches, G. Squillero [2] focuses on self-adaptation techniques, such as activation probability and mutation strength of the genetic operators, which tune the search process internally.



```
.text
.globl main
main:
    ori   $r1, $r1, 2
    addi  $r2, $r2, 4
    bne   $r1, $r3, comp
    andi  $r3, $r1, $r2
comp:
    addi  $r3, $zero, 10
    addi  $r4, $zero, 10
    ori   $r3, $r3, $r4
    sw    $r3, 3($r2)
    li    $v0, 10
    syscall
```

Figure 1.5: *Representation of an Intermediate Test Program of μGP Test Synthesis*

1.5.1 Test Code Preparation in MicroGP Methods

Evolutionary strategies (ES) are employed to automate the test synthesis using a DAG method. A test solution comprises a sequence of macros and is represented by a DAG, as shown in the example in Fig. 1.5. A DAG node has pointers to a macro element in the instruction library and the set of parameters as shown in Fig. 1.6. Epilogue and prologue nodes (I_0 and I_F) are the initial and final empty nodes. A $(\mu + \lambda)$ strategy of ES with an initial generation of μ DAG test solutions is carried out to develop efficient assembly programs that could validate the processor components. In every generation, new λ offsprings are created using a 1-point crossover and the following mutation operators:

- Add node: A node is added to the DAG. The new node could be inserted anywhere after the prologue node and before the epilogue node in the DAG. If a new macro is inserted between a branch and its target instruction, the target address in the branch instruction must be updated.

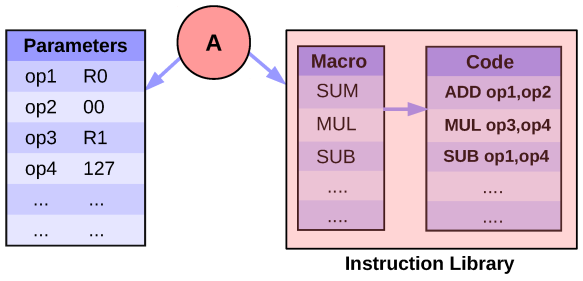


Figure 1.6: *A DAG Node with a Pointer to Instruction Library and a Pointer to Set of Parameters*

- Remove node: A node is removed from the DAG. Any node could be removed from the DAG except the prologue and epilogue nodes. If the removed instruction is in the region between a branch and its target instructions, the target address in the branch instruction must be updated.

- Modify node: A node is modified in the DAG. If the modified node is a branch instruction, its new target instruction must be within the DAG.

Among the $\mu + \lambda$ individuals of a generation, μ fittest offsprings are selected by the tournament selection operator of tournament size τ. If this tournament size τ is very large, fittest individuals will be selected always which would reduce the diversity of the offspring population. In each generation, these individual test programs are evaluated using behavioral fault model to select the fittest population. An efficient selection of the objective function would help in carrying out the progressive development of the genetic population in consecutive generations and the test solutions evolve through the generations until an optimal solution is achieved.

1.6 Test Quality Evaluation

1.6.1 Gate-level Fault Models

Gate-level fault models [42–46] like stuck-at fault models, bridging fault models, path delay fault models, or transistor fault models are used for test quality evaluation in structural testing approaches. Combinational faults, such as stuck-at faults could easily be detected using smaller test sets. So, a low-cost, low-speed ATE would be enough for detecting the combinational faults. On the other side, the testing of sequential faults, such as delay faults, must be conducted at the operational speed of the processor because these faults occur due to timing malfunctions.

The EDA tools [47] for combinational fault models are mature enough to develop smaller test sets with shorter test application time and shorter fault simulation time. As the EDA tools for sequential fault models cannot develop efficient test sets, fault simulation time and test generation time are high for the test sets that detect the sequential faults. However, the performance of EDA tools for sequential fault models is improving gradually [28].

1.6.2 Behavioral Fault Models

Recently, it has been difficult to test large complex processors using the gate-level fault models like stuck-at fault models, bridging fault models, delay fault models, or transistor fault models. A huge number of gate-level faults will have to be modeled for evaluating the test programs for the processors with a complex gate-level netlist, which is prohibitively expensive. So, it is necessary to adopt behavioral fault models. In most of the IC designs, gate-level structural descriptions are not available to generate the conventional fault models. So, various high-level fault models are generated using behavioral level fault modeling, where different faults such as the *stuck-at-0* and *stuck-at-1* faults are injected into the RTL descriptions. For example, in the input stuck-at fault model, the input is stuck to 0 or 1 for a *bit* or *bit_vector* type signal and stuck to $false$ or $true$ for a Boolean type signal in the RTL statements.

This approach has a higher level of abstraction compared to the gate-level fault modeling because the fault models are associated with the behavioral level descriptions [5,48]. However, most of the hardware faults are covered if the behavioral fault coverage is good enough because of the robust correlation (above 95%) between the behavioral faults and the physical faults as demonstrated in [49–51].

1.6.3 External Fault Simulation

The external evaluator of the principal SBST automation scheme, shown in Fig. 1.4, receives the new offspring test solutions from the evolutionary core and returns the fault coverage using the behavioral fault models illustrated in Subsection 1.6.2. Also, the progress of the evolutionary process is updated with the help of the external evaluator

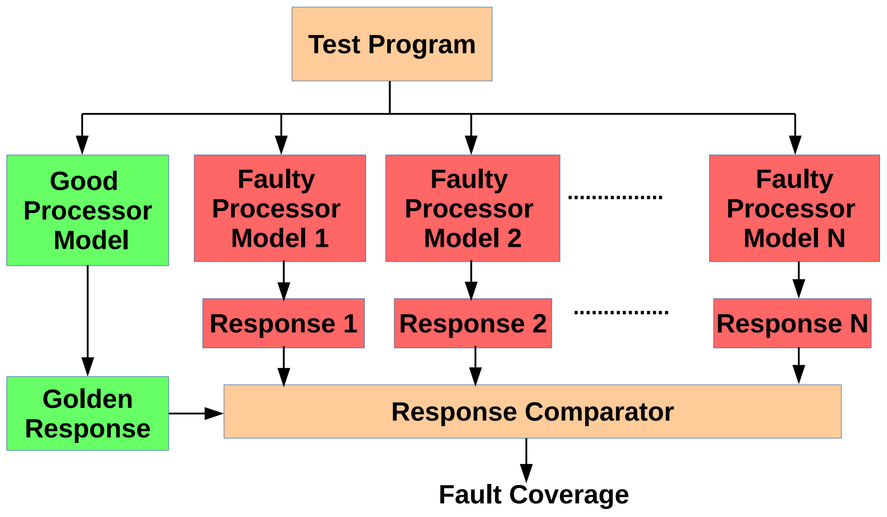


Figure 1.7: *Test Program Evaluation Overview*

to terminate the test synthesis judiciously.

As suggested in traditional SBST code synthesis [2,41,52,53], the automation of test programs for processors is performed with the help of an evolutionary approach that select the fittest test solutions using genetic operators. Each test solution is evaluated in terms of fault coverage for the selection of fittest programs. Faulty and non-faulty processor models are simulated for measuring the fault coverage of each test program. The responses of these simulations would comprise the contents of the observable locations (registers, memory updates, primary output, etc.). Further, the contents of the observable destinations of faulty processor models are compared with the expected response generated by the good processor model to realize hardware fault detection. In the SBST approach, the test quality is evaluated by the fault coverage and fault list extracted from the simulation responses, i.e., the contents of the observable locations. Intuitively, the fault coverage and fault list of an SBST test program are completely associated with the observability of the processor modules.

As shown in Fig. 1.7, the processor is simulated for a good reference model and N faulty models. Each faulty processor model is inserted with a single fault which logically represents the physical faults. All of these faulty models must be simulated independently to collect the test responses. Later, these responses are compared with the golden responses of the good processor model to assess the fault coverage. For example, if the responses of 75% of faulty processor models are observed to be different from the golden response upon the execution of a test program, the fault coverage of the test program is 75%. As each single fault simulation is time-consuming, fault simulation of all the faulty processor models would consume an enormous amount of time. Also, the response collection of test programs, where every observable point must be recorded in each cycle, is computationally intensive.

1.7 Motivation and Objectives

The test code synthesis (Phase C), test code optimization (Phase D), and test code application (Phase E) procedures are the most significant phases, in terms of computational cost and time consumption, of SBST procedure shown in Fig. 1.3. In the test synthesis phase, assembly-level test programs are automatically generated using an evolutionary method. However, due to the intrinsic randomness of the evolutionary process, the test synthesizer would not be able to search for the test solutions in the desired manner. Even though the synthesized test solutions detect the easily traceable faults, some of the hard-to-detect faults might go undetected. This leads to low coverage of faults, thereby delivering insufficient test quality, which may lead to system crash. So, to guarantee detection of hard-to-detect faults, the evolutionary test synthesizer must conduct a comprehensive search for exceptional test solutions.

Modern complex processors would need large test programs to test them. A large test program would lead to undesirably large test download time, and a longer test program would lead to longer test execution time. Existing test compaction methods attempt to compress the test programs, but are computationally intensive. To reduce the overall cost of SBST execution, a low-cost automated optimization method that yields high test compaction must be developed.

During online testing, the test codes are applied on the processor in a regular interval during its operational stage along with the normal applications. A high-reliability, low-cost fault detection and recovery technique is necessary for the processor with safety-critical applications executed in extreme operating conditions. Several works [16, 31, 32, 34, 54–62] have been proposed for the enhancement of online reliability enhancement of processors. During self-test execution, the intermittent faults, which are momentary in nature, could not be detected by large test programs with large test periods. These faults could be identified by smaller test programs with smaller test period. But smaller test programs would likely to have lesser test quality in terms of fault coverage. To deal with the challenges in the detection and recovery for the temporary faults, a fault-tolerant schedule with optimal test program size must be developed.

To summarize, the motivation for this thesis is "*to develop a low-cost, high quality, and faster SBST process for modern processors with the help of improved performance in the execution of SBST code synthesis, optimization, and application phases.*"

Now, we explain four major challenges in the SBST code synthesis, application, and optimization phases (Phases C, D, and E of the SBST procedure shown in Fig. 1.3).

1. **Detection of Hard-to-test Faults and its Preservation**
   Genetic algorithm-based approaches are widely accepted methods to search for high-quality self-test programs for processors. But the overall coverage of these test solutions was insufficient because the corner cases, which are the hard-to-detect faults, were never taken care of. During the automatic test program generation, the intermediate test programs comprising instructions which detect the hard faults must be sustained to constitute a final test solution of improved coverage. Further, these approaches could not guarantee the test quality because the fault evaluation metrics, such as the statement coverage, were not well-correlated with the physical faults. Generally, gate-level fault models are closely correlated with physical faults. So, we must adopt a high-level fault model which is closely correlated with the gate-level fault models and therefore, would be closely correlated with the real physical faults.

   When sequences of instructions which could possibly detect the corner cases are selected, a higher preference must be assigned to them to help them survive through the future generations. As these sequences may never get reproduced, it is necessary to conserve these test programs, which could contribute to the final optimal solution with higher coverage. So, our objective is to enhance the GA-based self-test synthesis by preserving the instruction sequences which detect the hard-to-detect faults. Also, the adopted fault models must have a high correlation with the gate-level fault models. Otherwise, a test program that detects every fault modeled by the adopted fault model may not be able to detect sufficient real physical faults.

2. **A Faster Test Program Synthesis**

   The overall test synthesis time consumed by the previous μGP approaches was inconsiderably huge. Further, if the evolutionary module comprehensively searches for the hard-to-detect faults, the test synthesis would be further delayed. This could be avoided by reducing the number of costly external fault evaluations for redundant test programs. So, our objective is to accelerate the test synthesis procedure along with the detection of hard-to-detect faults by reusing the existing test programs of identical characteristics. The test codes, developed using the evolutionary process, that produce similar fault simulation responses must be identified and reused for a faster fault evaluation.

3. **Effective Test Program Compaction and Test Execution Time Reduction**

   The automated SBST synthesis [2, 41] could be employed for developing either a monolithic test code for the whole processor circuit or a set of test codes for the processor modules. In both cases, the test code must have a huge number of selected instructions to sensitize all the testable processor functionalities. However, a larger SBST code leads to performance overhead due to higher test code download time [63], and a longer execution time escalates the test application time. Also, in the online testing of safety-critical embedded processors, smaller test code execution is an extremely crucial requirement. So, test code optimization, in terms of execution time and size, has been crucial for the effectiveness of SBST testing of processors [64].

   Some of the recent techniques in test code optimization [4, 65] have demonstrated redundant instruction elimination methods to maximize the test compaction. However, the number of fault simulations required to identify the redundant instructions of test code is proportional to the test code size. So, a huge number of fault simulations have to be conducted for the optimization of larger test code. As the number of fault simulation required for test compaction increases, the time consumed for test compaction, which is the overall computational cost, also increases. Instruction restoration and instruction removal techniques [4, 65] guarantee the elimination of redundant instructions in terms of coverage with a reasonable compaction rate. But the computational cost (CC) is huge for these techniques since efficient test compaction needs large number of fault simulations. So, our objective is to develop a test optimization technique with adequate compaction rate and reasonable computational cost.

4. **Online Detection of Intermittent Faults**

   During the operational stage of the processor, many temporary faults occur due to extreme operating conditions. These faults, called intermittent faults, appear momentarily, lasts for some time, and may turn into permanent faults. Generally, the intermittent faults are activated by processor wear out and excessive fluctuations in temperature and voltage. The fault detection latency is the time gap between the fault occurrence and its detection. If the fault detection latency is higher, the reliability, which is the probability that a system does not deteriorate during a specific time interval, would be lesser. A self-test task must provide an optimal fault detection latency to detect the intermittent faults and must be schedulable along with a set of periodic real-time tasks.

   Larger SBST codes would detect most of the faults but reliability will be lesser due to high fault detection latency. If an intermittent fault occurs just after a large test period, fault detection latency will be higher, which may cause system errors. Smaller SBST self-test codes with smaller fault detection latency realize rapid detection and recovery of intermittent faults. As the intermittent faults occur irregularly at the same location, smaller self-test codes must be regularly executed with a short test period to efficiently trace them. But these minimal test programs could have less reliability due to low fault coverage.

   To deal with this trade-off, optimal and reliable set of fragments must be discovered with significant self-test quality (coverage) and minimal fault detection latency. So, our objective is to design a high-reliability online fault detection model that could test low-cost, real-time embedded processors for the intermittent faults.

1.8 Contributions

The major contributions of the thesis are described below. These works address the challenges to meet the objectives explained in Subsection 1.6.3.

1. **Greedy-based Evolutionary Test Synthesis to Detect the Hard-to-detect Faults**
   We have proposed a greedy component integrated into the traditional evolutionary framework to develop test solutions that detect the hard-to-detect faults of processors and achieve improved fault coverage. Using this approach, we have synthesized the test programs that could detect the hard-to-detect exceptional faults besides the easily excitable ordinary faults. Although this approach yielded improved fault coverage, the convergence rate of the test synthesis was low. A faster convergence was achieved when the test programs were synthesized using the instructions with selection priorities, which is a testability-based ranking feature.

2. **An Accelerated Software-based Self-test Synthesis (Rapid SBST) for Processor Testing**
   While the evolutionary process of test synthesis progresses, it is highly likely that the evolutionary core develops individual solutions with similarities in fault simulation results. If the instruction sequences of two test individuals have similar functionalities, the fault simulation results could be reused to reduce the test synthesis time. So, this reusability enables a rapid convergence towards the development of high-quality test solutions.

   Following the test program execution, the processor faults are identified by comparing the contents of the observable locations on the processor. If the values stored in these observable locations following the simulation of an offspring solution are same as that of one of its parent solutions, the set of faults that they could identify will almost be same; i.e., equally-observable test solutions will probably have equal fault coverage. In that case, a re-simulation of the offspring solution could be avoided by reusing the identified fault list and the fault coverage of the parent solution. In this work, a faster SBST synthesis of processor cores is employed using an accelerated greedy-based evolutionary method (RSBST), where the test programs that could detect the hard-to-test faults are developed rapidly.

3. **SBST Compaction Techniques using Data Dependency Graphs and Instruction Restoration Techniques**

   The challenges of test optimization for SBST codes has become critical because large processors demand test codes with a large number of instructions to completely test their functionalities. In this approach, we have introduced an enhanced test compaction approach that works in two stages. In the first preprocessing stage, we remove the independent instructions using a data dependency graph representation of the test program. The approach used in the second stage is an enhancement of state of art ***A1xx*** algorithm [4]. In this stage, we reduce the computational complexity by avoiding the fault simulation of previously restored instructions. To evaluate these restored instructions, high-level logic simulations are conducted, which reduces the computational cost considerably.

4. **SBST Fragments for Intermittent Fault Detection of Processors**

   In this study, we address a major challenge for the online self-test execution in a real-time scenario. The tradeoff between test quality and fault detection latency has been crucial in online periodic self-testing, where the self-test tasks are executed interleaving the execution of real-time mission tasks. To address this challenge, we have developed an online self-testing approach, where the processor is subjected to fragmented testing for tracing the intermittent faults. So, the test code, synthesized for manufacturing testing, is replaced with many small and efficient test codes with adequate fault coverage. These test code fragments are examined and are periodically applied in small execution windows during a test period between the normal tasks of the processor. We design a methodology to generate the set of smaller test codes with adequate coverage for this purpose during the evolutionary test program synthesis itself. Also, we evaluate all the test programs suitably.

   From a testing point of view, these small but adequately efficient fragments, developed with the help of enhanced reliability analysis, replace the bigger test code to reduce the fault detection latency. In particular, we must set the shortest possible test execution window so that all intermittent faults could be detected. If any erroneous behavior of the processor is diagnosed, the mission task is rolled back to the previous checkpoint and re-executed. However, our method incurs inconsiderable overall fault coverage drop, compared to the coverage of the complete SBST test code.

1.9 Organization of the Thesis

- **Chapter 2**: *Literature Survey*
  Existing Processor Testing Techniques, SBST synthesis, optimization, and application techniques are elaborated in this chapter.

- **Chapter 3**: *Greedy Cover-based Evolutionary SBST Synthesis*
  This chapter presents a greedy-based evolutionary SBST synthesis approach for the detection of hard-to-detect faults. Besides, we make use of the testability of each instruction to evaluate the selection probabilities of instructions. The major objective of this work is to enhance the test quality by synthesizing high-coverage test programs.

- **Chapter 4**: *Rapid SBST (RSBST) Program Synthesis*
  This chapter deals with a test program reusability-based approach to speed up the traditional test synthesis while maintaining adequate fault coverage. This technique gets rid of the computational cost consumed by redundant test quality evaluations during the evolutionary test synthesis. This accelerated test development is useful when it comes to the synthesis of large test programs for modern processors.

- **Chapter 5**: *Automated Low-cost Compaction of SBST Programs*
  This chapter discusses a two-stage test program compaction technique which are enhancements of existing instruction removal and restoration techniques. The primary objective of this work was to optimize SBST test programs with less effort. We have validated the effectiveness of our approach using a comparison study with the existing compaction techniques on a large number of test program samples.

- **Chapter 6**: *Application of Fragments of SBST Programs for Online Testing*

In this chapter, we present an online periodic testing approach that maximizes the reliability of real-time systems by detecting the intermittent faults. To realize this, we apply smaller but high-quality test programs periodically so that the intermittent faults could be traced. These smaller test programs are selected during the evolutionary test synthesis to increase the operational reliability of processors. In this study, we have methodically dealt with the objective of identifying the optimal test periods for the execution of these efficient test code fragments during the processor operational stage.

- **Chapter 7**: *Conclusion and Future Perspectives*
  This chapter concludes the thesis with the discussions on future perspectives on SBST testing techniques.

Chapter 2

Literature Survey

In this chapter, we discuss the existing pioneering works related to processor testing in chronological order. Further, we elaborate the recent advancements in the three most significant phases of the SBST procedure, which are SBST code synthesis, optimization, and application (phases C, D, and E), as shown in Fig. 1.3 of previous chapter. The section 2.2.1 and section 2.2.2 of this chapter describe the background for the improvements in the functionalities of SBST code synthesis, section 2.2.3 describes the background of SBST code optimization, and section 2.2.4 describes the background of SBST code application.

2.1 Chronology of Processor Testing Methods

As the processor technology is complex and expanding, the reliability of embedded processors is highly critical during the phases of chip manufacturing, and operational stages. Before the design is committed for fabrication, hardware/software co-Verification [66, 67] verifies system software or firmware runs correctly on the hardware design. This process conducts early integration of software or firmware with hardware, before any chips or boards are physically available. In co-verification, firmware validation relies on the interacting hardware components which are usually not available until the late design stages. This co-validation is generally addressed through co-simulating C/C++ based firmware code and HDL hardware models (including SystemC). The co-verification tools like aglei from Eagle Design Automation and Seamless CVE from Mentor Graphics specifically target at solving the hardware/software integration problem for embedded systems.

In system-level verification [66, 68–70], the manufactured design (chip) is tested for all functional correctness in a lab setup. This process is conducted using the real chip assembled on a test board or a reference board along with every other system components. In system-level verification, several test cases are run on netlists to check whether the design behaves functionally correct or not. Test cases are run for various corner cases with random with constrained inputs to achieve maximum test/fault coverage. This verification process is generally considered very critical as part of design life cycle as any serious design, which are not discovered before tape-out, can eventually increase the overall cost of design process. Presently, SBST programs are developed and used only in BIST and online/concurrent testing of processors. These methods can be used to develop effective SBST soultions for both hardware/software co-validation and system-level verification.

Production/manufacturing testing screens manufactured chips for physical faults or defects before the chip is released into the market. This testing procedure must be conducted in the actual speed of the processor hardware which is in GHz. Traditionally, *ATPG methods* [14, 15] were used to generate test patterns using fault sensitization techniques that were applied on the processor using an ATE. In this approach, a sequence of test vectors is selected from every possible test inputs so that these test vectors can detect most of the processor faults. However, the selection of efficient test vectors remains a challenging task for large sequential circuits.

Some of the previous *random testing* techniques makes use of random test inputs where a huge volume of test inputs are applied for fault detection [9]. The storage and application of these purely random sequences would consume a huge amount of time and achieve lesser coverage. In *deterministic testing approach* [16], functional modules of the processor like ALU, Shifter, Multiplier, etc., can be tested using pre-computed test vectors developed by ATPG tools. These test vectors are applied as processor instructions to these modules to yield high fault coverage. Although this method attains high coverage with smaller test sets, gate-level details are necessary for test generation.

Later, several *DFT techniques* [17, 18] were introduced which exploit design-level alterations to achieve high quality test patterns. Scan-based DFT approaches make use of scan-in and scan-out operations. During the scan-in process, logic values are applied on the memory elements, such as latches, flip-flops and during the scan-out process, the content of each memory element is extracted out of the scan chain. Although scan-based

DFT makes the circuit nodes easily accessible, this method experiences an excessive test application time and hardware overhead.

In *pseudorandom testing* [19, 20], random patterns are generated repeatedly based on an efficient seed value and these patterns are applied as instruction sequences. A standard linear-feedback shift register (LFSR) can be used to generate the pseudorandom patterns for BIST of processors. The benefit of pseudorandom testing is that gate-level details are not required for test generation. In BIST, the golden responses are stored in a read-only memory (ROM) as a signature. The fault coverage is evaluated using an output response comparator which compares the compacted CUT response with the golden signature stored in ROM. However, this method needs exceedingly long test pattern sequences to achieve a level of acceptable fault coverage.

Manual test generation techniques [21, 22] select and use the processor instructions corresponding to the operations of every processor module to validate its functionality. The operands of each selected instruction are chosen randomly. But the manual selection of instructions and operands that test functionally large processors is laborious for the test engineers. So, automated test generation techniques like formal verification methods, evolutionary techniques, etc., were introduced. The functional ATPG patterns are automatically cultivated using formal verification methods [23–25]. Evolutionary techniques [26, 27] applies genetic operators, such as mutation, crossover, selection, etc., on a population of test solutions to develop better and diverse offspring test solutions. In this method, test solutions with highest fault coverage have more probability to get selected to the next generation which heuristically leads to the development of optimal test patterns.

2.2 Advanced SBST Techniques

The advancements in manual test program generation approaches [33, 35–37] have substantially contributed in developing effective SBST programs. In these works, complex functional test patterns are developed for testing pipelined processors with multithreading, dynamic instruction execution, and multicores. Nonetheless, the cost of test pattern development is a tradeoff because the assembly programmer has to devise complicated, high-coverage test programs for larger processors manually.

Psarakis et al. [36] discuss the taxonomy of various *structural* and *functional* approaches for SBST program development. Structural testing approaches use structural information, such as RTL descriptions, for test generation and functional approaches use functional information, such as ISA, for test generation.

The hierarchical *structural* approaches, where test programs are generated module-by-module, can be automated using powerful formal verification engines, such as BMC or SAT methods [25, 38, 39]. Zhang et al. [25] leverage on existing BMC tools in order to generate software-based self-testing programs from a global extended finite state machine (EFSM) model of the processor under test. Riefert et al. [38], describe an ATPG framework targeting stuck-at faults based on BMC. This framework allows the user to flexibly specify the requirements of SBST test programs in the considered scenario. Finally, they demonstrate how a set of properly chosen requirements can be used to generate test programs matching these constraints. However, formal methods would be computationally prohibitive for processors with complex sequential circuits.

Other prevalent *structural* SBST program generation approaches are ATPG, pseudo-random test generation, and deterministic testing approaches. ATPG techniques [14, 15] develop the test stimuli with the help of the gate-level netlist of the processor. Pseudorandom pattern generators [19, 20] could be used to generate random but efficient patterns with a low area overhead. Deterministic testing approaches [16, 33, 35] generate test set corresponding to the operations and functionalities of each processor module.

Constraint-based *structural* test generation methods [31, 71] consider the gate-level or the structural details of the MUT whereas the remaining processor modules are considered at a higher level. Now, these methods extract the constraints imposed by the execution of the instructions on the MUT and develop efficient test patterns for the MUT based on these extracted constraints. Finally, these module-level patterns are translated into SBST programs.

There are two types of *functional* testing techniques: 1) Randomizer and 2) feedback-based techniques. Code randomizer techniques [72–74] target the functional faults using a random sequence generator. Evolutionary technique [26, 27, 75] is a feedback-based *functional* testing approach, where test programs are evolved automatically using genetic algorithmic strategies. This population-based optimizer refines a set of test program solutions iteratively to develop high-quality test programs. In our research works, we have chosen evolutionary techniques [26, 27, 75] to develop SBST programs because this approach naturally develops smaller but efficient test programs with lesser computation cost.

2.2.1 SBST Code Synthesis

To realize at-speed testing of embedded processors, SBST methodologies [16,22,31,32,76] have cultivated software-based test codes to be applied on the processors as test routines. The SBST test codes are sequences of instructions with selected operands that could validate the processor functionality. The SBST approaches are non-intrusive because the chip design does not necessitate any modification for testing. These light-weight test codes are uploaded from a low-frequency ATE into the instruction memory, executed to collect the responses in data memory, and the responses are downloaded into the ATE and compared for the fault identification. Furthermore, SBST does not require any extra hardware which leads to a reduced test cost and zero chip area penalty [33]. For these reasons, SBST is exceedingly used for processor testing.

A different diagnosis-oriented method was proposed by Bernardi et al. in [77]. A set of existing SBST test sets are optimized using a *sifting* procedure and subsequently, the diagnostic ability of these programs are improved using an evolutionary tool. The faults are grouped into equivalence classes based on the diagnostic equivalences and 75.6% of gate-level faults are identified.

There has been several techniques to automate the synthesis of SBST programs [25, 27, 38, 39, 41, 75, 78–90]. One of the initial efforts in the automation of test synthesis is the *micro*GP (μGP) approach by Corno et al. [41], which is an evolutionary technique to automatically generate assembly code test programs for target microprocessors. In this approach, the fault coverage is improved with the help of the feedback from the test evaluation framework. In μGP evolutionary strategy, new individual assembly program solutions are synthesized in each generation which could be used as SBST test programs. These new test programs are combined with existing parent test program solutions for breeding a new generation of the population of test program solutions.

Later, G. Squillero improved the μGP technique [2] towards a flexible, modular, and adaptive architecture which helps the microprocessor engineers to conduct an enhanced search for the test solutions. These earlier μGP approaches [2, 41] employ statement coverage of HDL descriptions as the code coverage metric for the test program evaluation. But statement coverage is a high-level evaluation metric which does not yield the accuracy of the gate-level fault models because a majority of the gate-level faults are not detected by a test program with high statement coverage.

To enhance the accuracy of test evaluation, the μGP approach has been extended with a composite and intricate code coverage metric in [52, 53]. Along with statement coverage, toggle, branch, expression, and condition coverages are also calculated. Sanchez et al. [53] assimilates and reuses the existing test programs to simplify the test generation process and has integrated the aspects of the identification of clone codes. In this method, the statement and branch coverages are high whereas the toggle, condition, and expression coverages are inadequate (less than 80% on the average).

D. Gizopoulos et al. developed an SBST automation method [35] which focused on the testing of pipeline components which also enhanced the testing of functional components. This method identifies the blocks of the pipeline unit which have good testability and improves the coverage of the pipeline logic. With the help of a transition delay fault model, 89% fault coverage is achieved for pipelined RISC processors. The applicability of this method is shown only for the pipeline unit and not generalized for all components of the processor.

To further improve the test quality, J. Hudec and E. Gramatova introduced a test evaluation framework with an objective function that could evaluate the test solutions effectively [91]. A weighted average of statement, branch, toggle, and conditional coverages is used to evaluate the test quality. Here, the test patterns are generated competently with lesser test generation time and good fault coverage. But this method cannot assure the test quality because the correlation between these code coverage metrics (of HDL descriptions) and the gate-level faults is low.

To evaluate the synthesized test programs, we have adopted behavioral fault models [5, 48]. This approach is designed with the models of failures on the HDL constructs to achieve almost *reliable* fault representations. A fault model is *reliable* if there exists a close correlation with the physical defects of the circuit. The gate-level fault models are well-correlated with the physical defects and thus, are *reliable*.

Several studies have been conducted [49–51] on the correlation between the behavioral fault models and gate-level fault models on general circuits. In [49], Anton Karputkin and Jaan Raik analyzed the experimental results on ITC'99 benchmarks and concluded that the test programs that are developed based on a set of behavioral fault models could detect an average of 86% of the gate-level faults. According to Karunaratne et al., the test benches for behavioral faults could approximately cover 90% of the gate-level faults of the sequential cores of an SoC [51]. These studies asserted that for general circuits, the behavioral fault models are well-correlated with the gate-level fault models. So, we can say behavioral fault models are almost *reliable* for generic benchmark circuits.

Previous μGP techniques, proposed in [2, 41, 52, 53], could not guarantee the test quality because the hard-to-test faults were left undetected and the code coverage-based fault evaluation metrics do not hold strict correlation with gate-level fault models. These methods could not realize adequate gate-level fault coverage of more than 90% for the synthesized test programs.

So, in general, SBST code synthesis procedure should include the following aspects:

- The synthesized test programs of existing μGP approaches could not uncover the hard-to-test faults because the intermediate test programs that detect the exceptional faults may get dropped during the evolutionary process. So, the test quality is not ensured by these approaches. Although some of the intermediate test programs detect faults which are extreme corner cases, they may not survive to the subsequent generations of test program population. It is observed that these solutions are preserved only if the objective function of the evolutionary approach deals with the coverage of the freshly and exceptionally identified faults. So, we must choose a modified objective function to greedily preserve the test solutions with instruction sequences that could detect the uncovered faults during the evolutionary test synthesis.

- High-level behavioral fault models, which have a close correlation with the physical faults, could be adopted to evaluate the test solutions of each generation. The previous studies [49–51] have shown that a test code, which has high behavioral fault coverage, could detect most of the gate-level faults in the case of generic benchmark circuits. Regarding processor circuits, the correlation between gate-level fault models and behavioral-level fault models could be analyzed and verified with the help of a comparison study between the effectiveness of these two fault models.

2.2.2 Faster SBST Code Synthesis

The experiments conducted for the previous μGP techniques [2,41,52,53] converge in tens of hours. Although these test synthesis approaches are reasonably fast, an acceptable level of fault coverage is not attained. It has been a challenge for test engineers to cultivate high coverage test programs within a limited amount of time.

In [92], Kranitis et al. introduced a hybrid SBST (H-SBST) methodology for low-cost development of high-quality test programs. H-SBST has three phases. In the first phase, the MUTs are identified. During the second phase, MUTs are classified as functional and control components and are ranked based on their testability. Later, a combined test development strategy of structural SBST methodologies and random test program generation (RTPG) is discussed in the third phase. After applying the structural testing for the MUTs, RTPG is applied as the supplementary step to improve the fault coverage. In a case study of test development for OpenRISC 1200 processor, incompetent code coverage of 92.5% is achieved although consuming a low test execution time.

On the other hand, the recent SBST automation approaches [90, 91] have improved in terms of coverage but rely on time-consuming test generation techniques. Lu et al. [90] have developed a hybrid test program that combines deterministic test programs for each module and randomly developed instruction sequences for a high-performance self-testing of pipeline cores. The experiments on ARMv4 and miniMIPS processors demonstrate an improved gate-level fault coverage of more than 98% for this hybrid test generation method, which is close to the coverage achieved for a full scan chain-based technique. But the consideration of exhaustive gate-level fault models leads to a longer test synthesis for real-world, complex, pipelined processors.

Although test evaluation using behavioral fault models is simpler than that using gate-level fault models (as mentioned in Subsection 1.6.2 of Chapter 1), many of the behavioral-level fault simulations could be avoided exploiting the similarities between the test solutions. So, towards achieving a faster SBST synthesis, a smarter test evaluation method which may reuse the responses of identical test solutions could be proposed.

2.2.3 SBST Code Optimization

Modern processors comprising large and complex circuitries necessitate the development of test codes with a huge number of instructions. However, a larger SBST code leads to performance overhead due to higher test code download time [63]. Also, a longer execution time escalates the test application time. So, test code optimization, in terms of execution time and size, has been crucial for the effectiveness of SBST testing of processors [64]. Some of the recent techniques in test code optimization [4, 65] have demonstrated redundant instruction elimination methods to maximize the test compaction. However, in these methods, the number of fault simulations required to identify the redundant instructions of test code is proportional to the test code size. So, the optimization of a larger test code would consume a larger number of fault simulations, thereby increasing the computational cost.

The test development technique proposed in [63] demonstrates how low-cost test codes are developed for RISC processor cores. Initially, this method classifies the processor components into functional, control, and hidden components to prioritize them for test development. Thereafter, compact loops of test instructions that excite the component operations are developed for each component. In [64], D.Gizopoulos proposes four low-cost, online test development approaches which aim at small memory footprint, small execution time, and low power consumption. To reduce the CPU execution time of test codes, these techniques minimize the instruction and data memory interaction.

In the above methods [63, 64], the test compaction was carried out as one of the steps during the test development phase, i.e., the dedicated effort for test compaction was insignificant. As a consequence, the amount of shortened test code size or reduced test execution time due to the compaction procedure was low. However, the state of the art techniques of SBST compaction, such as [4, 65, 93, 94], employ a dedicated test compaction module to conduct a comprehensive, instruction-by-instruction test compaction procedure.

In [4], two test code compaction methods were introduced. The first method makes use of a random instruction removal algorithm, called as **A0** test compaction algorithm, where redundant instructions are greedily searched and removed from the original test code. In **A0**, random instruction is selected in each step. If any instruction does not contribute towards the overall fault coverage, it is permanently removed from the original test code. However, the remaining test code must execute and terminate properly without exceptions.

In the second test code compaction method proposed in [4], a restoration-based algorithm, called as **A1xx** test compaction algorithm, is employed. In **A1xx**, the authors construct the optimized test code by removing blocks of instructions of the original test code and subsequently, restoring them to identify the redundant instructions of these blocks. Initially, the test code is split into blocks of instructions with equal size. Now, each block is selected and removed from the test program one at a time. Following the removal of a block, its instructions are restored one by one until all the faults that may get undetected due to the block removal are detected. So, the redundant instructions of each group are not restored, thereby constructing a compacted self-test code.

One of the critical issue of **A0** and **A1xx** compaction techniques [4] is the occurrences of Length Dependent Faults (LDF). An LDF fault could be detected only if the test code has at least a specific number (n) instructions. If LDFs are present, test code length and execution time cannot be reduced beyond n using the conventional compaction techniques. So, the technique proposed by [94] extends the **A0** compaction algorithm [4] by inserting a NOP instruction on the removal of a redundant instruction. This placement of NOP instructions allows the test code to maintain the fault coverage by detecting the LDF faults, which eventually helps in the progress of test compaction process.

An advanced test compaction technique is introduced by Touati et al. [93] which discovers the smallest set of functional SBST codes that yield high fault coverage with reduced test time. To realize this, the redundant test codes are identified and removed from the set of test codes by comparing the list of covered faults. Also, different sequences of test code executions are investigated to find out the minimal test execution time for the optimized set of test codes.

The ARES (*Automated Reordering for Efficient SBST*) approach in [65] attempts to reduce the test execution time with maintaining the fault coverage. This method operates in two stages. In the first stage, the self-test code is partitioned into non-overlapping groups in all possible sequences, for a target number of groups. Among all grouping arrangements, the best grouping solution is selected with the help of a test length based quality metric evaluated using high-level logic simulations. In the second stage, these groups are reordered and fault simulated to discover the group ordering with minimum test execution time.

The required computational cost for a thorough, instruction by instruction test compaction would be very high. Instruction restoration and instruction removal techniques [4] guarantee the elimination of redundant instructions in terms of coverage with a reasonable compaction rate. Since efficient test compaction needs a large number of fault simulations and single fault simulation takes a significant amount of time, the overall computation cost of these test compaction techniques is large.

A low-cost, instruction-wise test optimization technique that yields adequate test compaction could be introduced by enhancing the existing **A1xx** technique [4]. In this possible modification, a high-level logic simulation could be used to replace the preceding blocks of a restoring block of instructions with a smaller set of instructions that yields similar fault coverage. Also, every group of independent instructions could be investigated to identify and remove the redundant groups which do not contribute towards the overall fault coverage.

2.2.4 SBST Code Application

SBST codes are applied on the processor for *i*) manufacturing testing and *ii*) online testing. In manufacturing testing, the test codes are applied after the processor chip is manufactured and before it is shipped into the market. In online testing, the test codes are applied periodically on the processor during its operational stage, interleaving the normal applications. *Online operational reliability of processors in an extreme operating environment with safety-critical applications necessitates the incorporation of low-cost fault detection and recovery techniques.* Recent advancements [16, 31, 32, 34, 54–62] in the reliability enhancement of online testing has proposed several techniques for efficient fault diagnosis.

During the operational phase, the erroneous behavior of the processor hardware is largely attributed to the existence of intermittent faults [95] in an extreme operating environment. These faults are temporary in nature, appear in irregular intervals at the same location, cause errors in bursts, and may eventually turn into permanent faults. Intermittent faults occur due to the processor wear out and may also get activated by excessive fluctuations in temperature, frequency, or voltage.

In online testing, also termed as in-field testing, processor circuitry, while it is operational, is tested for temporary and permanent hardware faults. Online processor testing methods could be categorized into *concurrent* and *non-concurrent* approaches.

Concurrent testing [96] detects the operational faults with the help of various methods for hardware redundancy, software redundancy, and time redundancy. In these methods, additional instances of a normal application, which is also termed as *mission task*, concurrently execute using extra hardware, software, or time. *Non-concurrent* testing employs test patterns periodically between the task execution [97]. To reduce redundancy cost and power consumption of online testing, *concurrent* testing techniques were replaced with *non-concurrent*, periodic testing methods.

The test patterns must be executed in the processor frequency in order to trace the intermittent faults, which are instantaneous in nature. If not, some of them might go undetected. To carry out a dynamic at-speed processor testing [28], where the test patterns are applied in the actual operating speed of the processor, SBST methodologies have been proposed [16,31,32,34,54] and are widely applied in the online testing domain. SBST approaches are non-intrusive as the circuit design does not require any modification for testing purpose. Besides, test execution has no hardware overhead which reduces the test cost and leads to zero chip area penalty [33]. For the above reasons, SBST methods are used for the detection of intermittent faults in online periodic testing.

The advances on online testing [97–100] discuss the periodic testing techniques with SBST programs being periodically executed to identify the faults in the processor components. Further, the recent progress in the fault-tolerance domain of the intermittent faults demonstrate efficient detection, diagnosis, and recovery approaches [95, 101].

A probabilistic cost function for SBST programs is introduced in [97] using the reliability analysis of intermittent faults in pipelined processors. This cost function helps in reducing the SBST execution cost during the online periodic testing. Also, this technique enhances the methodology proposed in [99] with the addition of advanced SBST programs for the pipeline and exception logic components to increase the fault coverage up to 96.67%.

Xenoulis et al. [100] proposes an SBST program development scheme for processors with single and double-precision floating point units. The test pattern generation (TPG) schemes for floating point units are selected corresponding to the target requirements [64] to develop low-cost self-test programs. However, the above cost-prohibitive self-test generation schemes [97, 99, 100] could not guarantee efficient test program utilization and schedulability.

D. Gizopoulos [98] has formulated a method for the effective utilization of self-test codes for the online periodic testing of medium cost, real-time embedded processors. With the help of rate-monotonic scheduling (RMS) mechanism, this technique could achieve an improved self-test utilization or self-test quality in addition to the schedulability realized for the real-time mission tasks. However, D. Gizopoulos [98] have focused on the enhancement of the self-test quality disregarding the instruction set characteristics of the self-test programs. Also, the schedulability of self-test programs and a larger set of mission tasks on multi-core processors would necessitate advanced scheduling approaches.

A Stochastic Activity Networks (SAN)-based recovery, proposed in [95], deals with the fault tolerance of multicore processors against intermittent faults. Based on the failure rate and the defective processor location, SAN devises a recovery action for the performance enhancement of an intermittent fault-sensitive processor.

Rashid et al. [101] evaluate the impact of intermittent faults on processors using a SPEC2006 benchmark and Alpha Sim, which is a microarchitectural fault injection tool. In their work, they injected 3000 faults to the considered microarchitectural components and subsequently observed that the majority of the intermittent faults lead to system crash. In other words, 67% of the intermittent faults were non-benign and 79% of these non-benign faults caused a system crash (i.e., a hardware trap). The goal of their work was to study the effectiveness of software-based techniques in the intermittent fault diagnosis and recovery. Although the software-based technique takes away the performance overhead of a separate test process, huge memory overhead is incurred as the processor state is to be logged continuously.

As the intermittent faults could occur irregularly at the same location, the self-test codes must be regularly executed with a short test period to efficiently trace them. D. Gizopoulos [98] suggested that the self-test quality could be improved when the self-test tasks are executed with larger execution time and enhanced self-test utilization. But the proposed techniques in [98] will increase the self-test period to achieve maximum utilization. If the self-test period is increased, fault detection latency also would increase, and subsequently, some of the instantaneous intermittent faults may be left undetected. If an intermittent fault occurs just after a large test period, fault detection latency will be higher, which may cause system errors. The tradeoff between test utilization and fault detection latency in [98] could be dealt only if efficient, small chunks of SBST codes are executed frequently between the mission tasks, i.e., smaller, coverage-efficient test programs are intermittently executed during a self-test period to reduce the fault detection latency.

Shorter, reliable SBST test code fragments must be discovered and executed intermittently in a self-test period to immediately detect the intermittent faults with minimal fault detection latency. But smaller test codes might have lesser fault coverage, which could leave some of the intermittent faults undetected. So, the test fragment synthesis must consider both fault detection latency and test quality (fault coverage) in developing reliable fragments to be applied in appropriate execution windows between the execution of the mission tasks.

The tradeoff between test quality and fault detection latency could be improved using the application of test program fragments with high reliability. Test programs with smaller execution time and adequate fault coverage could be selected in order to detect the intermittent faults. These test codes must be identified with the help of reliability evaluations to get executed periodically during the operational stage of the processor with shorter time periods.

2.3 Summary

In this chapter, we have discussed the existing techniques on the synthesis, optimization, and application of SBST programs. Following the component classification, these techniques attempt to synthesize, optimize, and apply SBST programs efficiently. But the complexity and large size of modern processor circuits necessitate the enhancement of following aspects regarding the existing SBST technique.

- Evolutionary synthesis of high-coverage SBST codes by preserving test solutions with instruction sequences that could detect the hard-to-test faults.

- Rapid evolutionary synthesis of SBST codes with adequate test quality reusing the simulation responses of equally-observable test solutions.

- Compaction of the SBST codes with a reasonable tradeoff between the amount of compaction and the computational cost for compaction.

- Reliability enhancement of online testing by executing optimally-sized test code fragments that has adequate coverage and also detect intermittent faults.

In the next chapter, we discuss a greedy cover-based evolutionary method for SBST synthesis of high-quality test programs.

Chapter 3

Greedy Cover-based Evolutionary SBST Synthesis

In this approach, the effectiveness of automation of self-test program synthesis (Phase C of the overall SBST procedure shown in Fig. 1.3 in Chapter 1) using evolutionary approach is improved by identifying the difficult-to-test faults with the help of greedy coverage and testability based ranking techniques. The self-adaptive μGP approach [2] could not develop test solutions that detect these exceptional faults. Further, the test quality was also not ensured by this approach as the test evaluation using code coverage metric of HDL descriptions is not well-correlated with the gate-level faults. So, we improve the test quality by uncovering the difficult-to-test faults of the processor using an effective test evaluation method. Further, the behavioral fault models are used in this study which are well-correlated with the gate-level faults.

The principal scheme for automatic self-test synthesis is shown in Fig. 3.1. The processor model describes the processor to be tested in any hardware description language like VHDL. The instruction library is created with the entries of every instruction. High-level fault models are generated using behavioral level fault modeling [5,48], where various failures of VHDL constructs are considered as faults. The greedy based evolutionary test generator develops test programs using a greedy cover method integrated into the GA-based code synthesis. This is the main component of the proposed scheme and will be discussed in Section 3.2. In the next subsection, we discuss the preliminaries and the working principle of the proposed greedy based approach.

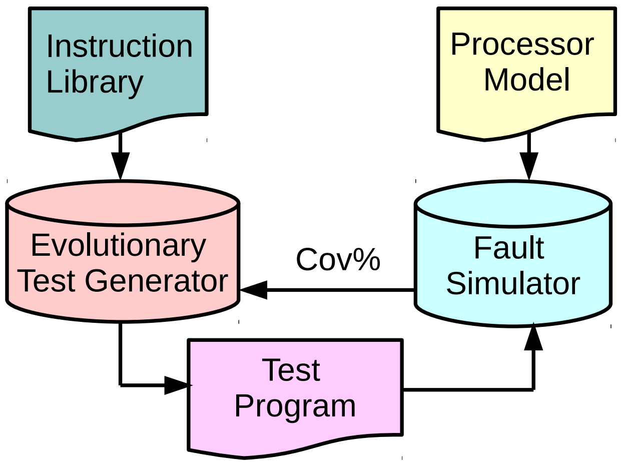


Figure 3.1: *Principal SBST Automation Scheme*

```
lw $1, $2($3)

•  Parameter Constant   R1 R2
•  Parameter Integer    -128 127
•  Parameter Constant   R1 R2
```

Figure 3.2: *A Sample Macro*

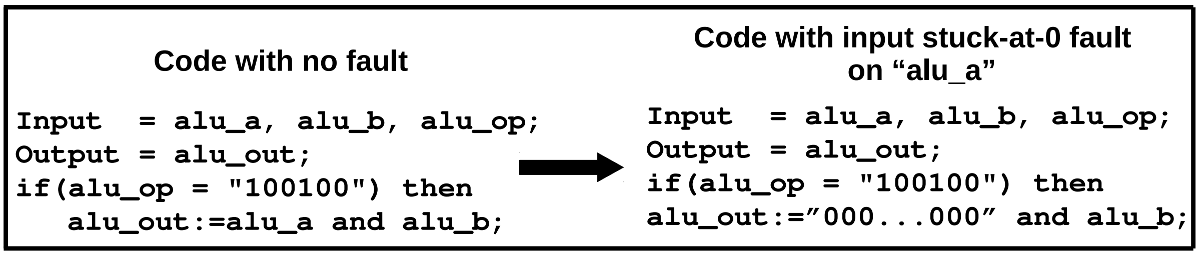
3.1 Preliminaries and Working Principle

The fragments of assembly code (macros) are taken from the instruction library to generate SBST test programs for the processor specified by the processor model in Fig. 3.1. A sample macro for the load instruction is shown in Fig. 3.2. Here, the load instruction is encoded between two registers and a 16-bit constant. The first and third parameters ($1 and $3) denote the two registers, each of them chosen from R1 and R2, respectively. The second parameter ($2) is the 16-bit constant, which may have a value between -128 and 127.

Due to the *reliability* of the gate-level fault models, the behavioral fault models are also guaranteed to be reasonably *reliable* [49–51]. But the correlation studies have been performed only for general benchmark circuits. This motivated us to adopt the behavioral fault models for our scheme and in Section 3.3.2, we demonstrate the close correlation between the behavioral faults and gate-level faults for the general instruction set processors. The behavioral fault representations proposed by Chen [5] are shown in Table 3.1. They listed the most common 10 behavioral fault models.

Table 3.1: *Behavioral Fault Models [5]*

Fault model	**Failure Type**
Input stuck-at fault	Primary input signal
Output stuck-at fault	Primary output signal
If stuck then fault	Else portion of an if construct
If stuck else fault	If then portion of an if construct
Elsif stuck then fault	Subsequent else portion of an if construct
Elsif stuck else fault	Elsif portion of an if construct
Assignment statement fault	Assignment of new values to signals
Dead clause fault	A Selected WHEN clause in case statement
Micro-operation fault	Micro-operations
Local stuck data fault	A signal object in a local expression



Code with no fault

```
Input  = alu_a, alu_b, alu_op;
Output = alu_out;
if(alu_op = "100100") then
   alu_out:=alu_a and alu_b;
```

Code with input stuck-at-0 fault on "alu_a"

```
Input  = alu_a, alu_b, alu_op;
Output = alu_out;
if(alu_op = "100100") then
alu_out:="000...000" and alu_b;
```

Figure 3.3: *Input Stuck-at-0 Fault in VHDL*

An example of a behavioral input stuck-at fault in the logical *and* operation statement of an ALU module is shown in Fig. 3.3. In these descriptions, an *and* operation is performed between two input 32-bit operands, which are *alu_a* and *alu_b*, to achieve the output *alu_out*. A stuck-at-0 fault in the 32-bit *alu_a* input signal transforms the *and* operation to: "*alu_out* := "0000 . . . 0000" *and alu_b*". As a result, an all-zero value is assigned to the output *alu_out*, regardless of the value of *alu_b*. This fault could be detected by the test program in Fig. 3.4 because in a non-faulty design, the operands R1 and R2 hold non-zero values which lead to an expected non-zero output value for the *alu_out* signal. If the stuck-at-0 fault in the *alu_a* signal exists, the output value for the *alu_out* signal becomes zero. In a normal case, the expected output is non-zero and thus, the fault is detected. Likewise, a stuck-at-1 fault in *alu_a* also could be detected

```
li   R1,1              // Loads R1
li   R2,1              // Loads R2
and  R3,R1,R2          // AND Operation
sw   R3,offset3(R4)    // Stores the result
```

Figure 3.4: *Test Program That Detects the Input Stuck-at-0 shown in Fig. 3.3*

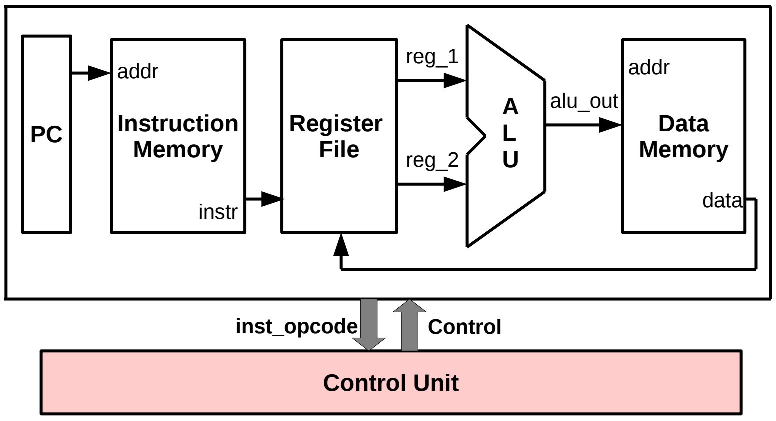


Figure 3.5: *A Simple RISC Processor Architecture [1]*

with the selected instructions and operand values.

Now we discuss the severity of the diagnosis of these behavioral faults. For this, the faults are classified into easy-to-detect and hard-to-detect faults. In Fig. 3.5, a MIPS processor model with a program counter, registers, control unit, and functional components/modules such as ALU are shown. Generally, the faults in the functional components which are directly associated with the instructions in the ISA are easily detectable by the majority of the instruction sequences. For example, the input stuck-at-0 fault shown in Fig. 3.3 is an easy-to-detect fault because this fault could be detected by many combinations of instruction sequences with *and* operation, such as the sequence shown in Fig. 3.4. The μGP [2] technique could uncover these easy-to-detect faults comprehensively whereas the hard-to-detect faults or the corner case faults were completely unidentified.

It is relatively challenging to identify the faults in the control unit and other non-functional components that belong to the control path of the processor. The failures in generating certain control signals could not be identified effectively because these signals are not regularly observable. If the collected test response, comprising the contents of response locations such as general purpose registers and memory, infrequently recognizes

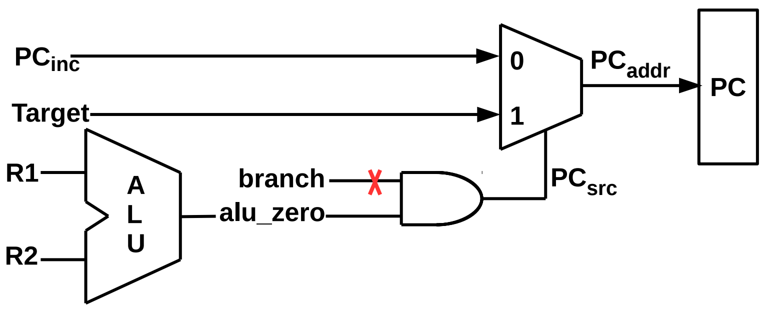


Figure 3.6: *MIPS Branch Instruction Execution [1]*

any deviations due to the existence of signal faults, dedicated test programs would have to be developed to take care of those faults.

Some of the controller faults are hard-to-detect, such as an assignment statement fault, where a signal is stuck to zero or one in a single VHDL statement in the control signal "*branch*". The propagation of *branch* signal towards any of the destination registers depends on the activation of the status signal *alu_zero* as shown in Fig. 3.6. The control unit activates the *branch* signal when a branch instruction, e.g., "*beq* $R1$, $R2$, $Target$", executes. The operands $R1$ and $R2$ are subjected to comparison by ALU, and if the result of the comparison is true, *alu_zero* also gets activated. If both the *branch* and *alu_zero* signals are triggered, the control goes to the instruction in the *target* address. Otherwise, the PC is incremented (PC_{inc}) to select the next instruction in the instruction memory. Here, any assignment statement fault in the *branch* signal is propagated ahead only if the value of the *alu_zero* signal is 1, which is carried out when $R1$ and $R2$ have equal values. In this case, the fault detection turns out to be difficult because of the dependence of *branch* signal on the *alu_zero* signal.

It may be noted that these operands (R1 and R2), most likely, would hold different values due to the randomness in the selection of operand values in optimization algorithms. To load these operands with equal values, appropriate *load* instructions must be executed before the *branch* instruction as shown in Fig. 3.7. Here, the operands $R1$ and $R2$ hold equal values $imm1$. So, the comparison result of the *beq* instruction becomes true, and the faults on the *branch* signal would propagate towards the observable locations. But the evolutionary core of the traditional μGP approach [2] may not generate test programs with such desired sequences of instructions frequently. So, these sequential dependencies of instructions and the operand constraints make the statement

```
            lui   R1,imm1
            lui   R2,imm1
            beq   R1,R2,Target
            addi  R1,R1,1
Target :    sw    R1,offset(R4)
```

Figure 3.7: *Instruction Sequence to Detect Branch Signal Corner Case*

assignment faults on *branch* signals undetectable.

Only a few sequences or combinations of opcode and operands could identify those faults which are less likely to get propagated to any of the response locations. During the test optimization process, such sequences hardly occur in the final test solutions achieved. Even if these sequences occur in any of the intermediate test programs, they may not survive to the future generations as the evolutionary process tends to preserve those with the sequences of instructions that can identify a large number of functional faults.

To summarize, the problem being addressed in this work is "*during the automatic test program generation, the intermediate test programs comprising instructions which detect the hard faults must be sustained to constitute a final test solution of improved coverage*". In the next section, we discuss how our proposed greedy approach would synthesize the test program solutions that detect the hard-to-test faults.

3.2 Evolutionary Approach for Test Program Synthesis

In this approach, a greedy fault coverage method is used to develop the test program solutions which reside in the uncovered and unexplored search spaces and also encompasses the instruction sequences which could cover the exceptional faults. Although these test programs cover a diverse set of faults, in some cases, they may hold a low fault coverage and therefore, do not get selected for the later generations.

When sequences of instructions which could possibly detect the corner cases, as illustrated in Fig. 3.7, are selected to generate the test program in our evolutionary approach, higher preference is assigned to them to help them survive through the future generations. As these sequences may never get reproduced, it is necessary to conserve

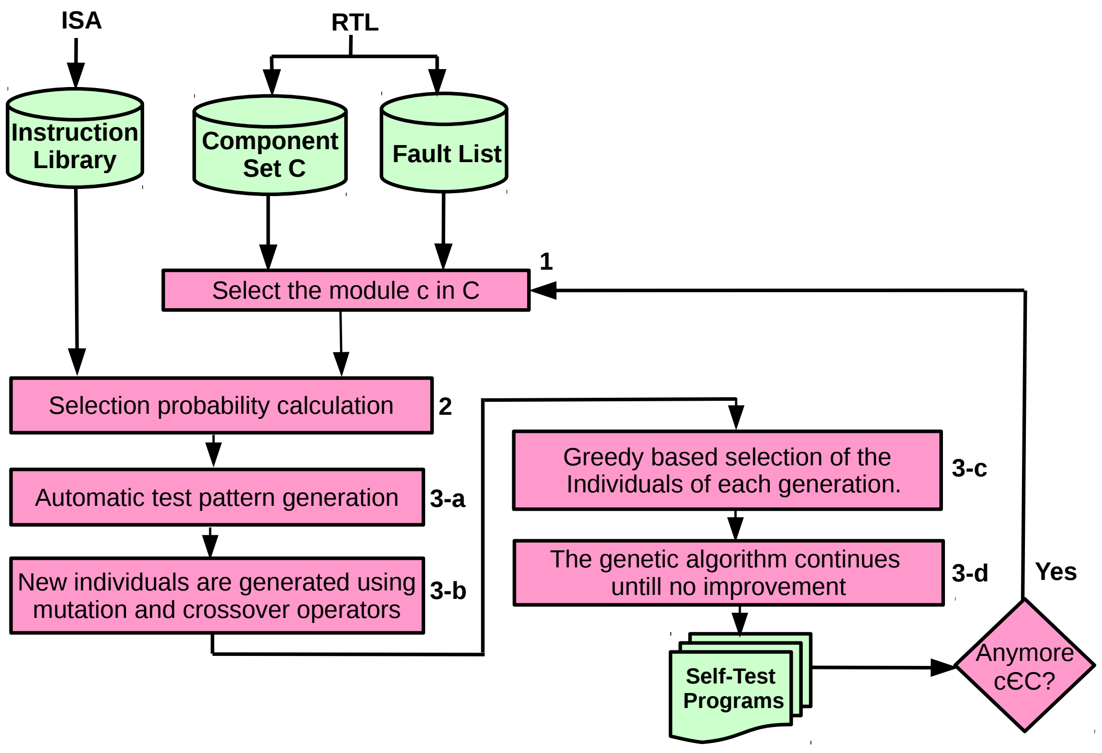

ISA
RTL
Instruction Library
Component Set C
Fault List
Select the module c in C
1
Selection probability calculation
2
Automatic test pattern generation
3-a
New individuals are generated using mutation and crossover operators
3-b
Greedy based selection of the Individuals of each generation.
3-c
The genetic algorithm continues untill no improvement
3-d
Yes
Self-Test Programs
Anymore c€C?

Figure 3.8: *Proposed Automation Method for Test Pattern Generation*

these test programs, which could contribute to the final optimal solution with higher coverage. So, in order to preserve these instruction sequences, a greedy mechanism is established and adjoined to the GA-based self-test synthesis.

As shown in Fig. 3.8, the proposed automation is performed in 3 steps. Initially, an instruction library, a component list, and a fault list are developed in the preprocessing phase. Using the instruction library, we generate a set of instruction templates for each of the macros as a preprocessing stage for the automated test synthesis. In the first step, the component with the highest number of faults is selected. Based on the testability values of each template for the module under test (MUT), the selection probabilities of each instruction are calculated (Step 2); it will be discussed in Section 3.2.1. Step 3 discusses the automated generation of test programs for the MUT using the proposed greedy technique.

To automate the test generation, evolutionary strategies (ES), which apply genetic operators to a population of individual solutions that iteratively search for better solutions, are employed using a directed acyclic graph (DAG) method. Here, a DAG is used to represent an individual solution, which is an assembly program, where the nodes of the DAG, as shown in Fig. 3.9, has pointers towards macro in the instruction library and its parameters. The node can be a sequential instruction or branch instruction as shown in Fig. 3.10. Epilogue and prologue nodes are the initial and last nodes, which are the empty default nodes of the DAG.

With the help of this graph representation, optimized assembly programs are generated and evolved through generations using the genetic operators, which are mutation and selection. As shown in Steps 3a-b (Fig. 3.8), ES carries out a $\mu + \lambda$ strategy, where the initial parent population includes μ empty programs that contain only the epilogue and prologue nodes. The internal nodes of the μ DAGs are inserted and modified with the random mutation operators with self-adaptive characteristics (will be discussed in Section 3.2.2), and non-random tournament selection operators.

In each generation, the algorithm makes use of the mutation operators and a 1-point crossover operator to create λ new individuals. After including these new λ individuals, the size of the parent population is escalated to $\mu + \lambda$, out of which μ offsprings are selected for the next generation. Each individual has to be loopless and syntactically correct. The mutation operators used in this strategy are:

- Add node: A new node is inserted to the DAG.
- Remove node: An existing node is removed from the DAG.
- Modify node: All parameters are modified or changed in a node in the DAG.

A tournament selection operator with size τ selects the μ fittest offsprings from the $\mu + \lambda$ individuals of the parent population. In the earlier methods, the fault coverage obtained for each individual was considered as the objective function. So, after selecting the fittest individuals, the new parent population would consist of test solutions with improved fault coverages than the previous generation. This evolutionary process goes until there is no more improvement in the generations. But this progression is insufficient to find out and preserve the instruction sequences that detect the hard faults. To accomplish this, the objective function is revised so that the search process grows towards a set of test solutions which could discover these faults (Step 3c-d of Fig. 3.8). The Steps 3a-d will be reiterated until improved test programs are not generated for a predetermined number of generations. Afterward, if there are more components for

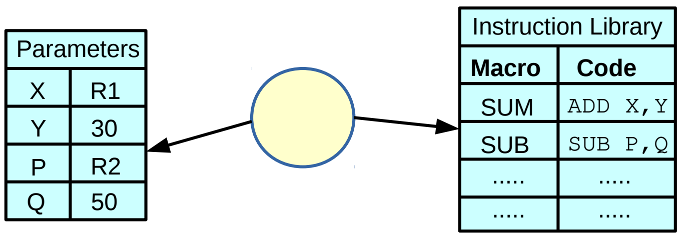


Figure 3.9: *A DAG Node*

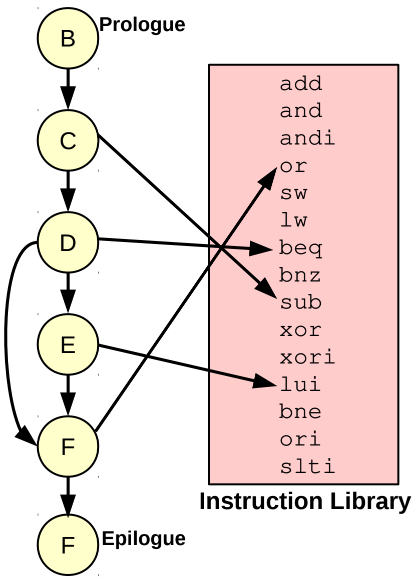


Figure 3.10: *A DAG Representation of Test Program*

which the test programs are to be generated, the next component which has the highest number of faults is selected, and Steps 2 and 3 are repeated.

The greedy coverage method is described in Algorithm 1. Let f_{branch}, which is the statement assignment fault on branch signal, be a corner case fault and FCC_i be the set of all faults detected by at least one of the test solutions selected from their parent population until i^{th} generation. FCC_i is updated in each generation with the newly detected faults. For example, a fault f_{branch} is added into FCC_i when detected for the first time by a selected individual, as shown in Step 3 of Algorithm 1.

As illustrated in Step 1 (Algorithm 1), the objective function is defined as $F = |FN|$, where FN is the set of newly detected faults by an individual test solution apart from the faults in the FCC_i of the previous generation. The cardinality of the FN is considered as the objective function in order to generate offsprings which cover a wide range of faults. So, if the corner case faults are detected by an individual, it would possibly

```
ALGORITHM 1: Greedy Coverage Method
  Input: A new generation of individuals $P_i^1, P_i^2, \ldots, P_i^{\mu+\lambda}$ with set of covered
         faults $FC_i^1, FC_i^2, \ldots, FC_i^{\mu+\lambda}$.
         $FCC_i$, which is a set of all covered faults until the generation $i$.
  Output: Modified $FCC_{i+1}$ in the $(i+1)^{th}$ generation.
1 Select new population $P_{i+1}^1, P_{i+1}^2, \ldots, P_{i+1}^{\mu}$ of the $(i+1)^{th}$ generation with fitness
    function of $j^{th}$ individual as the cardinality of the set of newly covered faults
    $FN_i^j = FC_i^j - FCC_i$ ;
2 If $|FN_i^j|$ is same for all individuals in the tournament, then the fitness function is
    $|FC_i^j|$;
3 If an individual $j$ is selected, update $FCC$ with its newly detected faults:
    $FCC_{i+1} = FCC_i \cup FN_i^j$;
```

be selected for the next generation as it is likely to have a higher $|FN|$. In this case, an individual that has detected all the previously covered, easily detectable functional faults, such as stuck-at-0 or stuck-at-1 fault in the *alu_out* signal, may not survive to the later generations.

If all the individuals that take part in the selection process have the same cardinality for FN, then the offsprings are selected with the usual fault coverage $|FC|$ as the objective function (Step 2 of Algorithm 1). In step 3 of Algorithm 1, FCC is updated with the elements of FN of each selected individual. When we consider $|FN|$ as the objective function the fault coverage will surely reach up to the optimal value because, in this method, the test sequences which cover the corner cases are given higher preference in selection. On the other hand, as a result of this modification, the overall test generation time may increase because individual solutions with highest fault coverages are not necessarily selected every time. To remediate this time consumption issue of the test generation, a testability based instruction selection method is introduced in the next subsection.

3.2.1 Testability of Processor Components

The testability implies the potential of a processor component to be tested by an instruction in the ISA. It is calculated from the observability and controllability characteristics, which are evaluated using a simulation-based template ranking method. To calculate the

```
load  x<a>, <temp1>
load  x<b>, <temp2>
add   x<a>, x<b>, x<c>
store x<c>, <dest>
```

Figure 3.11: *A General Instruction Template*

testability of a processor component corresponding to an instruction, a set of templates is created for that instruction. An instruction template is built around a key instruction which synthesizes the test patterns and captures the response.

Fig. 3.11 shows a single instruction test program template t, which is built around *add* macro. Here, *add* is the key instruction and the other 3 are supporting instructions. The output of the *add* operation is stored into an observable memory location *dest*. The terms $x\langle a\rangle$, $x\langle b\rangle$ and $x\langle c\rangle$ in the template t depict 3 general purpose registers and the values in angled brackets are the settable fields. For each macro in the instruction library, similar test templates are developed.

As shown in Algorithm 2, the selection probability for an instruction is evaluated using the testability values. Observability and controllability values are two testability parameters calculated using output and input matrices, *Obs* and *Cont*, respectively as shown in Algorithm 2. Each cell of these matrices refers to the observability (or controllability) of a component when a template instance is applied as the test program on the processor (Steps 3 and 4 of Algorithm 2). A template instance is an instruction template with random values assigned to the settable fields.

Definition 1. ***(Observability)*** *Let $O_1, O_2, O_3, \ldots, O_p$ be the output values of component c and $t_1, t_2, t_3, \ldots, t_r$ be the instances of the instruction template t. The observability matrix element $Obs(i,j)$ is 1, only if a random value which is injected in the output O_i is propagated to any of the observable destinations of the processor when the template instance t_j is applied on the processor. The overall observability of the template t for the component c is defined as:*

$$OBSERV = \frac{\sum_{i=1}^{p}\sum_{j=1}^{r} Obs(i,j)}{p \times r} \tag{3.1}$$

Definition 2. ***(Controllability)*** *Let $I_1, I_2, I_3, \ldots, I_q$ be the input values of c and $t_1, t_2, t_3, \ldots, t_r$ be the instances of the template t. The controllability matrix element $Cont(i,j)$ is 1, only if a random value which is assigned to any of the settable fields in* *the template instance t_j is propagated to the input I_i of the component, when the template instance t_j is applied on the processor. The overall controllability of the template t for the component c is defined as:*

$$CONTROL = \frac{\sum_{i=1}^{q} \sum_{j=1}^{r} Cont(i,j)}{q \times r} \tag{3.2}$$

Definition 3. ***(Testability)*** *The Product of OBSERV and CONTROL gives the final testability value of the template t for the component c as follows:*

$$TEST = OBSERV \times CONTROL \tag{3.3}$$

The above testability definitions are used to derive the selection probability of the instructions as shown in Steps 5-8 of Algorithm 2, which are to be included in the assembly code test program, corresponding to each processor component, during the evolutionary process for test program synthesis.

The testability of each component corresponding to the macros is calculated using Equations (3.1), (3.2), and (3.3). Using these testability characteristics, the selection probability of instruction i is derived as:

$$SEL_i = \frac{TEST_i}{\sum_{k=1}^{n} TEST_k} \tag{3.4}$$

Where $TEST_1$, $TEST_2$, $TEST_3$, $\ldots$, $TEST_n$ are the testability values of the n instructions for component c.

The mutation operator exploits the testability properties of instructions to make the test synthesis faster and adequately directs towards the best optimum solution. The *add_node* mutation operator selects a new macro from the instruction library with the selection probability of its instruction (Step 9 of Algorithm 2). Essentially, selection probability defines the probability for a macro to be in the test program, i.e., a macro with a larger SEL value for its instruction has a higher probability to get included in the test program.

3.2.2 Self Adaptation of Evolutionary Strategies

For fine-tuning the progress of the genetic population during the evolutionary process, the self-adaptive evolution strategies are employed. The self-adaption parameters, which

ALGORITHM 2: Selection Probability Calculation

```
   Input: Template instances $t_1, t_2, t_3, \dots, t_r$ of a template $t$
   Output: Selection probability of $t$ for component $c$
1  Let $O_1, O_2, O_3, \dots, O_p$ be the output values of $c$;
2  Let $I_1, I_2, I_3, \dots, I_q$ be the input values of $c$;
3  $Obs(i, j) = 1$, if output $O_i$ of $c$ is observable when $t_j$ is applied;
4  $Cont(i, j) = 1$, if input $I_i$ of $c$ is controllable when $t_j$ is applied;
5  The Observability of $t$ for $c$ is calculated using Equation 3.1;
6  The controllability of $t$ for $c$ is calculated using Equation 3.2;
7  The Testability of $t$ for $c$ is calculated using Equation 3.3;
8  Out of $n$ instructions, the selection probability of instruction $i$ for $c$ is calculated
     using Equation 3.4;
9  The add_node mutation operator selects the instruction $i$ from the instruction
     library with $SEL_i$ ;
```

are activation probability and mutation strength, are modified after each generation for an improved selection and application of the genetic operators used in the evolutionary process. Activation probability of a genetic operator is the probability of its activation in a generation. Initially, each genetic operator is assigned with an equal activation probability.

Definition 4. *The activation probability of the genetic operator Z for the next generation is:*

$$\Phi_Z^{new} = \alpha.\Phi_Z + (1 - \alpha).\frac{\Psi_Z}{\Theta_Z}, \tag{3.5}$$

Where Θ_Z is the number of activations of a genetic operator Z and Ψ_Z is the number of successful activations of a genetic operator Z in the current generation. The coefficient α is selected sensibly so that abrupt deviations are avoided.

Let Φ_{add_node}, Φ_{rem_node} and Φ_{mod_node} be the activation probabilities of the *add_node*, *rem_node*, and *mod_node* mutation operators, respectively. Initially, the activation probabilities are set to three equal values: $\Phi_{add\_node} = 1/3, \Phi_{rem\_node} = 1/3, \Phi_{mod\_node} = 1/3$. In the later generations, the activation probability of each mutation operator is calculated using Equation (3.5).

Definition 5. *The mutation strength for each generation is calculated as:*

$$\rho^{new} = \alpha.\rho + (1 - \alpha).\frac{H_\Omega}{\Omega.\lambda}, \tag{3.6}$$

where H_Ω is the number of successful mutations for the last Ω generations. In each generation, λ new solutions are created from a population of μ individuals. When a mutation, which is a genetic operator, is invoked, the number of consecutive elementary modifications expected is EM, where $EM = \frac{1}{1-\rho}$.

The mutation strength ρ, which indicates the depth of a mutation operation in terms of the elementary modifications realized on the parent, is defined in Equation (3.6). If ρ is large, the parent is subjected to a large number of modifications to produce an extremely dissimilar offspring. If most of the mutations are successful, probabilistically, the best optimal solution lies in some other region in the solution search space. Then we increase the mutation strength to find out the optimum solution. Generally, the mutation strength holds a large value during the initial stages of the search process. After exploring the search space exhaustively, mutation strength is reduced gradually in the later generations. So, self-adaptive approaches build up an improved population exploiting the attributes of the population in the previous generations.

Now, an assembly program is generated with an optimized fault coverage and we check whether the attained coverage is sufficient enough as per the fault coverage criteria. If not, we select the next component with the highest number of faults, rank the macros, and apply evolutionary strategies on the test programs. This process goes on until sufficient fault coverage is obtained. In each generation, the offsprings are created and are subjected to a syntactical analysis. But the instructions may impose some temporal and spatial constraints also. So, the semantic analysis of instruction sequences will allow us to generate better test programs.

Once the evolutionary process is over, the majority of the faults, which are sensitive to any of the possible test sequences, are identified. Even though our test program tries to cover all the faults, including the faults which are difficult to identify, some of the faults will never come across any test path. They remain undetected for any test program sequences either because those faults are injected on non-executable VHDL statements or because none of the observable locations are susceptible to those faults. In the next section, we discuss the implementation details and analyze the experimental results.

3.3 Experimental Study for MIPS Processor

3.3.1 Testability and Coverage Evaluation

In our approach, using the 10 behavioral fault representations shown in Table 3.1, 270 faults have been modeled and tested on the 7 components of a 32-bit configurable MIPS processor. The testability analysis was performed on a Xilinx ISE platform and the GHDL simulator was used for the coverage evaluation of the test program individuals during the automation. The observable locations, which are the memory locations and the 32 general purpose registers, are tracked and compared with an expected response for fault identification using scripts written in Python language.

This evaluation framework is associated with the test generation process to develop the proposed greedy based scheme as follows: *Initially, a group of macros corresponding to 8 sequential and one branch instructions is ranked based on their selection probability using a simulation-based technique and these instructions are stored in the instruction library. Further, the $(\mu + \lambda)$ evolutionary core module is developed using ANSI C for the test program synthesis.*

The μGP algorithm [2] is the baseline approach of the modern evolutionary techniques in SBST automation. The self-adaptative characteristic of this evolutionary approach internally improves the genetic operators to achieve the required test patterns. Initially, we investigate the coverage statistics of the traditional μGP technique [2].

To compare the μGP approach [2] and the proposed greedy based approach, a common fault model is required. We adopt the behavioral fault models because these models are highly correlated to the gate-level and the physical defects of the processor, as discussed in Section 3.1. However, the statement coverage calculation employed in the μGP approach could not accurately measure the quality of the test program because of the low correlation between statement coverage and gate-level faults [49]. So, we associate the evolutionary core of μGP [2] with the test quality evaluation of behavioral fault models [5] and compare the results with that of the proposed greedy based approach.

Fault coverage of each component obtained using μGP [2] is depicted in Fig. 3.12. For the computational functional component ALU, a consistent fault coverage of 99% is achieved after 140 generations of the population and more than 96% of faults of the

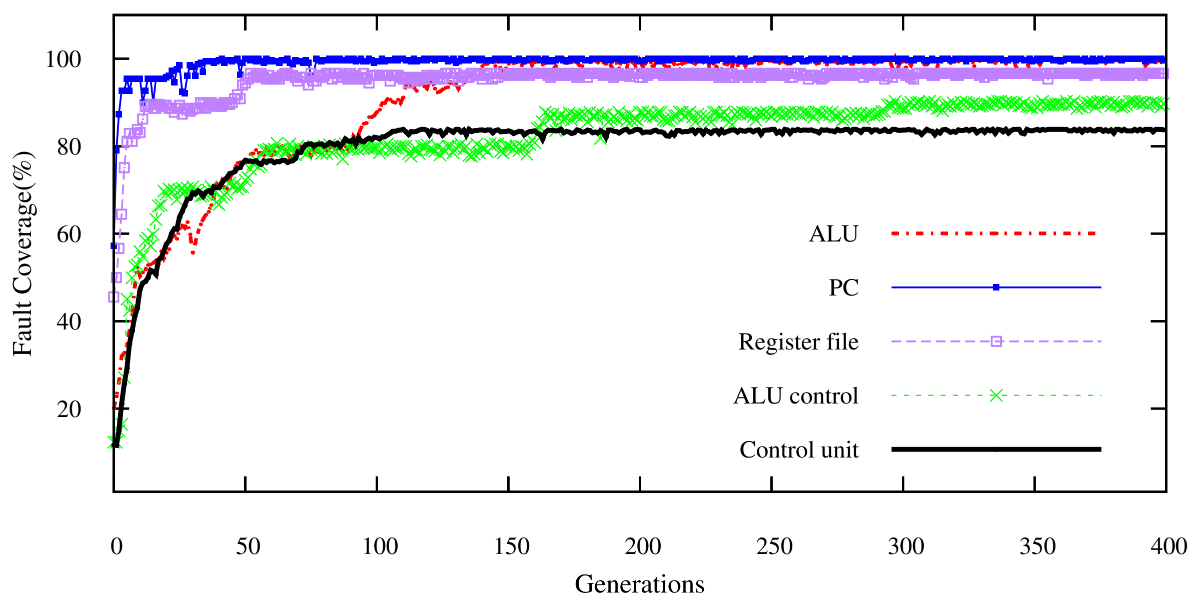
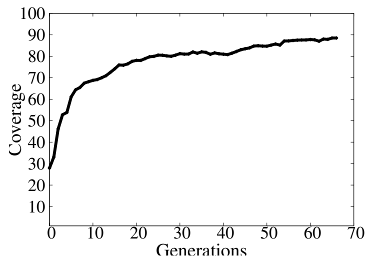


Figure 3.12: *Fault Coverage of Modules Using Traditional μGP [2] on MIPS Processor*

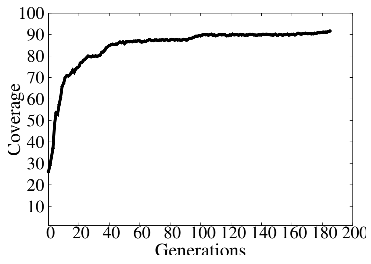
register file, which is a storage functional component, are also covered by the population of test programs after 50 generations. So, a high fault coverage is obtained for the functional components with the help of the test programs synthesized using the conventional μGP approach [2] on behavioral fault models [5].

On the other hand, certain control components, such as control unit, have a lesser coverage when compared with the functional components as illustrated in Fig. 3.12. The control components which have a very small number of possible faults and high fault coverage, such as program counter logic with 99% fault coverage, hardly enhances the overall coverage because the majority of the controller faults either belong to control unit or to ALU control unit. But after 150 generations, the fault coverage of the control unit is saturated at 83% and after 300 generations, the fault coverage of the ALU control unit is saturated at 89%. So, more than 10% of the overall controller faults remain undetected which altogether constitutes an undesirable coverage.

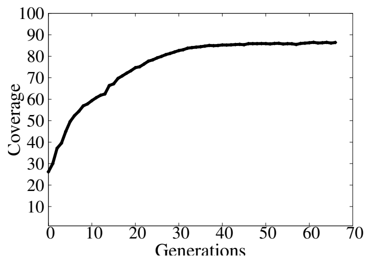
In the μGP platform, different sets of μ and λ values are applied, and the coverage distributions obtained after a definitive time period of execution are shown in Fig. 3.13. Each of these simulations consumed an equal execution period of 170 hours. A fault coverage of 88.5% was achieved by the test program developed using the ES with $\mu = 10$



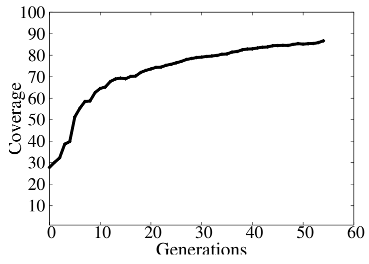
a) $\mu = 10$ *and* $\lambda = 20$



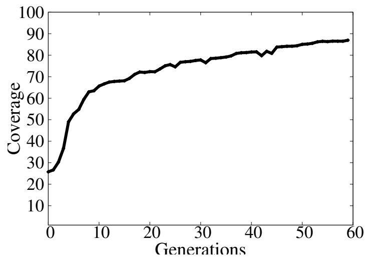
b) $\mu = 20$ *and* $\lambda = 10$



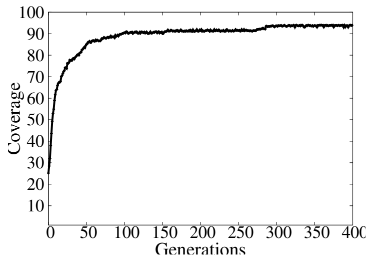
c) $\mu = 30$ *and* $\lambda = 20$



d) $\mu = 20$ *and* $\lambda = 30$



e) $\mu = 40$ *and* $\lambda = 5$



f) $\mu = 10$ *and* $\lambda = 5$

Figure 3.13: *Fault Coverage of Different μ and λ Using the Traditional μGP [2] on MIPS Processor*

and $\lambda = 20$, and 91.6% faults were detected when the $\mu = 20$ and $\lambda = 10$. In the latter case, the number of offsprings produced in each generation is less due to a smaller λ, which carries out a faster convergence. Likewise, a fault coverage of 86.6% is obtained for $\mu = 20$ and $\lambda = 30$, whereas 87.4% of the faults were detected with $\mu = 30$ and $\lambda = 20$, where λ is smaller. When $\mu = 40$ and $\lambda = 5$, the fault coverage is 87.2%, which demonstrates that a higher variation in μ and λ do not necessarily constitute a better coverage. But an adequately superior fault coverage of 95.6% is observed when the evolutionary parameters are $\mu = 10$, $\lambda = 5$.

Now we study the testability analysis on the same MIPS processor for the proposed greedy method. As the values, $\mu = 10$ and $\lambda = 5$ are concluded as efficient and the most appropriate μ and λ values for the test synthesis, these values are thereafter consistently used for the experiments of the proposed greedy based approach. The proposed greedy based $(\mu + \lambda)$ ES executes for 400 generations or until there is no improvement for 40 generations, which is observed as the minimum threshold for a steady-state to occur. The parameters and their specifications used for the proposed self-test synthesis are shown in Table 3.2. The selection method adopted here is tournament selection with tournament size $\tau = 2$.

To calculate the selection probability, the testability statistics are estimated with the

Table 3.2: *Parameters Used for the Proposed Test Generation Scheme*

Parameter	**Specification**
Population size (μ)	10
No.of offsprings (λ)	5
Selection Method	Tournament selection
Tournament size(τ)	2
No. of Generations	400
Steady-state threshold	40
Evolutionary approach	ES($\mu + \lambda$)
Coverage evaluation method	Behavioral fault simulation

Table 3.3: *Observability Values of MIPS Processor*

Module	**Instruction**								
	store	**load**	**addi**	**beq**	**add**	**sub**	**and**	**or**	**nor**
ALU control	10.0	10.0	10.0	5.0	1.0	4.0	3.0	5.0	4.0
ALU	8.5	9.0	4.0	4.0	9.0	9.5	9.5	9.5	7.5
Control unit	6.0	14.3	15.4	22.4	31.3	39.0	45.3	54.3	61.3
MUX	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
PC	10.0	10.0	10.0	1.0	7.0	8.0	8.0	9.0	8.0
Register File	3.5	4.5	3.5	6.0	5.0	6.0	5.0	6.0	2.0
Sign Extend	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table 3.4: *Controllability Values of MIPS Processor*

	Instruction								
Module	**store**	**load**	**addi**	**beq**	**add**	**sub**	**and**	**or**	**nor**
ALU control	10	10	10	10	10	10	9	10	10
ALU	14	13	12	11	20	20	18	20	20
Control unit	1	2	1	8	1	1	1	1	1
MUX	11	12	9	18	19	19	19	19	19
PC	4	4	1	1	1	1	1	1	1
Register file	41	35	38	40	39	39	38	39	38
Sign extend	10	10	10	10	10	10	10	10	10

help of 10 different template instances of each instruction applied as a test routine. If an input (or an output) of a processor component is controllable (or observable) using any of these routines, the controllability (or observability) value of that template instruction would be incremented by 1. Likewise, the average observability and controllability values of each instruction are evaluated as shown in Table 3.3 and 3.4, and their product would finally determine the average testability of each instruction as shown in Table 3.5.

Using the testability statistics, the selection probabilities of the instructions for dif-

Table 3.5: *Testability Values of MIPS Processor*

Module	Instruction								
	store	**load**	**addi**	**beq**	**add**	**sub**	**and**	**or**	**nor**
ALU control	100.0	100.0	100.0	50.0	10.0	40.0	27.0	50.0	40.0
ALU	119.0	117.0	48.0	44.0	180.0	190.0	171.0	190.0	150.0
Control unit	6.0	28.6	15.4	179.5	31.3	39.0	45.3	54.3	61.3
MUX	11.0	12.0	9.0	18.0	19.0	19.0	19.0	19.0	19.0
PC	40.0	40.0	10.0	1.0	7.0	8.0	8.0	9.0	8.0
Register file	143.5	157.5	133.0	240.0	195.0	234.0	190.0	234.0	76.0
Sign extend	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0

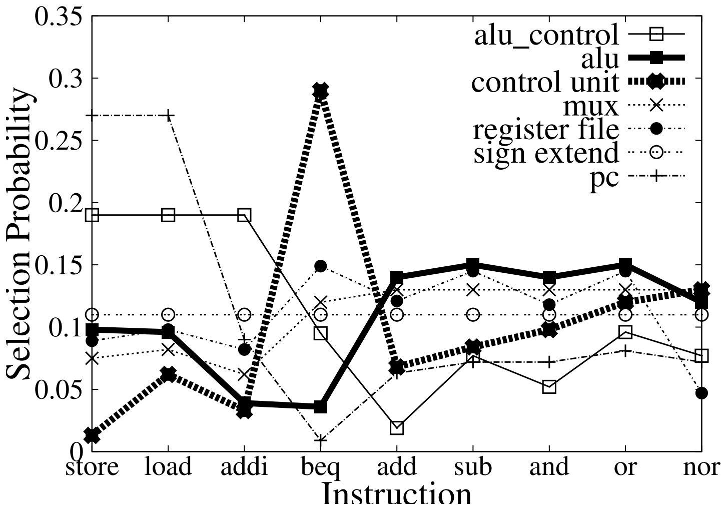


Figure 3.14: *Selection Probabilities of Instructions for Different Modules of MIPS Processor*

ferent processor components are calculated as shown in Fig. 3.14. For a processor component, an instruction is assigned with a larger selection probability if it holds a higher value for the testability metric. It illustrates that the control unit could be tested efficiently if the branch instruction is selected with a higher probability because of its substantially higher testability for the control unit. For the control unit, a higher selection probability of 0.29 is calculated with the branch instruction, whereas the load and store instruction have lesser probabilities of 0.062 and 0.013 respectively, to get selected. For ALU, add, sub, and, or and nor instructions have larger probabilities of 0.14, 0.15, 0.14, 0.15, and 0.12 respectively, but load, store and branch instructions hold values which are less than 0.1. This points out that the selection probabilities of the arithmetic and logical instructions are higher compared to the other instructions for ALU.

Apart from the greedy coverage buffers, a global buffer is also declared to identify

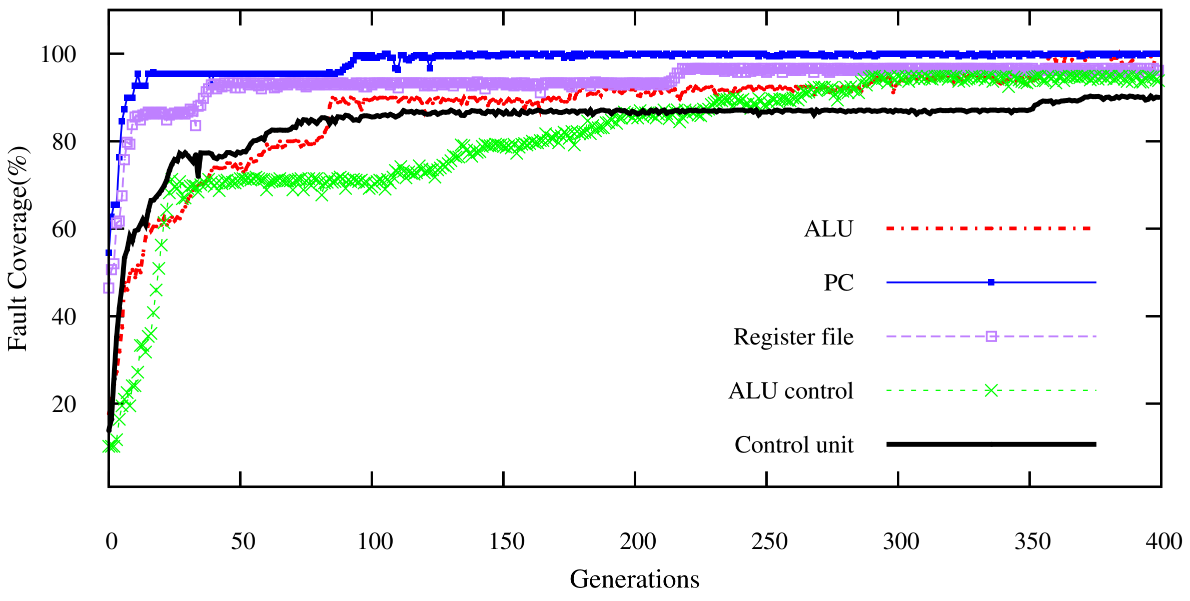


Figure 3.15: *Improved Fault Coverage of Modules Using the Proposed Greedy Based Approach for MIPS processor*

the untestable faults. For example, for the considered MIPS processor shown in Fig. 3.5, a behavioral assignment statement faults on *register_dest* signal, which selects the destination register for a write operation in the register file, and *mem_to_reg* signal, which selects the data to be written in the destination register from the output of memory and the output of ALU, are untestable during the execution of a store instruction. The global buffer is updated with the new faults whenever an offspring detects an uncovered fault regardless of its selection for the next generation. After 400 generations, 25 faults have remained uncovered in the global buffer and they were declared to be untestable.

Now, we conduct a comparison of the proposed greedy method and the μGP technique [2]. In Fig. 3.15, we have shown the improvement in the coverage of the controller component faults using the proposed greedy based method comparing with the traditional method [2] shown in Fig. 3.12. The test programs developed using the conventional evolutionary strategies [2] could not extensively test the main controller components. So, we have focused on the improvement of the coverage of those controller components when the test programs developed using our proposed ES greedy based approach are applied.

In an average based study of the coverage statistics, improved coverages of 93.5% for the ALU control component and 89.2% for the control unit component are observed. Eventually, these optimal test programs developed for each component are collected and sequentially combined to analyze the overall coverage advancement. In Fig. 3.16, we compare the traditional μGP algorithm with the proposed greedy approach for the overall coverage. It could be observed that until 100 generations, the convergence rate of the proposed greedy method is lesser when compared with the μGP technique [2]. But as the test program synthesis is carried out in a single run of the ES, a marginal difference in the test generation time is insignificant. For the proposed approach, after 350 generations, the combined coverage of all components gradually progresses towards coverage of 96.32% whereas the μGP approach achieves a maximum coverage of 93.9% and arrives at a steady-state after 300 generations.

For MIPS processor, nearly 80% of the faults are easy-to-detect faults and the remaining are hard-to-detect faults. The traditional μGP technique always selects the test programs with highest coverage whereas the proposed greedy method selects the test programs that detects hard-to-detect faults. In the initial 100 generations, the traditional method searches for instruction sequences that detect a majority of the easy-to-detect faults and the coverage easily reaches upto 80%. But later, the search process stagnates because the instruction sequences that detect the hard-to-detect faults are given no preference.

In the proposed greedy approach, a higher preference is given for the hard-to-detect faults. So, in the initial generations, the coverage grows gradually because the instruction sequences that detect easy-to-detect faults may not be selected. In the initial generations, the test programs developed by proposed method may detect hard-to-detect faults only but may not detect every easy-to-detect faults. So, the coverage would be lesser because 80% of the faults are easy-to-detect faults. This indicate that the convergence rate of the proposed greedy method is lesser when compared with the traditional μGP technique until 100 generations. But in the later generations, larger test programs are evolved which can detect every easy-to-detect faults along with a majority of the hard-to-detect faults.

The proposed greedy approach uses behavioral fault models to evaluate the test programs. But Gate-level defect has the highest correlation with the physical defects. The

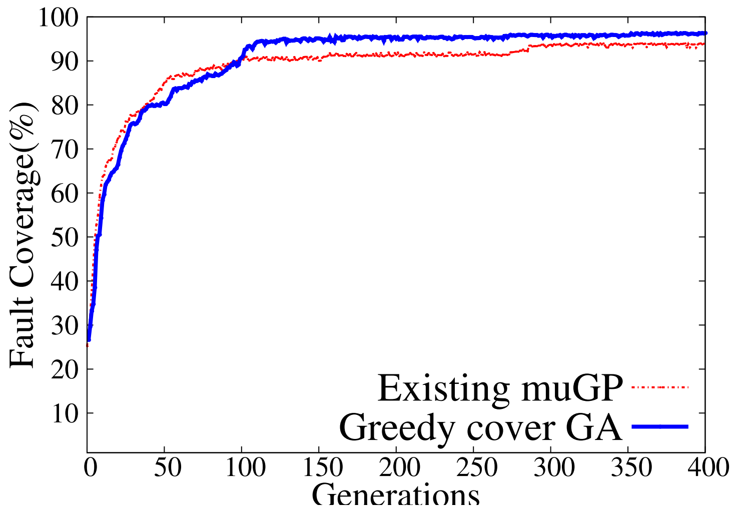


Figure 3.16: *Average Coverage over 400 Generations Using the Proposed Greedy Based Approach for MIPS Processor*

equivalence between the behavioral faults and the gate-level faults for general circuits has already been experimentally analyzed and verified [49–51]. Now, we study the correlation between the behavioral faults and the gate-level faults on MIPS processor which will establish the test quality of the proposed greedy approach.

3.3.2 A Study on the Effectiveness of Behavioral Fault Models

Nowadays, the gate-level processor testing has been an extraordinarily tedious task due to the high complexity of circuits. This difficulty persuaded the test programmers to develop high-level behavioral fault models for circuits. Due to the complexity of processors, we also adopt these behavioral fault models for processors. However, these behavioral fault models must be reliable and highly correlated with physical defects in terms of effectiveness. This correlation is measured by testing the processor with the behavioral fault models and comparing the achieved fault coverage with that of the gate level fault models. All the previous studies [49–51] have achieved the results that demonstrate the close correlation of gate-level and behavior faults for the general benchmark circuits. No such study has been reported for instruction set processors.

So, we study the correlation between gate-level and behavioral fault models of the MIPS processor for our test generation scheme. In Fig. 3.17, a block diagram of an ALU of MIPS processor is shown. This ALU has two 32-bit operands as input lines, a 2-bit opcode as select line and a status bit. The 2-bit opcode selects a function from the AND,

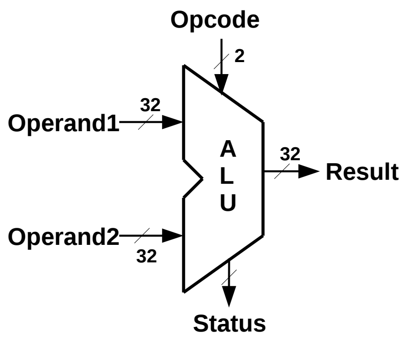


Figure 3.17: *A Block Diagram of 32-bit ALU of MIPS Processor*

```
alu_a, alu_b: IN;
alu_out: OUT;
if(alu_op = "00") then
          alu_out:=alu_a or alu_b;
               if stuck else fault
if(FALSE) then
          alu_out:=alu_a or alu_b;
```

Figure 3.18: *If Stuck Else Behavioral Fault*

OR and ADD operations to produce a 32-bit result. In Fig. 3.18, an $if - stuck - else$ behavioral fault of ALU block is shown. This behavioral fault occurs due to the failure in executing the IF block. To test the behavioral faults of a MIPS processor, a test program is generated as shown in Fig. 3.19. The highlighted part of the test program loads values on two registers R1 and R2 from memory, does OR operation on them, and stores the result back into memory. When this code snippet is executed, operand1 and operand2 of the ALU get values from R1 and R2, respectively. The opcode selects the OR operation and the values of operand1 and operand2 are ORed to produce the result.

In the example shown in Fig. 3.18, the input signals *alu_a* and *alu_b* would be assigned to the values of R1 and R2, respectively. Let the initial value of the *alu_out* signal be 0. If R1 (or R2) is loaded with a non-zero value from memory, *alu_a* (or *alu_b*) also becomes non-zero. Then the signal *alu_out*, the output of OR operation on *alu_a* and *alu_b*, becomes non-zero and eventually, *alu_out* and the expected primary output of ALU also becomes non-zero. But if the behavioral $if - stuck - else$ occurs in the ALU block, as shown in Fig. 3.18, it hinders the execution of the OR statement and therefore, the *alu_out* signal and the primary output of ALU remains zero. The expected output

```
        .......
        .......
lw      R1,offset1(R4)
lw      R2,offset2(R4)
or      R3,R1,R2
sw      R3,offset3(R4)
        .......
        .......
```

Figure 3.19: *SBST Test Code for ALU for MIPS Processor*

in a normal case is non-zero and thus, the fault is detected. Similarly, the behavioral faults of the other functionalities of ALU are also detected using the entire test program, synthesized using the proposed greedy scheme, of which a short code segment is shown in Fig. 3.19.

Now, consider the possible stuck-at-0 and stuck-at-1 gate-level faults in the structural descriptions of the ALU. A 32-bit ALU is built using 32 1-bit ALUs as shown in Fig. 3.20. In Fig. 3.21, the gate-level description of a 1-bit ALU is also shown. Let us consider a stuck-at-0 fault on the output net of the OR gate as shown in Fig. 3.21. Let the registers R1 or R2 of the test program shown in Fig. 3.19, has all-zero (000 . . . 000) and all-one (111 . . . 111) signal values, respectively. Therefore, the inputs, a and b, of each 1-bit ALU will have the values 1 and 0, respectively. Thus, the stuck-at-0 fault of the OR gate gets sensitized as shown in Fig. 3.21. When the OR instruction of the test program is executed, the opcode selects the output of the OR gate towards the output of the ALU. So, the value '0' will be propagated to the output result instead of a value '1'. The expected output result (1) of the 1-bit ALU is compared with the actual result (0) to identify the fault.

The outputs of the 32 1-bit ALUs altogether form the output of the 32-bit ALU. If a stuck-at-0 fault existed in one of these individual ALUs could be detected by the test program developed using the proposed greedy scheme, it could detect the fault for the 32-bit ALU also. There are 2,368 possible gate-level stuck-at faults in the 32-bit ALU of MIPS processor. We conducted a gate-level fault simulation using ModelSim and observed that 98% of the gate-level faults of ALU are detected by the test program developed using the proposed scheme.

This correlation is measured for the other MIPS processor components also. A cor-

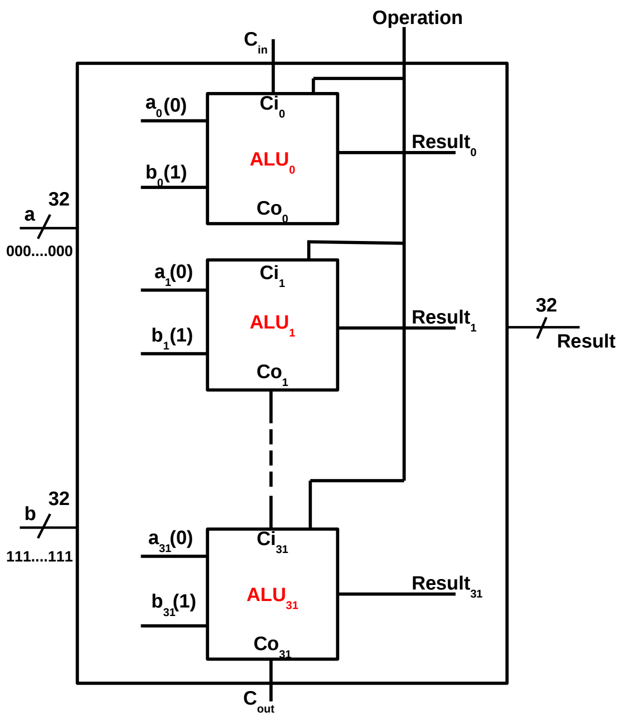


Figure 3.20: *A 32 bit ALU of MIPS Processor*

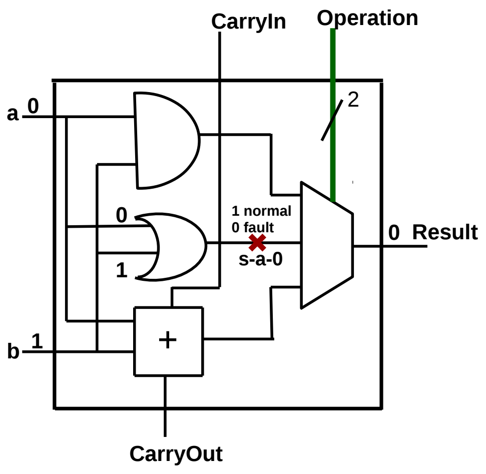


Figure 3.21: *A 1-bit ALU of MIPS Processor*

relation coefficient of 94% for RF, 96% for PC, 100% for MUX, 85% for the control unit, 100% for the sign extend unit and 88% for ALU control unit are achieved. Eventually, an overall correlation of 94% is observed between gate-level faults and behavioral faults of the MIPS processor.

So, the proposed greedy approach is certainly more coverage-efficient when compared with the functional test program synthesis method [2] because a higher fault coverage

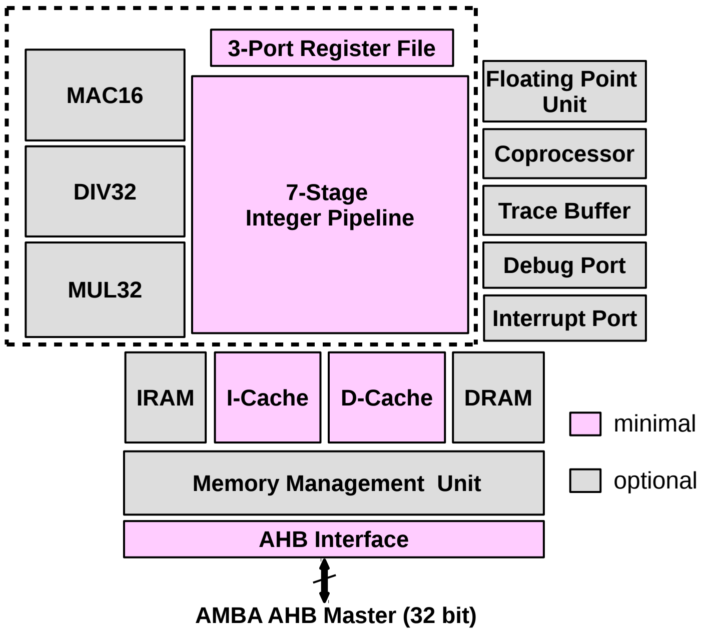


Figure 3.22: *Basic Architecture Block Diagram of Leon3 Processor [3]*

(96.32%) is achieved using a fault model (behavioral) of better correlation (94%) with the gate-level defects.

3.4 Experimental Study for Leon3 Processor

Now, we study the efficacy of the proposed greedy based test generation scheme on a Leon3 processor, which has a larger, and more complicated hardware implementation than a MIPS processor. This could demonstrate the scalability of our proposed greedy approach. The Leon3 processor model is a 32-bit configurable processor of SPARC V8 architecture. It has a 7-stage pipeline and exploits the IP cores of GRLIB IP library. The 10 behavioral fault representations that are used for MIPS processor, as shown in Table 3.1, are used for Leon3 Processor also. The command-line options of ModelSim simulator are used for the response calculation when the synthesized test programs are applied on the processor. From these responses, the contents of the observable locations are extracted and are matched with the golden responses to uncover the defects using a set of scripts written in Python language.

A block diagram of the basic architecture of the Leon3 processor is illustrated in Fig. 3.22, where the minimal components are shown in a magenta background and the optional components are shown on a grey background. In Leon3 processor, a 7-stage

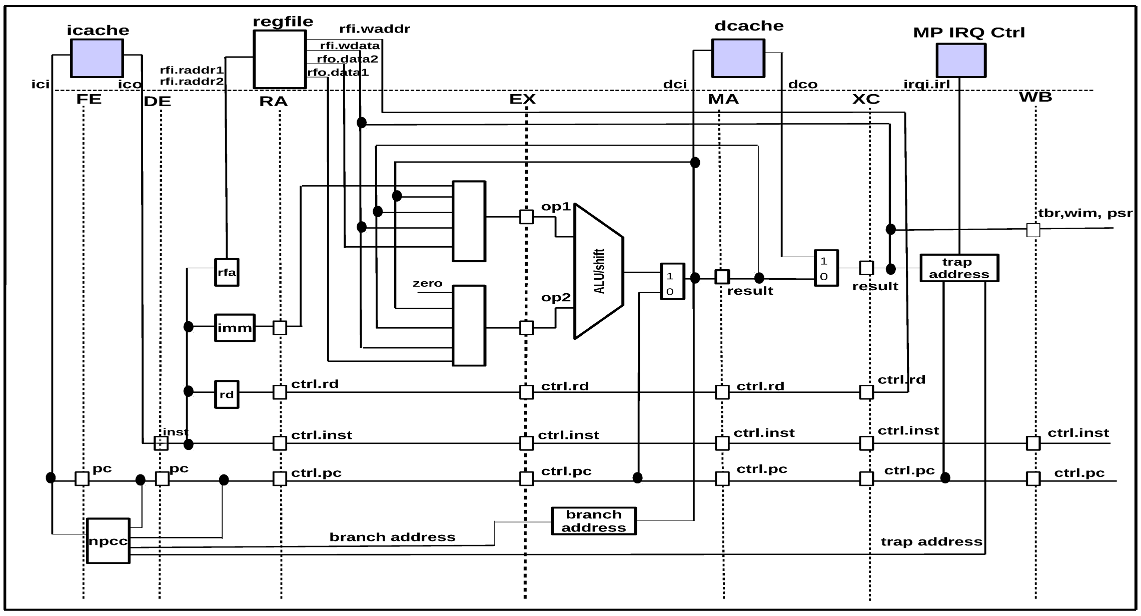


Figure 3.23: *Integer Pipeline Unit of Leon3 Processor [3]*

integer pipeline controls the processor execution. The 7 stages of this pipelining are independent and achieve high performance by using forwarding paths and delayed branches. For the test synthesis of the considered Leon3 processor, the main computational blocks are selected for testing. The main computational components include the minimal components, such as the integer pipeline, register file and some of the optional components, such as the hardware divide, multiply and MAC components. The non-computational blocks and memory blocks are not considered. These main computational blocks to be tested are shown in a dashed rectangle in Fig. 3.22.

In Fig. 3.23, the integer pipeline of the Leon3 processor is illustrated [3]. It has an ALU/Shift unit, 5-to-1 multiplexers, 2-to-1 multiplexers, and several control registers and special registers. The memory and other non-computational components are highlighted in the blue background and the processor components, which are to be tested, are shown in the white background.

The register file consists of 40 to 520 general-purpose 32-bit registers among which 8 registers are *global* and a window of 2-to-32 16-register sets. The 32-bit integer multiplier and divider, and the 16-bit MAC unit could be tested using the instruction set that carries out their functionality.

The controllability and observability values estimated using 10 template instances

Table 3.6: *Observability Values for Leon3 Processor*

Module	Instruction												
	add	**sub**	**umul**	**smul**	**udiv**	**sdiv**	**umac**	**smac**	**and**	**sll**	**bne**	**ldd**	**std**
3-port Register file	3.5	3.5	3.0	3.0	3.0	3.0	3.0	3.0	4.0	4.5	4.0	2.0	2.5
ALU/Shift	9.5	9.0	4.0	4.0	3.0	3.5	4.5	3.5	8.5	9.0	4.0	8.5	8.0
Control registers	2.6	3.0	2.4	2.4	4.0	4.0	2.3	2.3	3.0	3.0	5.5	2.0	4.3
Status registers	2.0	4.3	3.0	3.0	3.0	3.0	3.0	3.0	2.5	3.5	3.5	2.0	2.5
5-to-1 MUX	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2-to-1 MUX	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
PC	8.5	7.5	8.0	7.5	7.0	7.5	8.5	8.0	7.5	8.0	2.0	10.0	10.0
H/W Multiplier	2.3	2.0	9.0	9.0	2.0	2.3	1.0	1.0	3.0	3.0	3.0	3.0	3.0
H/W Divider	3.4	2.0	5.0	4.5	8.0	7.5	3.0	3.0	2.3	2.4	3.0	3.0	3.0
H/W MAC	3.0	3.0	2.5	3.5	2.3	3.3	7.0	7.5	1.0	3.0	3.5	2.3	2.3

Table 3.7: *Controllability Values for Leon3 Processor*

Module	Instruction												
	add	**sub**	**umul**	**smul**	**udiv**	**sdiv**	**umac**	**smac**	**and**	**sll**	**bne**	**ldd**	**std**
3-port Register file	37	37	35	35	34	35	34	34	32	34	34	35	35
ALU/Shift	13	13	8	7	7	7	8	8	12	12	6	8	7
Control registers	1	2	1	1	1	1	1	1	1	1	4	1	1
Status registers	3	3	2	3	3	3	3	3	2	3	4	3	3
5-to-1 MUX	10	10	10	10	10	10	10	10	10	10	9	6	5
2-to-1 MUX	10	10	10	10	10	10	10	10	10	10	10	10	10
PC	3	3	3	3	3	3	3	3	3	3	1	5	4
H/W Multiplier	3	2	9	9	3	3	3	3	3	3	3	5	5
H/W Divider	3	3	2	2	7	7	4	4	3	3	3	6	5
H/W MAC	1	1	3	3	2	3	7	6	3	3	2	5	4

for each instruction are computed as shown in Table 3.6 and Table 3.7. The testability values, which is their product, is shown in Table 3.8. The selection probability of each instruction is calculated from these testability values and are observed to be similar to that of the MIPS processor because of the equivalences in the architecture and similar RISC instruction sets of both MIPS processor and Leon3 processor. For the control registers, a higher testability value of 22 is observed with the *bne* branch instruction. For ALU/Shift component, evidently, arithmetic and logical instructions have higher testability values of above 100. Likewise, Hardware multiplier, divider, and MAC units have the highest selection probability values for mul, div and mac instructions, respectively.

A smaller μ and λ values ($\mu = 10$, $\lambda = 5$), and a tournament size of $tau = 2$ are chosen for the proposed greedy based ES strategy. These μ and λ values were chosen because they were observed to have an efficient fault coverage for MIPS processor as shown in Fig. 3.13. Our greedy-cover based algorithm runs for 400 generations with a steady-state threshold of 40 generations.

A comparison of results of the existing μGP techniques and the proposed greedy

Table 3.8: *Testability Values for Leon3 Processor*

Module	Instruction												
	add	**sub**	**umul**	**smul**	**udiv**	**sdiv**	**umac**	**smac**	**and**	**sll**	**bne**	**ldd**	**std**
3-port Register file	129.5	129.5	105.0	105.0	102.0	105.0	102.0	102.0	128.0	153.0	136.0	70.0	87.5
ALU/Shift	123.5	117.0	32.0	28.0	21.0	24.5	36.0	28.0	102.0	108.0	24.0	68.0	56.0
Control registers	2.6	6.0	2.4	2.4	4.0	4.0	2.3	2.3	3.0	3.0	22.0	2.0	4.3
Status registers	6.0	11.9	6.0	9.0	9.0	9.0	9.0	9.0	5.0	10.5	14.0	6.0	7.5
5-to-1 MUX	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0	9.0	6.0	5.0
2-to-1 MUX	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0
PC	25.5	22.5	81.0	67.5	21.0	22.5	25.5	24.0	22.5	24.0	2.0	50.0	40.0
H/W Multiplier	6.9	4.0	81.0	81.0	6.0	6.9	3.0	3.0	9.0	9.0	9.0	15.0	15.0
H/W Divider	10.2	6.0	10.0	9.0	56.0	52.5	12.0	12.0	6.9	7.2	9.0	18.0	15.0
H/W MAC	3.0	3.0	7.5	10.5	4.6	9.9	49.0	45.0	3.0	9.0	7.0	11.5	9.2

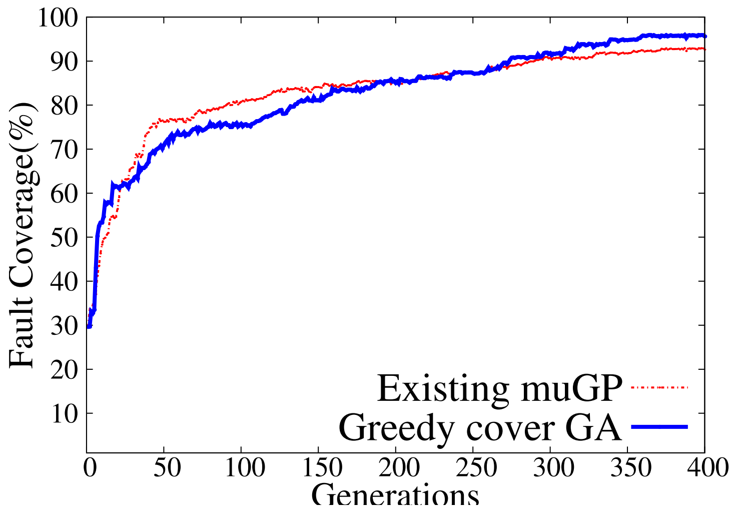


Figure 3.24: *Comparison of Existing μGP and Proposed Greedy Based Method over 400 Generations for Leon3 Processor*

method is shown in Fig. 3.24. An adequate coverage improvement is visible in Leon3 processor also. In the initial stages, the convergence is slower for the greedy based algorithm than the conventional μGP techniques. After the execution of 300 generations, both approaches achieve coverage of 90% and subsequently, the μGP process stagnates with coverage of 92.9% and the proposed greedy based method progresses towards a better coverage of 95.8%.

Although the gate-level implementation of a Leon3 processor is exceedingly larger and complicated than MIPS, the number of behavioral faults varies marginally. For example, let us consider the ALUs of MIPS and Leon3 processors. In accordance with the behavioral fault models in Table 3.1, the ALU block of a MIPS processor has 49 behavioral faults whereas the ALU block of a Leon3 processor has 55 behavioral faults.

There are 5 arithmetic/logical instructions for a MIPS processor and 8 arithmetic/logical instructions for a Leon3 processor. ALU has to select an operation from the arithmetic or logical operations performed by these instructions, and the result is propagated to the output of ALU.

An *alu_select* signal is assigned to select the operation from the arithmetic or logical operations by checking the *op_code* of each operation using $ELSE\_IF$ conditional blocks. For each $ELSE\_IF$ conditional block, 2 behavioral faults ($elsif\_stuck\_then$ and $elsif\_stuck\_else$) are possible. If the number of arithmetic or logical instructions is large in the ISA of a processor, the number of $ELSE\_IF$ blocks will be high and eventually, the number of behavioral faults related to $ELSE\_IF$ blocks will also be high. As the instruction sets of MIPS processor and Leon3 processor are equivalent, the number of behavioral faults associated with the $ELSE\_IF$ blocks are also nearly equal. This concept of a nearly equal number of behavioral faults in Leon3 and MIPS processors leads to the scalability of the scheme.

In the next section, we discuss the contributions of our proposed approach compared with the previous evolutionary approaches in the automation of test generation for processors.

3.5 Comparison and Discussions

The fault evaluation methods used by the previous evolutionary test development methods [2, 41, 52, 53] are not extensive enough because the code coverage metrics of HDL descriptions are not well correlated with the gate level faults and physical defects. The important experimental contributions of our paper are enlisted below:

- ***Scalablility of the proposed greedy based approach:*** The proposed greedy based approach is scalable because our evaluation framework makes use of behavioral level fault models. For large processors like Leon3, the number of possible behavioral faults is not considerably larger than that of the small processors like MIPS as discussed in Section 3.4. This is due to the high level of abstraction of behavioral fault models.

  The MIPS processor has 32k nets and the Leon3 has a total of 336k nets. For gate-level ATPG testing, the number of possible stuck-at faults would be 64k (32k $\times$ 2) for a MIPS processor and 672k (336k $\times$ 2) for a Leon3 processor, which is nearly 10 times that of a MIPS processor. But the number of possible behavioral faults to be tested is 270 for the MIPS processor and only 364 for the Leon3 processor. Here, the number of behavioral defects does not drastically increase with the processor size. So, the overall test generation time for the behavioral faults varies marginally only, which demonstrates the scalability of our method.

- ***Assurance of good test quality for the proposed greedy based approach:*** The previous evolutionary approaches for processor test generation [2, 41, 52, 53, 91] have used fault models which are not well-correlated with the gate-level fault models. So, these approaches could not assure good test quality. The proposed greedy based technique achieves a fault coverage of 96.32% for a MIPS processor and 95.8% for a Leon3 processor for behavioral fault models. Furthermore, we have shown that our behavioral faults have 94% correlation with the gate-level faults. This assures the test quality of our approach as more than 90% of the gate-level faults is detected according to our correlation study described in Section 3.3.2.

3.6 Summary

Genetic algorithm-based approaches are widely accepted methods to search for the high-quality self-test programs for processors [2, 41]. But the overall coverage of these test solutions was insufficient because the corner cases, which are the hard-to-detect faults, were never taken care of. Further, these approaches could not guarantee the test quality because the fault evaluation metrics, such as the statement coverage, were not well-correlated with the gate-level fault models. In this study, an enhanced SBST synthesis for the processor cores is employed by integrating the greedy coverage and the testability features into the traditional evolutionary core module of μGP for the detection of the hard faults. A comprehensive behavioral fault model is used to capture the possible faults in the processors. From the experimental results, we could conclude that our strategy synthesizes test solutions that could detect 96.32% of the testable behavioral faults of a MIPS processor and 95.8% that of a Leon3 processor which affirms the detection of 40% of the hard faults.

The scalability of the proposed scheme is demonstrated by analyzing and comparing the effort in testing the behavioral faults for a MIPS processor and a Leon3 processor. As the number of behavioral faults does not change considerably with the processor size, the test program generation time is nearly equivalent for both processors, which explains the scalability of our scheme. Finally, the close correlation (94%) between the gate-level faults and the behavioral faults establishes a high test quality (gate-level coverage of 90% or above), for our scheme. In the next chapter, we discuss a faster greedy cover-based automated test synthesis technique to yield test solutions quicker than the existing evolutionary techniques, maintaining the fault coverage.

Chapter 4

Rapid SBST (RSBST) Program Synthesis

4.1 Overall Approach of RSBST Program Synthesis

This work discusses a rapid software self-test technique termed as, rapid SBST (RSBST), where the test synthesis is faster compared to that of the conventional μGP approaches [2, 41, 52, 53] and the greedy-based μGP approaches described in the previous chapter (Chapter 3), at the same time does not drop the fault coverage. This faster test synthesis is realized by efficiently integrating the reusability of fault simulation results in the greedy-based μGP framework. In RSBST, redundant test solutions are identified and their simulation results are efficiently reused for a faster fault coverage evaluation. This reusability scheme would expedite the greedy-based μGP evolutionary test synthesis which covers many hard-to-test faults.

As discussed in section 1.6, the simulation of faulty processor models and the successive response collection and comparison are exceedingly time-consuming. In the proposed RSBST technique, we reuse the simulation responses (fault coverage and fault list) for equally-observable test programs to reduce the overall test generation time. The overall approach of RSBST automation scheme for a processor is shown in Fig. 4.1. In this scheme, an evolutionary test generator rapidly develops test solutions of optimal fault coverage exploiting an initial population of test programs, represented as DAGs, and an instruction library.

Formerly, all test programs were evaluated using an external evaluator as shown in Fig. 1.7. To reduce the cost of this external fault evaluation, we have introduced an observability comparator, which compares and identifies the test programs with similar observability. This would effectively reduce the number of fault simulations as the

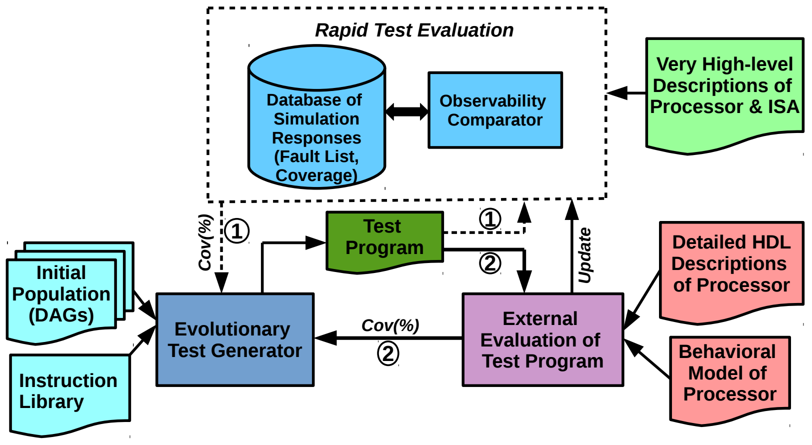

Rapid Test Evaluation
Database of Simulation Responses (Fault List, Coverage)
Observability Comparator
Very High-level Descriptions of Processor & ISA
Cov(%)
①
Test Program
①
②
Update
Detailed HDL Descriptions of Processor
Initial Population (DAGs)
Evolutionary Test Generator
Cov(%)
②
External Evaluation of Test Program
Behavioral Model of Processor
Instruction Library

Figure 4.1: *RSBST Automation Scheme*

simulation responses of the parent test programs, stored in a repository of simulation responses, could be reused for the offspring test programs with equal observability. The repository of simulation responses stores the observability values, fault coverage, and the fault list of the parent chromosomes. This repository is updated with the simulation responses after each external simulation.

The observability comparator calculates the contents of the observable destinations of a test program. To conduct this calculation, the observability comparator makes use of a high-level logic simulation technique which could rapidly evaluate test programs. The contents of the observable locations, which are the simulation responses obtained using the high-level logic simulation, are contrasted with the observability values of the parent chromosomes to check for the scope of reusability of the test program. If an offspring test solution and one of its parent test solutions have identical observability values, the fault simulation could be avoided for the offspring test solution.

The fault coverage of a test program, synthesized by the evolutionary core, is evaluated using either of path ① with dashed lines or path ② with bold lines, as shown in Fig. 4.1. Path ① denotes the reuse of simulation responses using a rapid test evaluation method with a high-level response collection and comparison whereas path ② denotes the regular, external evaluation of the test program. If the simulation responses of a parent chromosome could be reused, external evaluation (path ②) is avoided for the test programs.

Initially, when a test program is synthesized by the evolutionary core, path ① is selected for the rapid test evaluation. In path ①, the observability comparator conducts a rapid high-level logic simulation for the test program and collects the contents of the observable locations. If these observability values of the test program and one of its parent test programs are identical, the simulation responses (fault coverage, fault list, etc.) of that parent test program are reused for the offspring test program. Later, these responses are delivered to the evolutionary test generator.

If the observability values of the test program and any of its parent test programs are not identical, path ② is chosen. In path ②, the external evaluator conducts a time-consuming, detailed fault simulation, as shown in Fig. 1.7, for the test program and delivers the simulation responses to the evolutionary test generator. Later, the repository of the simulation responses is updated with these simulation responses. So, in RSBST, high-level processor descriptions and high-level simulations are used for rapid test evaluation (path ①) whereas detailed HDL descriptions of the processor and time-consuming HDL simulations are used for the external evaluation of test programs (path ②). In the next section, the conventional framework for building SBST programs with the help of a high-level fault modeling approach is discussed.

The test synthesis time is high for the greedy-cover based test synthesis, discussed in Chapter 3, because the test solutions with high fault coverage are not selected always. In fact, the test solutions with considerably lesser fault coverage could even be selected for the future generations if it detects few hard-to-test faults. So, the convergence of the traditional SBST will require more generations of chromosomes and thus, is slower. So, we introduce a reusability technique, which could avoid the fault simulation of test programs of similar influence on the observable locations of the processor, for a faster test synthesis.

4.2 Observability-based Reusability of Test Programs

While the evolutionary process progresses, it is highly likely that the evolutionary core develops individual solutions with similarities in fault simulation results. If the instruction sequences of two test individuals have similar functionalities, the fault simulation results could be reused to reduce the test synthesis time. As the initial μ chromosomes of a generation are replicated from the population of the previous generation, their responses could be naturally reused, thereby avoiding re-simulation. But the new λ chromosomes, which are cultivated using the μ individuals of the current generation,

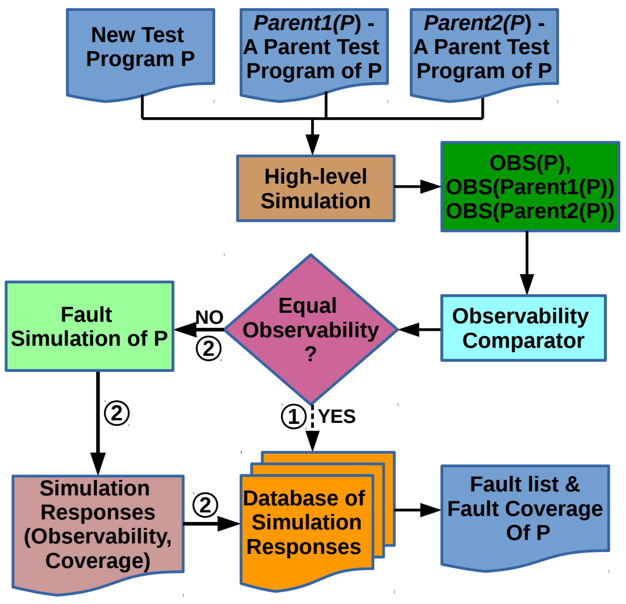

New Test Program P
Parent1(P) - A Parent Test Program of P
Parent2(P) - A Parent Test Program of P
High-level Simulation
OBS(P), OBS(Parent1(P)) OBS(Parent2(P))
Observability Comparator
Equal Observability ?
NO
②
Fault Simulation of P
②
①
YES
Simulation Responses (Observability, Coverage)
②
Database of Simulation Responses
Fault list & Fault Coverage Of P

Figure 4.2: *Test Program Evaluation in RSBST*

has to be dealt with a faster and high-level comparison of the states of the observable destinations.

Processor faults are identified by comparing the contents of the observable locations on the processor. If the values stored in these observable locations following the simulation of an offspring solution are same as that of one of its parent solutions, the set of faults that they could identify are likely to be same; i.e., equally-observable test solutions will likely to have equal fault coverage. In that case, a re-simulation of the offspring solution could be avoided by reusing the identified fault list and the fault coverage of the parent solution.

In Fig. 4.2, the framework for the rapid test evaluation is elaborated. Here, a high-level test program simulation, which has low timing requirement, is used to get the contents of all observable locations. Further, the observability of a newly generated test program and its parents are compared to identify the scope of reusability. As crossover operator is one of the genetic operators employed for the SBST synthesis, each offspring solution P would be composed of the genetic information of two parent solutions. If any of these parents holds the same observability as that of P, the simulation responses of that parent solution could be taken from the repository of simulation responses, where the simulation responses of the test solutions of the previous generations are stored. If no parent holds the same observability as that of P, it is to be fault simulated for the responses, which is a time-consuming process.

In Fig. 4.2, the path ① with dashed lines denotes the rapid test evaluation method and path ② with bold lines denotes the external evaluation of the test program. After the external fault simulation, the repository of simulation responses is updated with the achieved responses. Finally, the contents of the observable locations, the fault list, and the fault coverage are achieved from the responses, stored in the simulation repository. In the next subsection, we discuss how the repository of simulation responses is developed and updated with respect to the fault simulation of each test program solution.

4.2.1 Repository of Simulation Responses

The repository of simulation responses, as shown in Table 4.1, stores these observability values along with the fault coverage and the fault list of each test program solution. Let P_i^j be the j^{th} individual of the population in the i^{th} generation and OBS_i^j be the values of the observable locations after the execution of the test program solution P_i^j on the processor. The content of these observable locations are the simulation responses using which the fault coverage, fault list, etc. are evaluated. The set OBS_i^j is a combination of:

- M_i^j: Memory updates after the execution of the test program solution P_i^j.
- R_i^j: Contents of the register locations after the execution of the test program solution P_i^j.
- O_i^j: Primary output values after the execution of the test program solution P_i^j.

So, $OBS_i^j = \{M_i^j, R_i^j, O_i^j\}$, is the overall test response based on which the quality of the test program solution P_i^j is evaluated. The test quality is dependent on the fault detection parameters, such as fault covered list, fault coverage, etc., which are evaluated using the observability values OBS_i^j.

This repository has a record for each test program solution P_i^j, as shown in Table 4.1. Following the execution of P_i^j, the contents of observable locations OBS_i^j are stored in the record corresponds to P_i^j in the repository. Let F_i^j be the fault coverage achieved by the test program solution P_i^j and FC_i^j be the set of faults covered by P_i^j. Now, F_i^j and FC_i^j, obtained using OBS_i^j values, are also stored in the record correpsonds

Table 4.1: *Repository of Simulation Responses for the Test Program Solutions of i^{th} Generation*

Test solutions of i^{th} generation	**Values of Observable locations**	**Faut Coverage**	**Faut List**
P_i^1	$OBS_i^1 = \{M_i^1, R_i^1, O_i^1\}$	F_i^1	FC_i^1
P_i^2	$OBS_i^2 = \{M_i^2, R_i^2, O_i^2\}$	F_i^2	FC_i^2
...	...	...	...
...	...	...	...
$P_i^{\mu+\lambda}$	$OBS_i^{\mu+\lambda} = \{M_i^{\mu+\lambda}, R_i^{\mu+\lambda}, O_i^{\mu+\lambda}\}$	$F_i^{\mu+\lambda}$	$FC_i^{\mu+\lambda}$

to P_i^j in the repository shown in Table 4.1. As the fault simulation that produces the simulation responses, i.e., OBS_i^j, is time-consuming, we adopt a dynamic, high-level logic simulation as discussed in the next subsection.

4.2.2 High-level Simulation

To conduct a high-level logic simulation of test programs, the encoding and syntax of each instruction are modeled and the instruction operations are simulated using the functionalities of high-level C programming language. Initially, this high-level component allots limited memory for the observable locations of updated memory, registers, and primary outputs. The memory updates are observed with the help of the occurrences of store instructions in the test program. The contents of all registers and primary outputs are observed for logic simulation. Later, each instruction is selected and is logic simulated. After the logic simulation, the contents of these observable locations are collected as the test program responses.

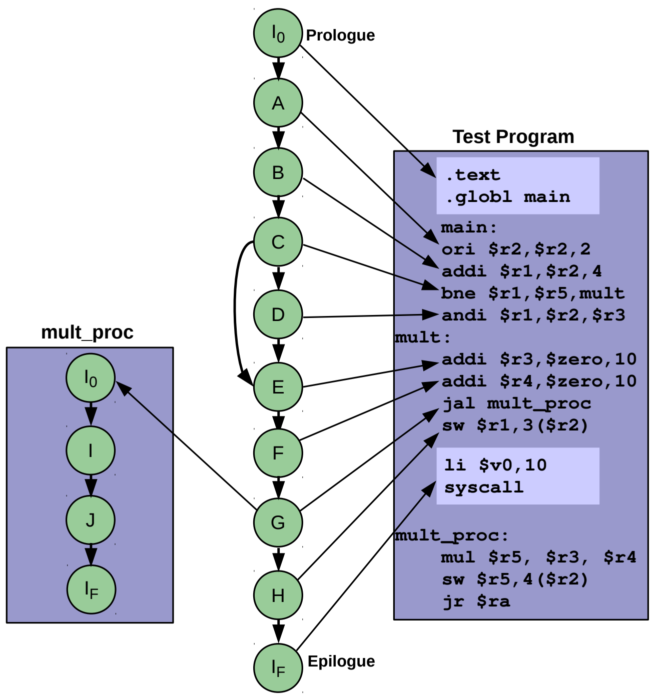
Let p denote the updated memory locations, q denote the register locations, and r denote the primary outputs which are the observable locations of the processor. During the fault simulation of the test program solution P_i^j, developed during the evolutionary test synthesis, the data and control signals are dumped into a simulation log file in each clock cycle. Further, this simulation log file is parsed to observe the memory updates $M_i^j = \{m_1,m_2,m_3,\ldots,m_p\}$, contents of register locations $R_i^j = \{r_1,r_2,r_3,\ldots,r_q\}$, and the primary outputs $O_i^j = \{o_1,o_2,\ldots,o_r\}$. Finally, the overall observability $OBS_i^j = \{m_1,m_2,m_3,\ldots,m_p,\ r_1,r_2,r_3,\ldots,r_q,\ o_1,o_2,o_3,\ldots,o_r\}$ is stored in the record corresponds to P_i^j in the simulation repository.

Each instruction is subjected to logic simulation using the information of its opcode and operands. To realize the operations of instructions, a high-level procedure is developed for each opcode which could be reused for the logic simulation of every instruction composed of the same opcode. The opcode of each instruction is parsed to learn the operation to be performed on the observable locations. These operations are executed on the source operands and simultaneously, the observable locations corresponding to the destination operands are modified. Finally, the contents of observable locations are compiled to form the test program response, based on which the reusability is determined.

A memory update value of set M_i^j is represented using a $\langle address, value\rangle$ pair where $address$ is the updated memory location and $value$ is the updated value at $address$ following the logic simulation of each store instruction. In this set, each value is represented using an $\langle address, value\rangle$ pair which corresponds to each store instruction. For example, a MIPS store word instruction `sw R1,offset(R2)` has $\langle (R2+offset), R1\rangle$ as the $\langle address, value\rangle$ pair, i.e., the memory location $(R2 + offset)$ is updated with the value in $R1$. So, if the test program has n store instructions, n $\langle address, value\rangle$ pairs are observed. These values are extracted using logic simulation to constitute the set of memory updates M_i^j. *However, the test synthesis do not allow more than p store instructions in the test program, which is much less than the overall number of memory locations, to reduce the cost of observability comparison procedure.*

In Fig. 4.3, an imaginary example of an intermediate test program solution of the evolutionary test synthesis is shown. This test program has 8 internal nodes that represent 8 instructions, and a subgraph, with 2 internal nodes, that represents multiplication operation. Suitable procedures are defined corresponding to the opcode of each instruction in the ISA. Also, in the state before the test program execution, every observable location is initialized to zero. Each procedure passes the values of source operands as the input arguments, conducts the operation defined by the opcode, and returns the value of the updated destination operand.

Initially, we select the first instruction ori of the test program shown in Fig. 4.3, with register $r2$ and an immediate value 2 as operands. To conduct logic simulation of this instruction, the ori procedure is activated and modifies the value of $r2$ as 2 in the set of register values R_i^j. Likewise, procedures of each opcode in the ISA is activated for the logic execution of each instruction. For the logic simulation of all the 13 instructions



```
.text
.globl main
main:
ori $r2,$r2,2
addi $r1,$r2,4
bne $r1,$r5,mult
andi $r1,$r2,$r3
mult:
addi $r3,$zero,10
addi $r4,$zero,10
jal mult_proc
sw $r1,3($r2)
li $v0,10
syscall

mult_proc:
mul $r5, $r3, $r4
sw $r5,4($r2)
jr $ra
```

Figure 4.3: *Representation of an Intermediate Test Program of μGP Test Synthesis*

of the test program shown in Fig. 4.3, 9 procedures (*ori*, *addi*, *bne*, *jal*, *sw*, *li*, *syscall*, *mul*, *jr*) must be activated. In this simulation, *addi* procedure is activated 3 times and *sw* procedure is activated 2 times.

After the execution of every instruction in the test program shown in Fig. 4.3, the values in the register locations $r1$, $r2$, $r3$, $r4$, $r5$ of R_i^j becomes 6, 2, 10, 10, 100, respectively. Also, the memory locations m_1 and m_2 of M_i^j are updated with $\langle address, value \rangle$ pairs $\langle 5, 6 \rangle$ and $\langle 6, 100 \rangle$, respectively, and they correspond to two store word instructions. Primary outputs remain unchanged. So, OBS_i^j, which is the compilation of contents of R_i^j, M_i^j, and O_i^j after the execution of P_i^j, are evaluated using the procedures of logic simulation, and are stored in the record corresponds to P_i^j in the simulation repository. In the next subsection, we demonstrate how the observability values of two test programs are compared for the discovery of equally-observable test programs.

4.2.3 Observability Comparator

In a generation of test individuals, μ offsprings are selected and replicated directly from their parents. So, the fault simulation results of the candidate test solutions of a specific generation could be reused for the selected μ offsprings of the next generation. Therefore, the fault evaluation of μ test solutions out of the $\mu + \lambda$ test solutions of any generation becomes effortless. If the test synthesis undergoes a huge number of generations, time consumed for the fault evaluation of the remaining λ test solutions would be enormous. Therefore, while cultivating the remaining λ individuals using the genetic operators, an observability-based reusability method of RSBST could be used to further reduce the test synthesis time.

The observability comparator analyzes the contents of the observable locations of a test program and its parent test programs. Based on the analysis results, the method of test program evaluation is chosen. For example, the test program 1 shown in Fig. 4.4(a) has equal observability as that of the test program 2 shown in Fig. Fig. 4.4(b). Let us assume that the contents of all the registers are the same before the execution of these test programs. In these programs, identical functionalities are executed on the same registers and the eventual memory updates are same; i.e., following the execution of both test programs, the registers $r0$ and $r1$ are assigned with X and $X + 1$, respectively. Also, the memory update corresponds to the store instruction of test program 1 is $\langle(r4 + offset), (r2)\rangle$ and the memory update corresponds to the store instruction of test program 2 is $\langle(r4 + offset), (r3)\rangle$. As the contents of $r2$ and $r3$ were initially the same and remain unchanged, these memory updates are also identical. So, when these two test programs are executed independently on equivalent initial processor states, all the updated memory values, register values, and primary output values are observed to be identical.

Now, consider the actual, computationally intensive fault simulation of these two test programs, as discussed in Section 1.6.3 and Fig. 1.7. The processor is simulated with each of these test programs for N faulty models and a good reference model. Initially, the test program 1 is executed on these models and the simulation responses are collected for every cycle of execution. The fault coverage is evaluated by comparing these simulation responses, which are the contents of the observable locations. However, test program 1 and 2 would have identical contents of observable locations for all $N + 1$ processor models since both of them realize the same functionality on each observable location. So, test program 1 and 2 would have equal coverages too.

As equally-observable test programs are likely to have equal fault coverage, the simulation responses of test program 1 could be reused for test program 2 and vice versa.

```
addi $r0,$zero,X
addi $r1,$zero,X+1
sw   $r2,offset($r4)
```

a) *Test Program 1*

```
ori  $r0,$zero,X
ori  $r1,$zero,X+1
sw   $r3,offset($r4)
```

b) *Test Program 2*

Figure 4.4: *Equally-Observable Test Programs*

ALGORITHM 3: Reusability of Greedy Coverage Method in RSBST Approach

```
Input: $i^{th}$ generation of solutions $P_i^1, P_i^2, \ldots, P_i^{\mu+\lambda}$.
       $FCC_i$ be the set of all covered faults until the generation $i$.
       $OBS_i^j$ be the values of observable destinations when $P_i^j$ is executed.
Output: Set of covered faults $FCC_{i+1}$ of the $(i+1)^{th}$ generation.
1  if $OBS_i^j = OBS_{i-1}^{parent1(j)}$ then
2      Faut list $FC_i^j$ of $P_i^j$ = fault list of $P_{i-1}^{parent1(j)}$ ;
3      Coverage $F_i^j$ of $P_i^j$ = coverage of $P_{i-1}^{parent1(j)}$ ;
4  else if $OBS_i^j = OBS_{i-1}^{parent2(j)}$ then
5      Faut list $FC_i^j$ of $P_i^j$ = fault list of $P_{i-1}^{parent2(j)}$ ;
6      Coverage $F_i^j$ of $P_i^j$ = coverage of $P_{i-1}^{parent2(j)}$ ;
7  else
8      Evaluate fault list $FC_i^j$ and coverage $F_i^j$ of $P_i^j$ using fault simulation. ;
9      Update the simulation repository of $P_i^j$ with the fault list $FC_i^j$ and coverage
         $F_i^j$ achieved using fault simulation ;
10 The fitness function of $P_i^j$ is $|FN_i^j|$, which is the cardinality of the set of its newly
     covered faults, where $FN_i^j = FC_i^j$ - $FCC_i$;
11 If the individual $P_i^j$ is selected, update $FCC_{i+1}$ with $FCC_i$ and the set of newly
     detected faults $FN_i^j$, i.e., $FCC_{i+1} = FCC_i \cup FN_i^j$;
```

The next subsection illustrates the algorithm of Rapid SBST technique which has two significant aspects: 1) observability-based reusability of fault simulation results of test programs, and 2) greedy-based test synthesis for the detection of hard-to-test faults.

4.3 Design of RSBST Scheme

The proposed method of reusability of the fault simulation responses are described in the Step 1-9 of Algorithm 3. Let $P_{i-1}^{parent1(j)}$ and $P_{i-1}^{parent2(j)}$ be the two parent test program solutions of P_i^j. Let OBS_i^j, $OBS_{i-1}^{parent1(j)}$, and $OBS_{i-1}^{parent2(j)}$ are the observability contents of P_i^j, $P_{i-1}^{parent1(j)}$, and $P_{i-1}^{parent2(j)}$, respectively. These two test solutions of $(i-1)^{th}$ generation are subjected to crossover and mutation operators to synthesize P_i^j in the i^{th} generation. After the test program solution P_i^j is synthesized, a high-level internal logic simulation is conducted for the test quality evaluation of P_i^j. Now, we extract the contents of the observable locations of P_i^j from the simulation responses and of its parents $P_{i-1}^{parent1(j)}$ and $P_{i-1}^{parent2(j)}$ from the repository shown in Table 4.1. If OBS_i^j is equivalent to either $OBS_{i-1}^{parent1(j)}$ or $OBS_{i-1}^{parent2(j)}$, the fault coverage and the fault list of the equally-observable parent are selected from the repository in Table 4.1 and are reused for P_i^j, as illustrated in Step 1-6 of Algorithm 3.

If the observabilities of the offspring solution and none of its parent solution are not identical (Step 7-9 of Algorithm 3), the fault coverage and fault list of P_i^j are evaluated using behavioral fault simulation (Step 8 of Algorithm 3). After the test evaluation, the repository record for P_i^j is updated with the fault coverage F_i^j and fault list values FC_i^j, achieved using the fault simulation (Step 9 of Algorithm 3). These simulation responses could be reused for the further generations, i.e., if $OBS_{i+1}^{offspring(j)}$ of the test program solution $P_{i+1}^{offspring(j)}$ in the $i+1^{th}$ generation matches with OBS_i^j of P_i^j, the fault list and coverage of P_i^j are reused for $P_{i+1}^{offspring(j)}$.

In this approach, a set FCC_i refers to the list of all covered faults until the generation i, and FN_i^j is the set of newly detected faults by P_i^j. In Step 10, the objective function is defined as $|FN_i^j|$, which is the cardinality of the set of the newly covered faults. This greedy approach tends to protect the chromosomes that detect the hard-to-test faults through the generations. Finally, FCC_{i+1} is created by merging FCC_i and the set of fresh faults FN_i^j that are detected by the selected chromosomes in the i^{th} generation, as described in Step 11 of Algorithm 3. The experimental results that validate a faster test synthesis with the help of RSBST technique are demonstrated in the next section.

4.4 Experimental Results

For the experimental evaluation of our RSBST code synthesis, we have used a similar kind of setup, benchmark processors, and fault models considered in the last chapter. We have also used a 32-bit MIPS processor and a Leon3 processor model of SPARC V8 architecture with a 7-stage pipeline to be tested with the help of 10 behavioral fault representations shown in Table 3.1 of Chapter 3. The MIPS processor is synthesized

Table 4.2: *Specifications for the Proposed Automated Test Synthesis*

Specification	**Value**
Number of generations	400
Selection methodology	Tournament selection
Size (τ) of the tournament	2
Steady-state threshold	40
Evolutionary methodology	ES($\mu + \lambda$) approach
Fault coverage evaluation method	Behavioral fault evaluation

using 810 lines of VHDL code and the Leon3 processor has 5017 lines of VHDL code. The command-line options of ModelSim 10.5b simulator are used to execute the synthesized test programs on the faulty and non-faulty models of the processor. The fault simulation responses are extracted to evaluate the test programs and thereafter, the fittest solutions are selected. Generally, the synthesized test programs are of 250-300 lines of assembly code.

The functional and control components of the MIPS processor are tested in a software simulation environment of 270 fault models. To evaluate the test program, memory updates, contents of general purpose registers, and the primary outputs are extracted from the simulation responses using Python scripts and compared with the golden responses. The ($\mu + \lambda$) evolutionary-based test synthesizer is developed using an ANSI C implementation of 934 lines with three mutation and 1-point crossover operators.

The parameter values used for the proposed automated synthesis are shown in Table 4.2. The chromosomes of each generation are selected using a tournament selection operator where the tournament size (τ) is 2. The evolutionary core executes the test synthesis for 400 generations and terminates if there is no improvement for 40 generations, which is the steady-state threshold. For the conventional μGP and the greedy-based μGP, the size of the initial population (μ) is taken as 10 and the number of offsprings to be generated in each generation (λ) is taken as 5. But the faster convergence of RSBST could be exploited for achieving adequate coverage using a larger population size of test solutions. So, for RSBST code synthesis, we expand the search space by adopting μ as 20 and λ as 10. In the next subsection, we discuss how the observability comparator makes use of test program observability for faster test synthesis.

4.4.1 Observability Analysis of Test Programs

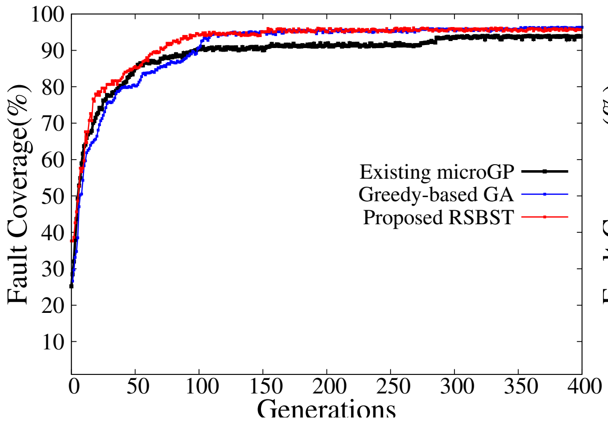
To identify the equally-observable test programs, the observability comparator is developed using ANSI C on the greedy-based evolutionary test synthesizer with 1309 lines of code for the MIPS processor. This module stores the contents of the observable destinations to identify the redundant test programs which could be internally evaluated. We observe 64 memory updates (each corresponds to a store instruction), contents of all 32 registers, and 2 primary outputs for the high-level logic simulation of test programs.

Let the set of memory updates be $M_i^j = \{m_1,m_2,m_3,\ldots,m_{64}\}$, contents of register locations be $R_i^j = \{r_1,r_2,r_3,\ldots,r_{32}\}$, and the primary outputs be $O_i^j = \{o_1,o_2\}$ after the execution of the test program solution P_i^j, which is the j^{th} individual of the population in the i^{th} generation. So ,the overall observability OBS_i^j for test program solution P_i^j becomes $\{m_1,m_2,m_3,\ldots,m_{64},\ r_1,r_2,r_3,\ldots,r_{32},\ o_1,o_2\}$, i.e., contents of 98 observable locations.

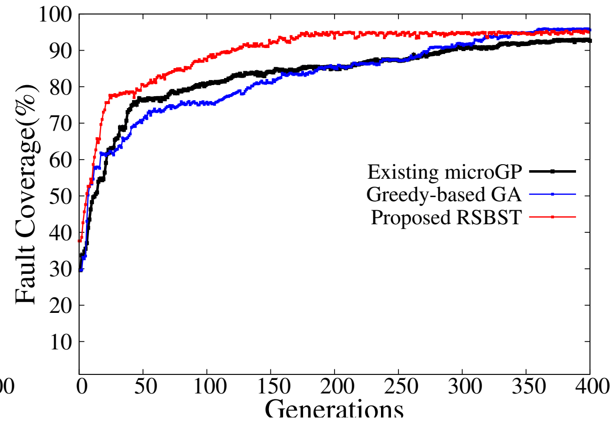
Now, the simulation repository is loaded with the contents of the observable locations ($OBS_{i-1}^{parent1(j)}$) of $P_{i-1}^{parent1(j)}$ and ($OBS_{i-1}^{parent2(j)}$) of $P_{i-1}^{parent2(j)}$, which are the the parent test programs of P_i^j. Let $OBS_{i-1}^{parent1(j)}$ be the set of values $\{m_1^{'},m_2^{'},m_3^{'},\ldots,\ m_{64}^{'},\ r_1^{'},r_2^{'},r_3^{'},\ldots,r_{32}^{'},\ o_1^{'},o_2^{'}\}$ and $OBS_{i-1}^{parent2(j)}$ be the values $\{m_1^{''},m_2^{''},m_3^{''},\ldots,\ m_{64}^{''},\ r_1^{''},r_2^{''},r_3^{''},\ldots,r_{32}^{''},\ o_1^{''},o_2^{''}\}$. If the observability OBS_i^j of test program P_i^j is equivalent to either the observability $OBS_{i-1}^{parent1(j)}$ of one of its parent $P_{i-1}^{parent1(j)}$ or the observability $OBS_{i-1}^{parent2(j)}$ of the other parent $P_{i-1}^{parent2(j)}$, the fault lists and fault coverages of the parent solution can be reused for the evaluation of P_i^j. We have applied and validated these equivalences on a MIPS processor and a Leon3 processor and the results are illustrated in the next subsection.

4.4.2 Case Studies for MIPS Processor and Leon3 Processor

A set of macros corresponding to the instructions of the MIPS processor are used for developing the constituents of the instruction library. For the MIPS processor, the enhancement in the development of the test synthesis using the proposed RSBST scheme is shown in Fig. 4.5(a). The average fault coverage of the conventional μGP scheme [2] with behavioral fault model achieves an adequate coverage (80-85%) after 50 generations, whereas the greedy-based GA proposed in the last chapter could only cover 85% of the faults after 75 generations. Eventually, 93.9% of the behavioral faults are detected by the



a) *Average Coverage for MIPS Processor*



b) *Average Coverage for Leon3 Processor*

Figure 4.5: *Average Fault Coverage for MIPS Processor and for Leon3 Processor over 400 Generations using 1) μGP [2] with Behavioral Fault Model 2) Greedy-based GA proposed in the last chapter 3) Proposed RSBST*

μGP approach [2] and 96.3% of faults are detected by the greedy-based GA. However, our RSBST code synthesis yields more than 85% of fault coverage before 50 generations and conclusively, carries out an adequate coverage of 96.1%.

For the Leon3 processor, the progress in the achieved fault coverage using the RSBST scheme is shown in Fig. 4.5(b). The conventional μGP approach [2] with the behavioral fault model, yields a fault coverage of 80-85% before 150 generations whereas the greedy-based GA accomplishes above 80% coverage only after 200 generations. Finally, μGP approach [2] could detect 92.9% of the faults and the greedy-based GA comes up with a fault coverage of 95.8%. However, the proposed RSBST code synthesis covers more than 85% of the possible faults before 100 generations and ends with a final fault coverage of 95.5%.

The test synthesis for MIPS processor was conducted module-by-module whereas monolithic test programs were generated for the Leon3 processor. The processor model describes the RTL model of the processor to be tested in the hardware description language VHDL, either in synthesizable or simulatable form. Here, the RTL model is subjected to module partitioning which is realized by breaking down the RTL design into several functional units and testing them separately. So, each processor module corresponds to a single hardware block and therefore, there are as many modules as the number of valid digital blocks in the processor model.

The coverage and test synthesis time for the five major modules of the MIPS processor

Table 4.3: *Achieved Coverage and Synthesis Time of MIPS Processor Modules*

MIPS Processor Module	Conventional μGP by G.Squillero [2]		Greedy GA proposed in the last chapter		Proposed RSBST	
	Coverage (%))	Synthesis Time (Hours)	Coverage (%))	Synthesis Time (Hours)	Coverage (%))	Synthesis Time (Hours)
ALU	100.00	24.50	100.00	33.73	100.00	18.07
PC	100.00	11.50	100.00	15.83	100.00	8.48
RF	96.67	15.00	96.67	20.65	96.67	11.06
ALU Control	90.24	21.50	94.87	29.63	94.27	15.86
Control Unit	83.83	49.50	90.32	68.16	90.12	36.53
Total	**93.90**	**122.00 Hrs**	**96.30**	**168.00 Hrs**	**96.10**	**90 Hrs**

is shown in Table 4.3. The overall test set constitute the test programs synthesized for the validation of each module. In the conventional μGP [2], achieved coverage (83.83%) was inefficient for the control unit module but the synthesis was reasonably fast (49.5 hours). So, the coverage of the control unit was improved towards 90.32% for the greedy coverage method, which encounters a longer test synthesis of 68.16 hours.

Some of the behavioral faults of the control unit could only be tested using rare sequences of instructions only. So, we make use of a larger solution space for the population of test individuals. Since the RSBST code synthesis is faster, this larger population of test individuals helps in developing instruction sequences that could detect harder behavioral faults for the control unit. As a result, test programs with coverage above 90% is synthesized for the control unit within 36.53 hrs using the proposed RSBST technique. For the remaining modules, minimum coverage of 94% is guaranteed with the overall evolutionary test synthesis terminates in 90 hours. Now, the amount of simulation responses reused for the chromosomes, bypassing the fault simulation, is discussed in the next subsection.

However, the fault coverage for ALU control and control unit achieved by RSBST technique is slightly smaller than that of Greedy GA method. This reduction is due to the high-level logic simulation of test programs in RSBST approach. In RSBST approach, we conduct a high-level logic simulation for a new offspring test program to check its reusability. If its logic simulation results are matching with that of any of its parents, the actual simulation results of parent test program are reused for the offspring test program. In those cases, actual fault simulation of the offspring is not conducted.

Table 4.4: *MIPS Processor - Achieved Coverage and Time of the 1) μGP [2] with Behavioral Fault Model 2) Greedy-based GA proposed in the last chapter 3) Proposed RSBST Method*

Framework	**Simulation Environment**	**Behavioral Fault Coverage**	**Test Synthesis Time**	**Chromosome Reuse**	**Remarks**
Conventional μGP by G.Squillero [2]	Modelsim Version 5.7a	93.9%	122 Hours	66.6%	Lesser fault coverage but test synthesis is faster.
Greedy GA proposed in the last chapter	GHDL	96.3%	168 Hours	66.6%	Improved fault coverage but test synthesis consumes huge time
Proposed RSBST	Modelsim Version 10.5b	96.1%	90 Hours	82.1%	Adequate fault coverage and faster test synthesis

Table 4.5: *Leon3 Processor - Achieved coverage and Time of the 1) μGP [2] with Behavioral Fault Model 2) Greedy-based GA proposed in the last chapter 3) Proposed RSBST Method*

Framework	**Simulation Environment**	**Behavioral Fault Coverage**	**Test Synthesis Time**	**Chromosome Reuse**	**Remarks**
Conventional μGP by G.Squillero [2]	Modelsim Version 5.7a	92.9%	142 Hours	66.6%	Lesser fault coverage but reasonable test synthesis time.
Greedy GA proposed in the last chapter	GHDL	95.8%	172 Hours	66.6%	Improved fault coverage but longer test synthesis
Proposed RSBST	Modelsim Version 10.5b	95.5%	98 Hours	80.8%	Adequate fault coverage and faster test synthesis

It may also happen that the offspring test program detects new faults but its logic simulation results may match with that of its parents. In these cases, new faults may get discarded. As most of the hard-to-detect faults are in the control components, it is quite likely that the faults in control unit and ALU control may left undetected in the RSBST scheme.

4.4.3 Chromosome Reusability of RSBST

In Table 4.4, the fault coverage, test synthesis time, and the amount of chromosome reuse are shown. The chromosome reuse refers to the percentage of chromosomes (test program solutions) reused throughout the test synthesis except for the first generation. For the first generation of test program solutions, $\mu + \lambda$ fault simulations must be performed to load the simulation results into an empty simulation repository. From the second generation, we investigate the scope of reusability and thereby reduce the test development time.

For the MIPS processor, the μGP approach [2] consumes 122 hours for the test synthesis and the greedy-based GA proposed in the last chapter takes 168 hours. The RSBST approach consumes only 90 hours, which is 46.4% faster than the greedy-based GA, and with adequate coverage of 96.1% as shown in Table 4.4. To synthesize monolithic test programs for the Leon3 processor, the μGP approach [2] takes 142 hours and the greedy-based GA consumes 172 hours. The proposed RSBST approach consumes only 98 hours, which is 43% faster than the greedy-based GA, with coverage of 95.5% as shown in Table 4.5.

The chromosome reusability is exploited to accelerate the convergence of the greedy-based GA. For the μGP and the greedy-based GA (proposed in the last chapter), the simulation responses of the selected chromosomes ($\mu = 10$) of each generation are adopted and reused directly from the parent chromosomes, which saves 66.6% of test synthesis time. For our proposed RSBST approach, the offspring chromosomes ($\lambda = 5$) are substituted by the equally-observable parent chromosomes along with the reuse of the selected chromosomes ($\mu = 10$). Eventually, the overall reusable chromosomes would mount up to 82.1% for the MIPS processor and 80.8% for the Leon3 processor.

4.5 Summary

In this work, a faster SBST synthesis of processor cores is employed using an accelerated greedy-based evolutionary method (RSBST), where the test programs that could detect the hard-to-test faults are developed. From the results, we could conclude that using a more comprehensive fault model, our strategy develops test solutions that could detect 96.1% of the testable behavioral faults of the MIPS processor in 90 hours and 95.5% that of the Leon3 processor in 98 hours. This affirms a chromosome (test program) reuse of 82.1% for the MIPS processor and 80.8% for the Leon3 processor.

A faster and profound test synthesis could be developed using the fragment-wise reusability of test programs. Even if the observability of 2 test programs are different, the identical, and data-independent code fragments (chunks) could be extracted from these test programs and reused. Also, the fault equivalence techniques could be used for reducing the volume of simulations and the test generation time.

Chapter 5

Automated Low-cost Compaction of SBST Programs

5.1 Basics of SBST Compaction

The self-test code employed for validating complicated processor functionalities would require a huge number of instructions. A large self-test code increases the test download time and test application time, which scales down the test performance [63]. In addition, smaller and efficient self-test codes are required for the online testing of safety-critical processors. To realize this, an effective self-test code optimization process (Phase D of the SBST procedure shown in Fig. 1.3 of Chapter 1) is conducted after the self-test synthesis phase (Phase C of the SBST procedure shown in Fig. 1.3 of Chapter 1).

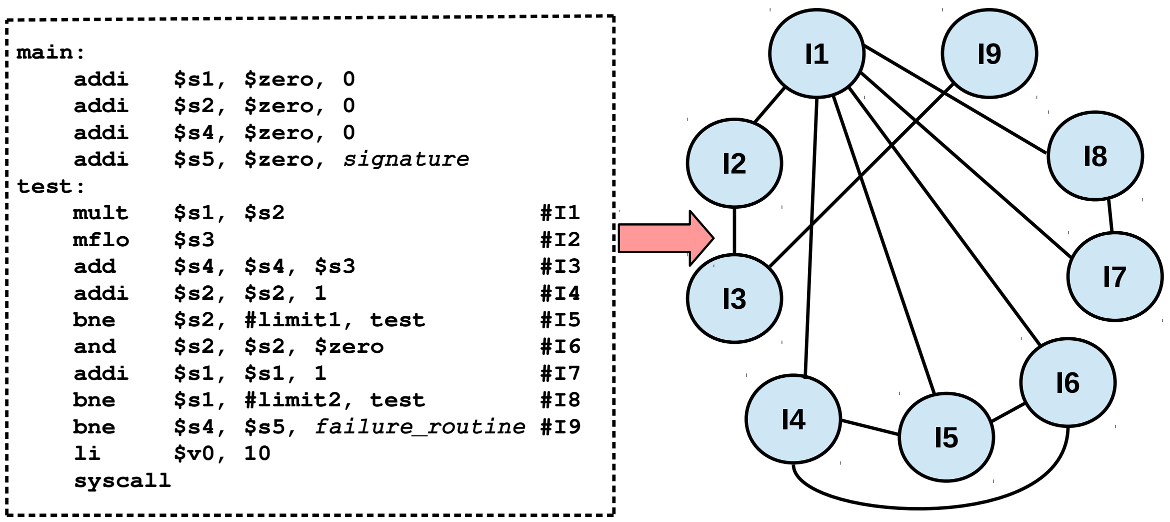
The recent advancements in the domain of test code optimization [4, 65] focus on the reduction of test execution time and test code compression. These techniques aim to remove the instructions which do not contribute towards the overall fault coverage. To efficiently remove these redundant instructions, the test code must be subjected to a large number of fault simulations, which is the overall computational cost of test optimization. So, the primary objective of any test code optimization technique is to figure out an alternate redundant instruction elimination method which can bypass as many number of fault simulations as possible. However, these techniques must guarantee that the fault coverage of the original test program would be preserved after the test code compaction.

The most significant execution time reduction techniques [4] are of two types: 1) redundant instruction removal techniques, which are computationally intensive but yields a high amount of compaction, and 2) instruction block restoration techniques, which yield lesser compaction with lesser computational effort. So, for a better tradeoff between computation cost and the amount of compaction, we propose an enhanced two-stage compaction method. In the first stage, the test program is preprocessed using a dependency graph-based independent instruction removal technique. In this technique, the connected components of a data dependency graph are discovered for the SBST test program instructions. These connected components are used to identify the independent instruction sequences that do not cover any fault exclusively. Eventually, these independent instruction sets are removed from the original test program yielding an adequate amount of compaction.

Following the elimination of independent instructions, we employ a faster, top to bottom instruction restoration technique using logic simulation of the test programs in the second stage of test compaction. In this technique, test programs are divided into equally sized blocks and the contribution of each block towards fault coverage is evaluated. The instruction blocks are removed and restored from top to bottom for better compaction. When a block is restored, its upper blocks are replaced with few instructions that reinstate the initial state for that block. The initial state of a block is calculated using logic simulation of the blocks above it. So, this could reduce the cost of fault simulations, enabling faster and high test compaction. In the next section, we discuss the first stage of our proposed work where redundant groups of independent instructions are identified and removed.

5.2 Redundant Instruction Group Removal Using Data Dependency Graphs

In this approach, we break down the test program into groups and eliminate the redundant groups which do not contribute to the overall fault coverage. To achieve this, the read-write dependencies between the test code instructions are analyzed. If instructions of the test program are tightly coupled to each other in terms of dependencies, the grouping would be difficult. This is because some of the processor faults could be detected only with the help of multiple dependent instructions. So, if dependent instructions are classified into separate groups for compaction, these faults may go undetected, which would reduce the overall fault coverage. But for the compaction of test programs with loose coupling between instructions, several independent groups can be identified,



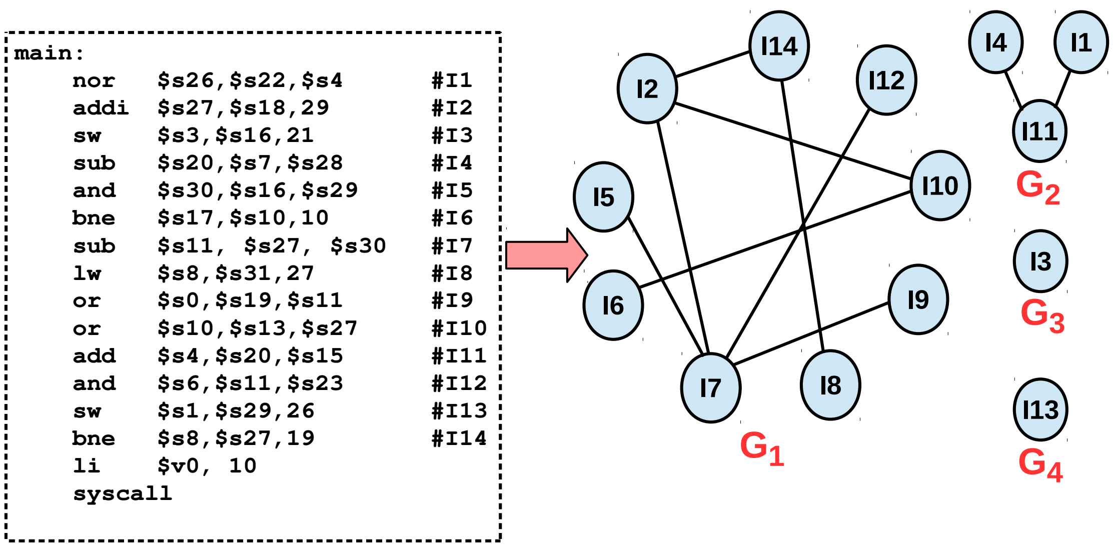
```
main:
    addi   $s1, $zero, 0
    addi   $s2, $zero, 0
    addi   $s4, $zero, 0
    addi   $s5, $zero, signature
test:
    mult   $s1, $s2                    #I1
    mflo   $s3                         #I2
    add    $s4, $s4, $s3               #I3
    addi   $s2, $s2, 1                 #I4
    bne    $s2, #limit1, test          #I5
    and    $s2, $s2, $zero             #I6
    addi   $s1, $s1, 1                 #I7
    bne    $s1, #limit2, test          #I8
    bne    $s4, $s5, failure_routine #I9
    li     $v0, 10
    syscall
```

Figure 5.1: *Data Dependency Graph with Single Connected Component*

maintaining the fault coverage. Further, these groups can be evaluated to eliminate the instructions of redundant groups among them. However, the effectiveness of our approach lies in the fact that the dependencies between instructions of test solutions developed using evolutionary test synthesis would be loose due to the random nature of the evolutionary approach.

A data dependency graph representation of instruction sequences depicts flow dependence, anti dependence, and output dependence between the operands of instructions. In Fig. 5.1, an undirected data dependency graph corresponds to a test program for the multiplier module of a 32-bit MIPS processor is shown. In this test program, every consecutive integer value of a set, 0 to *limit*1, is multiplied with the consecutive integer values of another set, 0 to *limit*2, and the sum of these product values is compared with the expected response signature. The admissible region of a test program is the middle section where the instructions can be reordered or removed. In the test program shown in Fig. 5.1, 9 instructions ($I1$ to $I9$) lies in the admissible region and its undirected data dependency graph is a connected component, i.e., each instruction in the admissible region holds dependencies with at least one of the other instructions. These inter-instruction dependencies restrict the extraction and removal of redundant dead codes, which do not contribute fault coverage.

In the example shown in Fig. 5.2, the data dependencies between the instructions of a test program developed using automated test synthesis is shown. An adaptive evolu-



```
main:
    nor   $s26,$s22,$s4       #I1
    addi  $s27,$s18,29        #I2
    sw    $s3,$s16,21         #I3
    sub   $s20,$s7,$s28       #I4
    and   $s30,$s16,$s29      #I5
    bne   $s17,$s10,10        #I6
    sub   $s11, $s27, $s30    #I7
    lw    $s8,$s31,27         #I8
    or    $s0,$s19,$s11       #I9
    or    $s10,$s13,$s27      #I10
    add   $s4,$s20,$s15       #I11
    and   $s6,$s11,$s23       #I12
    sw    $s1,$s29,26         #I13
    bne   $s8,$s27,19         #I14
    li    $v0, 10
    syscall
```

Figure 5.2: *Data Dependancy graph with Multiple Connected Components*

tionary method μGP is used for the automation of self-test development. Although the fault coverage is optimized, the size and execution time of evolutionary test program solutions could be large due to random instruction selection. During the evolutionary self-test generation, the instructions and operands are selected randomly. So, the dependencies between instructions would be loose and thus, more redundant instructions could be observed. These instructions could be removed to minimize the SBST code size and to reduce the self-test execution time.

The dependencies of the test program shown in Fig. 5.2, which has 14 instructions ($I1$ to $I14$) in the admissible region, are represented using four connected components G_1 to G_4. Each subgraph represents a group of instructions which are independent of any of the instructions in the other subgraphs. The subgraph G_1 has 9 instructions, subgraph G_2 has 3 instructions, subgraph G_3 has one instruction, and subgraph G_4 has one instruction. The 9 instructions of subgraph G_1 are independent of the instructions of subgraph G_2, subgraph G_3, and subgraph G_4. The fault coverage of subgraph G_1 is evaluated by simulating the instruction sequence of subgraph G_1 separately. Likewise, the instruction sequences of subgraphs G_2, G_3, and G_4 are simulated to obtain their individual coverages.

To remove the redundant instructions from the actual test program T, we heuristically discover the largest dependency subgraphs with completely redundant fault lists. Thereafter, all the instruction sequences of these redundant subgraphs are removed from

ALGORITHM 4: Stage 1: Independent Instruction Removal

```
Input: Test Program $P$ with $u$ instructions represented by a dependency graph
       $G$ with $v$ connected components $G_1, G_2, G_3, \cdots, G_v$;
Output: Compacted test program $P'$;
1 for $i \leftarrow 1$ to $v$ do
2     Evaluate the set of faults covered $FC_{G_i}$ by the instructions of $G_i$ using fault
      simulation;
3 for largest connected component $G_i$ in $G$ do
4     if $FC_{G_i}$ is a subset of $\bigcup_{i=1}^{v} FC_{G_j}$, where $j \neq i$ then
5         Remove the instructions of $G_i$ from T to form $P'$, i.e., $P' = P - G_i$ ;
6     Remove $G_i$ from $G$ and decrement $v$ by one;
```

T. The stepwise illustration of dependency subgraph removal technique is shown in Algorithm 4. Initially, the dependencies between instructions of the input test program P are identified to construct a dependency graph G. Now, we evaluate the list of covered faults of the connected components $G_1, G_2, G_3, \cdots, G_v$ of graph G (Step 2). Among this set of connected components, the largest subgraph G_i is selected and its redundancy, in terms of fault coverage, is checked in Step 4. If the subgraph is redundant, its instructions are removed from the original test program, as shown in Step 5. Further, the next largest connected component is selected and this process (Steps 3-6) goes on until the redundancy of the smallest connected component is analyzed.

For the example of the test program shown in Fig. 5.2, the instructions of the largest graph G_1 could be removed from T if the fault list FC_{G_1} of graph G_1 is a subset of the set union of FC_{G_2}, FC_{G_3}, and FC_{G_4}. If not, the next largest connected component (G_2) is selected for redundancy check and so on. Let us say, the self-test code has u instructions which could be represented using v ($<< u$) different data dependency subgraphs. As every instruction of each subgraph is simulated only for a single time, the computational effort for the removal of redundant instructions using this method would be insignificant. But the amount of compaction would not be consistently large as instruction-wise redundancy is not evaluated. Towards achieving a low-cost, instruction-wise redundancy checking technique, we introduce the second stage of our proposed work, which is an enhanced top to bottom instruction restoration policy using high-level logic simulations. This technique deals with the compaction versus cost tradeoff efficiently and is discussed in the next section.

5.3 Enhanced Instruction Restoration Method

5.3.1 Top to Bottom Compaction Policy

In this method, the test program left after the first stage of compaction is divided into equally sized blocks. Initially, a set of *reliant faults*, which are the faults detected by an instruction in the test program for the first time, is recorded for each instruction. Now, an instruction block is selected and removed from the test program. Thereafter, we conduct the fault simulation of the remaining test program for the faults that may get undetected due to the block removal. In other words, the fault simulation is conducted only for the reliant faults of the instructions in the currently restoring block and the following blocks. If these faults are detected by the remaining test program, we permanently eliminate the instruction block from the test program. Otherwise, instructions of the removed block are restored one at a time and the test program is fault simulated again until it detects all the reliant faults of that block and the blocks below. Each block of the test program is compacted using this restoration method.

Block-wise restoration could be either in 1) bottom to top or 2) top to bottom order. In the existing bottom to top restoration policy [4], the blocks are selected for restoration from the last block of the test code, i.e., after the compaction (using restoration) of a block, its preceding block will be selected. For this policy, the restoration of the top blocks of the program would have lesser cost for fault simulations because the blocks below would already be compacted. Thus, if the blocks are selected from the last one, the cost of compaction could be reduced.

However, a few critical instructions in the lower blocks may get eliminated when bottom to top restoration policy is employed. Instruction is critical if it could replace multiple instructions of its upper blocks in the test program. During bottom to top restoration, a critical instruction may get removed when all the reliant faults of its block and the blocks below are covered by the instructions of the lower blocks, thereby reducing the spectrum of compaction.

When top to bottom instruction removal policy is employed, blocks of instructions are removed and restored starting from the first block of the program. In this approach, the redundancy of block instructions is evaluated by selecting and inserting instructions of each block one by one, which is similar to the bottom to top method. After the elimination of this redundant instruction, each of its reliant faults is added to the set of reliant faults of the instruction in the lower blocks that detect it. In other words, a critical instruction in the lower blocks would be updated with multiple reliant faults of the instructions in the upper blocks and thus, the number of reliant faults of a critical instruction will be very high. So, it would be difficult for the lower blocks of instructions to replace a preceding critical instruction which detects many reliant faults. To summarize, the top to bottom instruction removal policy helps in conserving the critical instructions, and eventually leads to better compaction but have a higher cost as compared with bottom to top policy.

5.3.2 Restoration Using High-level Logic Simulation

To reduce the computational cost of top to bottom restoration policy, we replace computationally intensive fault simulations with faster and effortless high-level logic simulations. In each step of block restoration, the existing **A1xx** compaction technique [4] evaluates the test program using high-cost fault simulations for the faults that could become undetected due to the removal of that block. A high-level logic simulation of previous instructions would help in reducing the computational effort required for the fault simulations of the whole test program during the restoration of every instruction.

In our high-level logic simulation, we model the syntax and encoding of every instruction of ISA using high-level ANSI C programming language. Each instruction operation is logically simulated using high-level functionalities. Also, the observable locations, such as updated memory locations and updated register locations, are allotted with a definite amount of memory. Only those memory locations, which are updated by the execution of the *store* instructions in the test program, are observed, whereas every updated register value is observed. The instruction operations are realized using high-level procedures developed for each opcode. The operations are performed on the source operands that modify the observable locations correspond to the destination operands.

In this method, the initial state of a restoring block is evaluated using the high-level logic simulation of the previous instructions. This initial state comprises the contents of the observable locations, which are modified after the execution of all previous instruc-

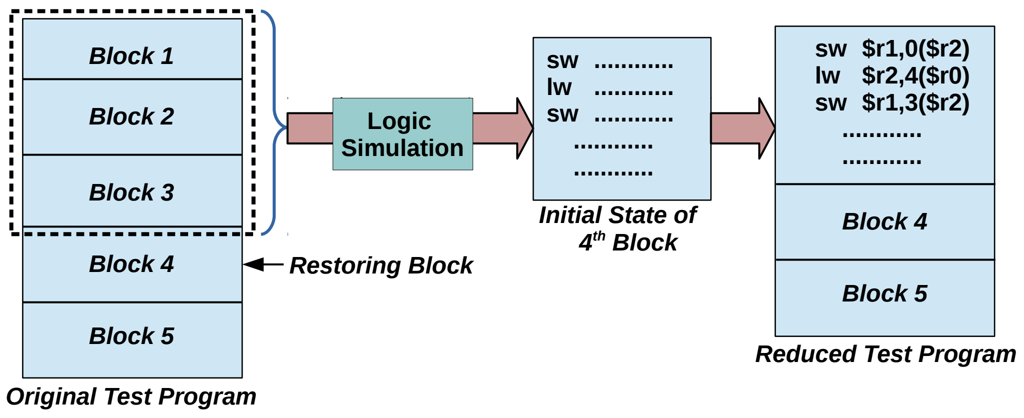


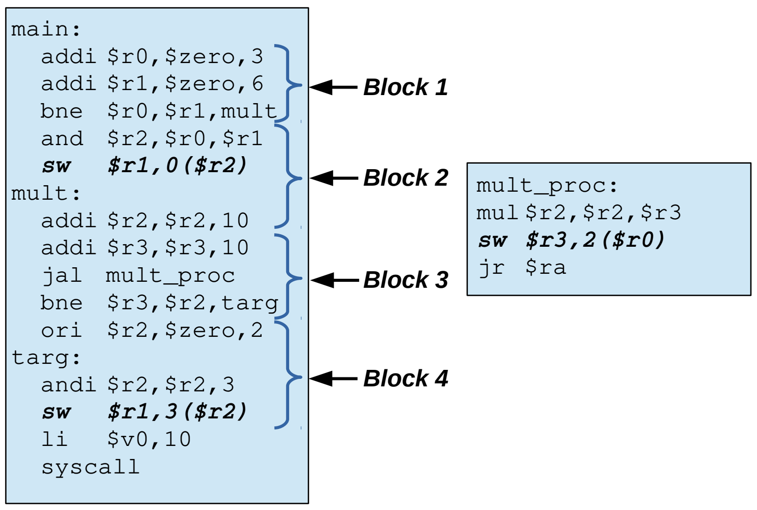
Figure 5.3: *Example of Reducing Test Program Using Logic Simulation*

tions above the restoring block. These updated observable locations are evaluated using logic simulations and are compiled to form the initial state for the block restoration. Now, the blocks of instructions above the currently restoring block is replaced with the contents of the initial state. These contents are retrieved with the help of a few *load* and *store* instructions added before the restoring block. These added instructions change the contents of the observable locations in a similar manner the logic simulations would modify. If possible, these initial states can be restored from previous simulations, which would be less costly.

An example of reducing the test program using logic simulation is shown in Fig. 5.3. In this example, the original test program is divided into 5 blocks of instructions. During the restoration of the 4^{th} block, the top 3 blocks are subjected to logic simulation to develop the initial state of the 4^{th} block. Finally, this initial state replaces the top 3 blocks of the original test program to yield the reduced test program.

Let m be the number of updated registers and n be the number of modified memory locations of the processor. Following the logic simulation of previous instructions of the restoring block, the contents of the observable locations, which are the set of updated register values $Reg = \{Reg_1, Reg_2, Reg_3, \ldots, Reg_m\}$ and the set of updated memory values $Mem = \{Mem_1, Mem_2, Mem_3, \ldots, Mem_n\}$ are logged. Eventually, the observability $OBS = \{Reg, Mem\}$ is evaluated to learn the initial state of the restoring block.

During logic simulation, the modified memory values correspond to each store instruction is recorded. An updated memory value of Mem_i is depicted using an $\langle addr, val\rangle$ representation, which indicates that the store instruction modifies the memory address



```
main:
  addi $r0,$zero,3
  addi $r1,$zero,6
  bne  $r0,$r1,mult
  and  $r2,$r0,$r1
  sw   $r1,0($r2)
mult:
  addi $r2,$r2,10
  addi $r3,$r3,10
  jal  mult_proc
  bne  $r3,$r2,targ
  ori  $r2,$zero,2
targ:
  andi $r2,$r2,3
  sw   $r1,3($r2)
  li   $v0,10
  syscall
```

```
mult_proc:
mul $r2,$r2,$r3
sw  $r3,2($r0)
jr  $ra
```

Figure 5.4: *Blocks of SBST Test Program in the Admissible Region*

location *addr* with a value *val*. For example, the test program shown in Fig. 5.4 has 3 store instructions with memory update values $\langle \$r2, \$r1\rangle$, $\langle \$r2{+}3, \$r1\rangle$, and $\langle \$r0{+}2, \$r3\rangle$.

This test solution has 12 instructions, divided into four blocks of size 3, in its admissible region, which excludes the macro definitions and procedure definitions. Now, the initial state for each block is calculated using the memory and register values updated after the logic simulation of previous instructions. For example, the initial state for block 3 would be $OBS = \{\langle\ 2{,}6\ \rangle, r0, r1, r2\}$, i.e., the value of memory update $\langle \$r2, \$r1\rangle$ due to the store instruction in block 2 becomes $\langle\ 2{,}6\ \rangle$, and the registers $r0, r1, r2$ are updated since these registers are the destination registers in at least one of the instructions of block1 and block 2.

To reinstate the initial state of a block, its upper blocks are replaced with few *load* and *store* instructions. For the restoration of the fourth block, every instruction of first, second, and third blocks are logically simulated and the updates of observable locations are recorded as $OBS = \{\langle\ 2{,}6\ \rangle, r0, r1, r2, r3\}$. Now, we place one store instruction for the memory update $\langle\ 2{,}6\ \rangle$ and four load instructions for the updates in registers $r0, r1, r2, r3$ before block 4. So, we replace all the 9 instructions preceding the fourth block with these 5 instructions that produce the initial state for the 4^{th} block.

The steps of the enhanced restoration method are described in Algorithm 5. Initially, the test program P is divided into blocks $B_1, B_2, B_3, \cdots, B_\delta$, which are subjected to elimination and restoration. From the analysis in Section 5.3.1, we have observed that

ALGORITHM 5: Stage 2: Enhanced Test Program Restoration

```
Input: Test Program $P$ with the remaining $\sigma$ instructions after the first stage of
       compaction divided into $\delta$ blocks $B_1, B_2, B_3, \cdots, B_\delta$;
       Set of reliant faults $FR_1, FR_2, FR_3, \cdots, FR_\delta$ of each block;
       Initial states $I_{B_1}, I_{B_2}, I_{B_3}, \cdots, I_{B_\delta}$ for each bock;
Output: Compacted test program $P^{'}$;
1  for $i \leftarrow 1$ to $\delta$ do
2      $P^{'} = P$ - $B_i$ ;
3      $P^{''} = P^{'}$ - {all the blocks from start till $B_{i-1}$};
4      Logic simulate the upper blocks of $P^{'}$ till $B_{i-1}$ to yield the initial state $I_{B_i}$ for
        block $B_i$ ;
5      Reinstate the initial state $I_{B_i}$ with corresponding load and store instructions
        in the beginning of $P^{''}$;
6      Fault simulate $P^{''}$ for the faults in $FR_i$,$FR_{i+1}$,...,$FR_\delta$;
7      if all the faults in $FR_i$,$FR_{i+1}$,...,$FR_\delta$ are covered by $P^{''}$ then
8          Remove the faults in $FR_i$ that are detected first by the eliminated
            instructions;
9          Update sets from $FR_{i+1}$ till $FR_\delta$ correspond to the instructions that
            detect the faults removed from $FR_i$;
10         $P = P^{'}$;
11     else
12         Select an unrestored instruction $s$ randomly from $B_i$;
13         Restore $s$ in $P^{'}$, i.e., $P^{'} = P^{'} \bigcup s$;
14         go to step 3;
```

the top to bottom restoration policy gives better compaction than the bottom to top policy. So, the top blocks are selected first for restoration as shown in Step 1. In Step 2, we construct another test program $P^{'}$ by writing all the blocks of P, except B_1, into $P^{'}$. Further, a test program $P^{''}$ is constructed by writing all the blocks of $P^{'}$ from the top block until the block B_{i-1} (Step 3).

In Step 4, the top $(i-1)$ blocks of test program $P^{'}$ is logically simulated to yield the initial state I_{B_i} for block B_i. This initial state is applied to the test program $P^{''}$ with a set of load and store instructions correspond to each modified observable location (Step 5). Now, In Step 6, the test program $P^{''}$ is subjected to fault simulation only for the sets of reliant faults FR_i,FR_{i+1},...,FR_δ, which comprises the faults that could become undetected when the block B_i is removed.

If every fault of FR_i,FR_{i+1},...,FR_δ is covered by the test program P'', the remaining instructions of the block B_i are removed. In this case, we also remove the faults from the set FR_i that are detected by the eliminated instructions (Step 8) for the first time. Subsequently, in Step 9, the sets from FR_{i+1} till FR_δ are updated with respect to the instructions that detect the faults removed from FR_i. Later, the test program P', using which the test program P'' is constructed, becomes the new original test program P, as shown in Step 10. If the test program P'' does not detect all the faults in FR_i,FR_{i+1},...,FR_δ, an instruction is selected randomly from the instructions in block B_i and is restored into the test program P' (Steps 12-13). In Step 14, the algorithm jumps back to Step 3, where the test program P'' is created from the test program P' with a newly restored instruction, and this process continues until the last block B_δ is selected and compacted.

Finally, this enhanced top to bottom restoration optimize the test codes with higher compaction and faster logic simulations. In the next section, we analyze the efficacy of the proposed method and compare the results with that of the existing compaction techniques using the experiments on a group of test programs synthesized using evolutionary techniques.

5.4 Experimental Results

The effectiveness of these two compaction methods (subsections 5.2 and 5.3) are corroborated with the results in comparison with the results of existing **A0** and **A1xx** algorithms [4]. We have automated the SBST synthesis using μGP evolutionary techniques [2, 41, 53] and developed a monolithic code of 294 instructions that yield 96.3% fault coverage for a 32-bit MIPS processor model. The test programs are evaluated on a ModelSim 10.5b simulator platform, and the proposed 2-stage compression module is developed with the help of 1216 lines of ANSI C program.

In our experiments, the performance of compaction is measured in terms of compaction ratio of time (CRT), compaction ratio of size (CRS), and $Cost$, where

- CRT is the ratio between execution times of the optimized test code and the

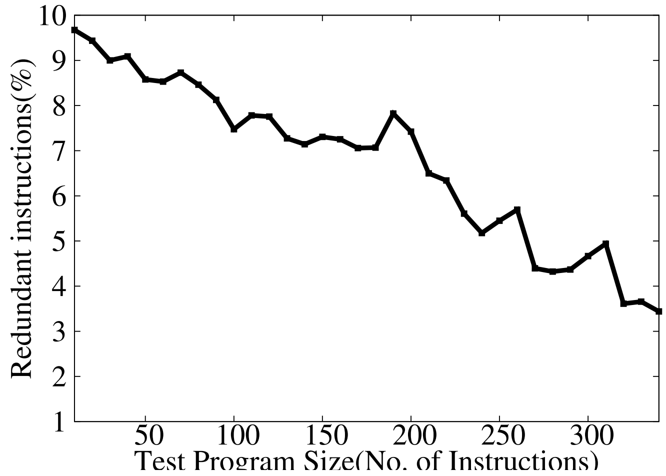


Figure 5.5: *Amount of Compaction for Different Test Programs*

Table 5.1: *Results of Compaction for Restoration Methods*

Method	**CRS**	**CRT**	**Cost**
Bottom to top Restoration [4]	0.85	0.855	159
Top to bottom Restoration	0.84	0.84	164

original test code, and

- *CRS* is the ratio between sizes of the optimized test code and the original test code, and
- *Cost* is the ratio between the time consumed for compaction and the time consumed for a single full simulation of the original test code.

If the amount of compaction is high, *CRS* and *CRT* would be lesser.

In Fig. 5.5, we depict the first stage of optimization with respect to test programs with various sizes. The amount of dependency graph-based optimization is estimated in terms of the percentage of redundant instructions identified in the test program. This diagram shows that the scope of this low-cost compaction reduces as the test size increases. For example, the smaller test programs with less than 50 instructions have more than 9% of redundant instructions, i.e., *CRS* is less than 0.91. But larger test programs with size more than 310 instructions have less than 4% of redundant instructions, i.e., *CRS* is more than 0.96. This decline in compaction is due to the increased dependencies between the randomly constructed instructions of larger test programs. The dependencies

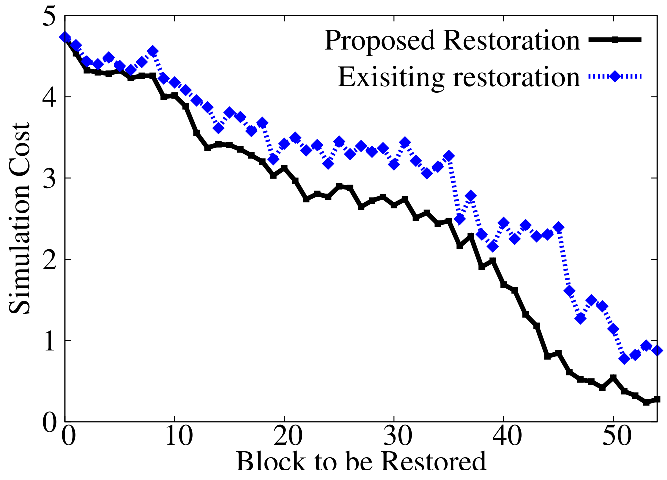


Figure 5.6: *Block-wise Simulation Cost for Proposed Restoration and Existing Restoration [4] techniques on a Test Code with 294 Instructions*

Table 5.2: *Results of Compaction for Existing and Proposed Methods on a Test Code with 294 Instructions*

Method	**Size compaction (No. of Instructions)**	**CRS**	**Reduced Time (Clock Cycles)**	**CRT**	**Cost**
Instruction Removal (**A0**) [4]	294 to 229	0.778	2820 to 2200	0.780	197.00
Instruction Restoration (**A1xx**) [4]	294 to 250	0.850	2820 to 2411	0.855	159.00
Proposed Method Stage 1	294 to 283	0.965	2820 to 2723	0.965	1.33
Proposed Method Stage 2	283 to 238	0.840	2723 to 2290	0.840	141.00
Overall Stage 1 + Stage 2	294 to 238	0.810	2820 to 2290	0.812	142.33

increase for randomly generated larger test programs because the number of registers and memory locations are limited.

An enhanced compaction technique is discussed in the second stage of our proposed technique. In the initial phase of this approach, we study the pros and cons of the existing bottom to top instruction restoration technique [4] and compare them with that of the proposed top to bottom instruction restoration technique (section 5.3.1), as shown in Table 5.1. In bottom to top restoration, the compaction ratios are equal to or above 0.85 with a cost of 159, whereas top to bottom technique yields more compaction (< 0.85) with high computational cost of 164.

As higher compaction is achieved by top to bottom restoration, we adopt this technique and further, seek to reduce its cost using logic simulations of preceding blocks proposed in section 5.3.2. The costs of existing **A1xx** restoration technique [4] and proposed restoration technique are shown in Fig. 5.6. Both techniques consume equivalent cost for the restoration of the top 5 blocks and further, the costs of both techniques reduce in restoring the lower blocks. However, the logic simulation of preceding blocks reduces the cost of the proposed technique more rapidly than that of the existing technique [4]. Finally, the cost of existing technique remains above 0.8, whereas the cost of proposed technique goes below 0.5 for the last 4 blocks. Thus, the introduction of logic simulations leads to significant cost reduction of top to bottom restoration techniques.

The elaborated results pertaining to the cost and the compaction rate of existing removal-restoration techniques [4] and the proposed 2-stage techniques are shown in Table 5.2. The cost of **A0** is, however, optimally minimal ($CRT = 0.778$, $CRS = 0.78$) because thorough instruction-wise redundancy check is conducted. But the cost for compaction is undesirably larger (197). The **A1xx** technique [4] carries out a less-intensive block-wise compaction and realizes cost reduction (159) with adequate compaction rate ($CRT = 0.85$, $CRS = 0.855$). The 2 stage of our proposed technique further reduce the cost to 142.33, besides achieving better compaction ($CRT = 0.81$, $CRS = 0.812$).

Following the first preprocessing stage, the test program is compressed with CRS and CRT values as 0.965. This minimization eliminates the redundant and independent groups of test codes. For a larger test code with 294 instructions, the instruction dependencies would be stronger, which restricts the grouping and elimination of instructions in terms of dependencies. So, the degree of compaction yielded by this method is lesser ($CRS = CRT = 0.965$). However, the computational cost is insignificant (1.33) in this stage, where a single fault simulation is consumed for the compaction, in addition to the cost of computing the redundant subgraphs (0.33).

The execution of the second stage requires a cost of 141 with CRS and CRT values as 0.84. Altogether, the overall CRS is reduced to 0.81 and the overall CRT is reduced to 0.812 with a reduced overall computation cost of 142.33. So, this approach earns us a good amount of compaction with reduced computational complexity compared with the existing **A0** and **A1xx** techniques [4].

5.5 Summary

To compress larger SBST programs developed for complex processors, a two-stage algorithm is proposed and validated in this method. These stages are enhancements of existing instruction removal and instruction restoration techniques. In the first stage, the programs are preprocessed by removing the groups of independent instructions which are redundant in terms of fault coverage. In the second stage, high-cost top to bottom restoration technique is employed for better compaction than bottom to top restoration. Further, the cost of top to bottom restoration is reduced by high-level logic simulations of the preceding blocks of the currently restoring block. So, the large SBST programs are compacted using dedicated, low-cost optimization techniques which also grants a good amount of compaction.

In fact, we have considered the development of self-test codes with a reasonable tradeoff between test execution time and cost for compaction in this chapter. This self-test quality enhancement phase is carried out after the test synthesis phase. In the next chapter, we propose a high-reliability online testing method, where smaller and efficient self-test codes are searched and identified during the test synthesis phase itself. The selected self-test codes guarantee a better tradeoff between test execution time and fault coverage.

Chapter 6

Application of Fragments of SBST Programs for Online Testing

In extreme online operating conditions of the processor, intermittent faults, which are temporary in nature, may momentarily appear and disappear. These intermittent faults could damage the gate-level logic of the processor and may eventually turn into permanent faults. To achieve high self-test quality, i.e., high fault coverage, the self-test codes detect these intermittent faults too. If fault detection latency, which is the time gap between fault occurrence and its detection, is large for the self-test code, intermittent faults not get detected efficiently and as a result, the fault coverage will be lesser.

Along with the real-time, periodic mission tasks, a self-test task is also executed in regular intervals for the online testing of the processor. If the execution interval between these self-test tasks is large, the fault detection latency also would be longer. Generally, large self-test codes are executed on the processor to test the complicated functionalities processor. But the execution intervals between such large test codes will be large, resulting in undesirably high fault detection latency. In other words, even if the fault coverage of large self-test codes are high, due to large intervals between their executions, many intermittent faults incur large detection latency (e.g., faults that occur just after the completion of execution of the test code).

If smaller test codes are applied, individual fault coverage would be less for these fragments. But as these code fragments are executed with smaller test periods, they effectively validate the processor for intermittent faults with lesser fault detection latency. Also, multiple efficient small test code fragments, whose overall coverage is almost similar to that of the larger test code, can replace the larger test code for the online testing

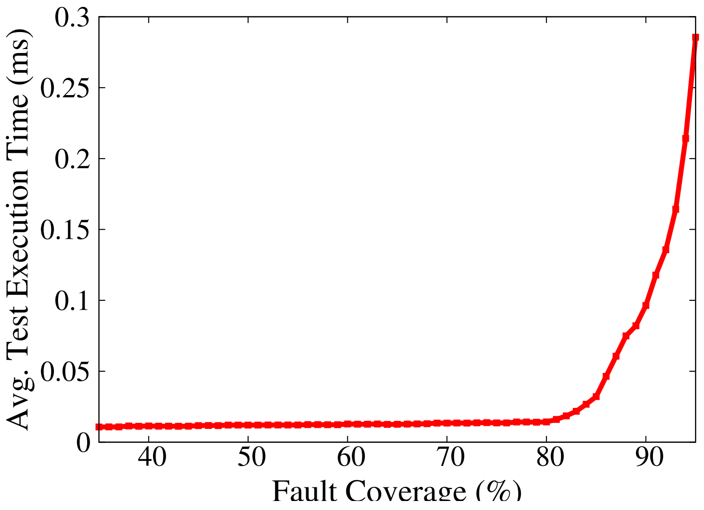


Figure 6.1: *Avg. Execution Time for Different Groups of Fragments on 100 MHz MIPS Processor Model*

of the processor. In the previous chapter, we have reduced the test program with the help of compaction techniques. But this amount of compaction or reduction would not be enough in order to generate test programs that can effectively test processor for intermittent faults.

The average execution times of synthesized test codes during evolutionary test synthesis with equal fault coverage is shown in Fig. 6.1. This behavior can be observed for every set of test codes which are developed during the evolutionary test synthesis. These test codes, which are developed during the evolutionary test synthesis, are stored exclusively for our online testing technique. The execution times of test codes with coverage between 35% and 80% are in the range (0.01, 0.014), whereas the execution times of FTPs with coverage between 81% and 95% are in the range (0.016, 0.285). The execution time increases gradually from the test codes with 35% coverage to the test codes with 80% coverage. But the change in execution time is significant from the test codes with 80% coverage to the test codes with 95% coverage.

The test codes with coverage between 75% and 85% have considerably lesser execution time and thus, lesser fault detection latency, compared with the test codes with coverage above 85%. Also, these test codes maintain adequate test quality (75% to 85%). So, a reliability analysis of the test codes in this range must be conducted to identify the optimal test codes that provide the best tradeoff between fault detection latency and coverage.

In our approach, efficient, reliable fragments of a self-test code are discovered by means of selection of test code fragments that maintains the online test quality (fault coverage) and minimizes the fault detection latency. To identify the optimal fragments, we evaluate the reliability of self-test fragments of different sizes. However, these fragments can be executed as self-test sub-tasks only if they satisfy certain scheduling criteria; the modified response time of each mission task and the modified overall utilization following the inclusion of these self-test sub-tasks must not exceed their corresponding limit. If these conditions are satisfied, the self-test fragments are executed as sub-tasks, scheduled with appropriate execution windows between the execution of mission tasks. Also, the overall coverage of these fragments must be nearly equivalent to the coverage of the unfragmented full test program.

As the intermittent faults occur irregularly at the same location, the self-test codes must be regularly executed with a short test period to efficiently trace them. D. Gizopoulos [98] suggested that the self-test quality could be improved when the self-test tasks are executed with larger execution time and enhanced self-test utilization. But the proposed techniques in [98] will increase the self-test period to achieve maximum utilization. If the self-test period is increased, fault detection latency also would increase, and subsequently, some of the instantaneous intermittent faults may be left undetected. If an intermittent fault occurs just after a large test period, fault detection latency will be higher, which may cause system errors. The tradeoff between test utilization and fault detection latency in [98] could be dealt only if efficient, small chunks of SBST codes are executed frequently between the mission tasks, i.e., smaller, coverage-efficient test programs are intermittently executed during a self-test period to reduce the fault detection latency.

In our approach, shorter, reliable SBST test code fragments are discovered and executed intermittently in a self-test period to immediately detect the intermittent faults with minimal fault detection latency. But smaller test codes might have lesser fault coverage, which could leave some of the intermittent faults undetected. So, the test fragment synthesis must consider both fault detection latency and test quality (fault coverage) in developing reliable fragments. These minimal code fragments are applied in appropriate execution windows between the execution of the mission tasks.

We summarize the problem as follows: *smaller SBST self-test codes with smaller* *fault detection latency realize rapid detection and recovery of intermittent faults. But these minimal test programs could have less reliability due to low fault coverage. Larger test programs would detect most of the faults but reliability will be lesser due to high fault detection latency. To deal with this trade-off, optimal and reliable set of fragments must be discovered with significant self-test quality (coverage) and minimal fault detection latency.*

So, in this chapter, a high-reliable online fault detection model to test a low-cost, real-time embedded processor for the intermittent faults is demonstrated. The contributions of this fragmented testing approach are:

- The instruction sequences of a larger SBST code is prudently replaced with smaller, coverage-efficient, online Fragment of Test Programs (FTPs) to be executed intermittently during a test period to detect the intermittent faults with minimal fault detection latency and good test quality (fault coverage). To meet this, we evaluate the reliability of the system with respect to different fragment sizes. From the maximum permissible fragment size, minimum permissible test execution window size is also assessed.

- We demonstrate the reliability-based FTP selection using a set of 12 mission task workloads on a 100 MHz MIPS processor model.

- A fault tolerant self-test schedule is proposed to deal with the challenges in the detection and recovery for the intermittent faults.

In the next section, the basic definitions of schedulability, system reliability, and recovery models are discussed for the demonstration of the proposed synthesis of online SBST program fragments.

6.1 Preliminaries

Given a set of periodic, online, real-time mission tasks to be executed on processors in extreme working environment, the overall reliability of the system must be maximized. In this section, we discuss the existing parameters and their specifications required to model our proposed system. These parameters include utilization factor, least upper bound of utilization, schedulability, reliability, fault recovery, and worst-case response time.

In this approach, we use static deadline monotonic (DM) approach for scheduling the set of periodic mission tasks of the processor. DM scheduling is a pre-emptive scheduling approach where tasks are assigned with fixed-priority; the highest priority is assigned to the task with the shortest deadline. In rate-monotonic scheduling (RM), and earliest deadline first (EDF) scheduling approaches, the relative deadline are assumed to be equal to the task period, whereas the deadline could be less than or equal to the task period in DM scheduling [102]. So, we have adopted DM scheduling as smaller self-test tasks will have an individual deadline which is lesser than the overall test period of the self-test task. Buttazzo and Giorgio [102] have demonstrated that the schedulability of these tasks is assured only if the least upper bound condition defined for the DM algorithm is satisfied. Also, the worst-case response time should not exceed the corresponding deadline of each task. So, to embed a self-test task into the set of mission tasks, the schedulability must be validated, i.e., the utilization and response time conditions must be satisfied for all the tasks.

6.1.1 Utilization Factor of Real-time Applications

Let $\Gamma = \{\tau_1, \tau_2, \tau_3, \ldots, \tau_n\}$, be a set of n periodic real-time mission tasks, with each periodic task τ_i is represented by a tuple (T_i, D_i, E_i), where T_i represents the period length, D_i represents the deadline, and E_i represents the execution time of the periodic task τ_i. In a non-faulty processor system, deadline of each mission task in the DM schedule would always be less than or equal to its period of execution ($D_i \leq T_i$). The utilization factor U_i of a mission task τ_i is E_i/D_i, and the overall utilization of CPU is:

$$U = \sum_{i=1}^{n} U_i = \sum_{i=1}^{n} \frac{E_i}{D_i}, \tag{6.1}$$

which is in the range [0,1). To evaluate the schedulability of static DM scheduling for periodic tasks, a least upper bound (U_{LUB}) could be used [102].

6.1.2 Least Upper Bound

The least upper bound is the fundamental criteria for the verification of schedulability and depends solely on the number of mission tasks. The set of n periodic tasks Γ, with overall utilization U, could be scheduled with DM algorithm if

$$U = \sum_{i=1}^{n} U_i \leq n(2^{1/n} - 1) \tag{6.2}$$

So, the least upper bound of processor utilization for DM algorithm is $U_{LUB} = n(2^{1/n} - 1)$.

6.1.3 Reliability Analysis

Reliability is the probability that a system does not deteriorate during a time interval [0,t]. Stanisavljevi et al. [103] defines reliability $Rel(t)$ as follows:

$$Rel(t) = e^{-\beta t}, \tag{6.3}$$

where β is the failure rate. Reliability decreases as time interval and failure rate increase.

Let F be the percentage of faults that can be detected and recovered using a set of test routines. This recovery mechanism leads to the minimization of effective failure rate from β to $\beta(1 - \frac{F}{100})$. Therefore, the reliability (Equation 6.3) of a system executing along with a set of test routines of coverage F during a time interval [0,t] is modified as:

$$Rel(t) = e^{-\beta(1-\frac{F}{100})t} \tag{6.4}$$

So, the system reliability increases with efficient fault detection which is followed by the recovery of the mission task execution.

6.1.4 Recovery Scenarios

The three different intermittent error recovery models discussed in [95] are

- Rollback-only recovery: In this conventional model, upon fault detection, the execution is rolled back to the previous checkpoint and re-execute.

- Core-level reconfiguration: To identify the fault type and the fault-prone core, an error discrimination mechanism is applied to each core of multicore systems. The recovery action is to shut down the core temporarily.

- Unit-level reconfiguration: A defective microarchitectural unit of a core could be disabled for a specific duration only if the activities of the unit do not affect the core operations. Otherwise, the core continues its operation without disabling the unit.

Rollback-only recovery is a time-redundant, process recovery mechanism. In this technique, equidistant checkpoints are assigned and upon fault detection, the recovery is conducted using the inputs of last stored checkpoint. To accomplish fault tolerance in low-cost, real-time embedded processors, rollback-only recovery mechanism would be simpler, cheaper, and efficient as fault discrimination, diagnosis, and hardware reconfiguration are not required for this model.

6.1.5 Worst Case Response Time

Zhang et al. [104] discuss an optimal checkpointing approach with the minimized worst-case response time. During the execution of task $\tau_i(T_i, D_i, E_i)$, p_i checkpoints are inserted equidistantly to handle utmost k faults in each task instance. Let C_s be the time consumed for saving a checkpoint and C_r be the time consumed for retrieving a checkpoint. The worst-case response time of τ_i is minimum for $p_i = \sqrt{kE_i/C_s} - 1$ [104].

For a real-time periodic task τ_i, the worst-case response time should incorporate checkpointing overhead of p_i number of store and restore operations for tolerating k faults, to be calculated in [104] as:

$$\begin{aligned} WCRT_i^{j+1} = \Big[E_i + k(C_s + C_r) + p_iC_s + \frac{kE_i}{p_i+1}\Big] + \\ \sum_{h=1}^{i-1}\left\lceil\frac{WCRT_i^j}{T_h}\right\rceil\Big[E_h + k(C_s + C_r) + p_hC_s + \frac{kE_h}{p_h+1}\Big], \end{aligned} \tag{6.5}$$

where $WCRT_i^{j+1}$, the worst-case response time of τ_i, is computed recursively until

- $WCRT_i^{j+1} = WCRT_i^j$ and $WCRT_i^j < D_i$ for some j *or*
- $WCRT_i^{j+1} > D_i$

If the first condition is satisfied, τ_i is schedulable. In the second case, τ_i is not schedulable [104]. The first expression in Equation 6.5 represents the response time of τ_i, whereas the second expression represents the sum of response times of higher priority

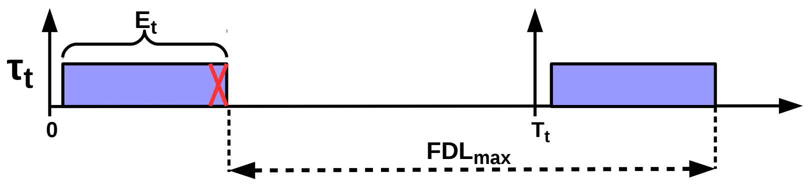


Figure 6.2: *Maximum Fault Detection Latency for Self-test Task τ_t*

tasks than that of τ_i, in accordance with DM scheduling. These two terms constitute the worst-case response time of real-time periodic tasks.

In the proposed framework described in the next section (Section 4.4), the conventional model of SBST synthesis and its reliability issues in the detection of intermittent faults are discussed.

6.2 SBST Programs for Intermittent Fault Detection

To comprehensively test all the processor functionalities, large self-test codes are applied periodically between the execution of mission tasks. In fact, the scheduler considers self-testing as a real-time task and includes it in the set of mission tasks to be scheduled. So, larger test codes will lead to larger test periods, which could be inefficient in the detection of intermittent faults.

The fault detection latency will be higher for a process schedule with infrequent self-test executions with longer test periods. If the intermittent faults occur just after the self-test execution, a larger self-test period will lead to an undesirable, higher fault detection latency, i.e., the fault detection latency will be maximum (FDL_{max}) when the fault occurs at the end of the execution of test process τ_t, and is detected only during the execution of next instance of τ_t as shown in Fig. 6.2. An intermittent fault might instantaneously disappear after its occurrence. So, if the time interval between two consecutive self-test executions is large, these faults would barely get detected. If self-testing frequently fails to detect and recover from these faults, subsequently, it could lead to system crash.

To reduce fault detection latency of intermittent faults, efficient, small chunks of SBST codes could be executed frequently between the mission tasks. So, the synthesized self-test program could be replaced with smaller FTPs. These FTPs must be coverage-efficient and are executed intermittently during a self-test period to detect the intermittent faults quickly following the fault occurrence. To tolerate every testable fault and reduce fault detection latency, we discuss a reliable, fragmented testing approach for processors in the next section.

6.3 Fragmented SBST for Testing Intermittent Faults

In this self-testing approach, the processor is subjected to fragmented testing for tracing the intermittent faults. In this approach (Subsection 6.3.1), the test code, synthesized for manufacturing testing, is replaced with a number of small and efficient test codes with adequate fault coverage. These test code fragments are examined and periodically applied between the normal tasks of the processor. From a testing point of view, these small but adequately efficient fragments replace the bigger parent test code to reduce the fault detection latency maintaining the system reliability. During a test period, these FTPs are applied on the processor periodically in small execution windows between the execution of the mission task. In particular, we must set the shortest possible test execution window so that all intermittent faults could be detected. If any erroneous behavior of the processor is diagnosed, the mission task is rolled back to the previous checkpoint and re-executed.

6.3.1 Synthesis of Smaller Latency Self-test Programs with Adequate Coverage

In the traditional self-test task methods, self-test code is synthesized using evolutionary automation techniques based on genetic programming of microprocessors (μGP). Various self-test solutions are generated in each generation of the μGP test development, and finally, self-test solutions with maximum fault coverage are selected for online testing. These high-coverage test programs are executed between the mission tasks periodically.

Fig. 6.3 shows the traditional online self-test scheduling of processors. Let H be an instance of a mission task τ_i. Let $\tau_t = (T_t, D_t, E_t)$ be the self-test task and S be an instance of τ_t. Between the execution of the mission task τ_i, the self-test task τ_t, corresponding to the high-coverage self-test program, is executed and the responses are observed for the detection of every testable hardware fault. To tolerate the occurrence of processor faults, we use checkpoints with rollback recovery mechanism. This time redundancy technique helps to recover from any irregularities in the processor circuit

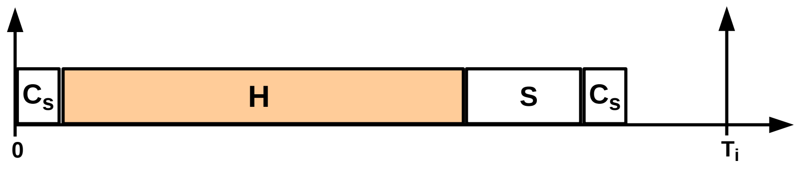


Figure 6.3: *Time Diagram of Traditional Testing over Period T_i*

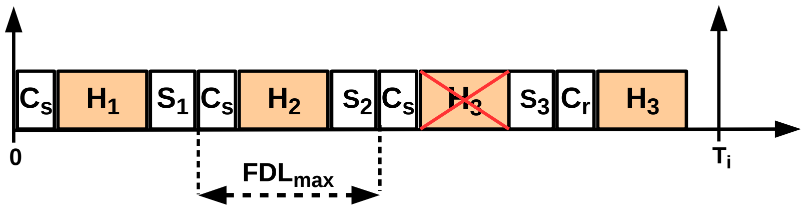


Figure 6.4: *Fragmented Testing over Period T_i*

by assigning equidistant checkpoints and storing (C_s) the non-faulty processor state. If the self-test code could identify any processor fault, the execution is rolled back to the latest checkpoint, process inputs are restored, and the mission task H is re-executed completely.

However, these self-test codes, with high fault coverage, might not trace the intermittent faults discreetly as the gap between two consecutive self-test executions would be very large. This gap could increase the maximum fault detection delay and eventually, intermittent faults could not be detected and recovered. To solve this, a reliable set of smaller self-test codes must be executed periodically.

So, we extract smaller FTPs from the test solutions synthesized during different generations of the μGP test synthesis. Each self-test solution, which is a sequence of instructions, and its coverage are logged during the evolutionary test development. Further, these self-test codes are extracted and applied as test fragments with smaller test periods, replacing the actual self-test code.

In the example shown in Fig. 6.4, the self-test task S is replaced with 3 sub-tasks S_1, S_2, and S_3. When the self-test sub-task S_3 identifies a fault in the third segment (H_3) of the mission task, every input of the system from the previous checkpoint are restored (C_r). This would reduce the maximum fault detection latency from the execution time of H to that of H_3 as the majority of the testable processor faults could be detected by S_3.

6.3.2 Calculation of Test Periods for FTPs

Let us assume self-test task S is replaced with m shorter self-test task instances S_1, S_2, S_3, $\ldots$, S_m. Each of these shorter self-test task instances, termed as self-test sub-tasks, would comprehensively test those processor functionalities which are highly susceptible to irregularities, in a single test period of execution. These code fragments are categorized into different sets of self-test codes for convenient reliability analysis. Each set of FTPs corresponds to a group of self-test codes with the same coverage. For example, the actual self-test program is split into groups, say, $G_{75}, G_{76}, G_{77}, \ldots, G_{95}$, of FTPs with coverage $F = 75\%, F = 76\%, F = 77\%, \ldots, F = 95\%$, respectively. Every FTP with coverage $F = 80\%$ are included in group G_{80}, and every FTPs with coverage $F = 81\%$ are included in group G_{81}, and so on. However, the test execution times of self-test codes in a group are observed to be approximately equal. For example, a set of FTPs of the group G_{85}, which have an average execution time of 0.032 milliseconds, could replace the actual self-test code with 96.3% coverage, to be executed as the self-test sub-tasks during T_t. But the fragmentation should be lossless such that the selected set of FTPs must cover the faults which could be detected by the unfragmented, actual self-test code.

In this approach, conditions for the synthesis of FTPs are:

1. Let $F(S_j)$ be the fault coverage of the FTP S_j. We select m FTPs such that the union of coverages of these m FTPs would be approximately equal to the coverage of the actual self-test code S, i.e.,

$$F(S) \approx F(S_1) \cup F(S_2) \cup \cdots \cup F(S_m) \tag{6.6}$$

2. Let τ_{t_f} be the self-test sub-task executing the largest FTP of a group. Then,

$$U = \sum_{i=1}^{n} U_i + U(\tau_{t_f}) \leq (n+1)(2^{1/(n+1)} - 1), \tag{6.7}$$

   i.e., the sum of the utilization values of n mission tasks and the largest sub-task should be less than the least upper bound for $n + 1$ tasks (n mission tasks and a self-test sub-task) to achieve a valid DM schedule. If this inequality holds true for the largest FTP of a group, for which the largest or worst-case utilization needed, the smaller FTPs of that group would also satisfy this condition. The remaining self-test sub-tasks in the group of τ_{t_f} are executed with equal test period as that of τ_{t_f}. So, these sub-tasks are treated as repeated executions of τ_{t_f}.

3. Also, the worst-case response time of each of the mission tasks and test sub-tasks calculated using Equation 6.5 must be less than the task deadline, i.e., τ_i and τ_{t_f} are schedulable only if $WCRT_i < D_i$ and $WCRT_{t_f} < D_{t_f}$.

Using Equation 6.7, an optimal permissible execution window size for the self-test sub-tasks in each group is evaluated. The window size of each group is determined from the utilization $U(\tau_{t_f})$ of the sub-task corresponding to the largest FTP in that group. The execution window of a self-test sub-task is equivalent to the sub-task period. As the window size of τ_{t_f} reduces, the utilization will increase. Eventually, the minimum window size for τ_{t_f} which satisfies the least upper bound condition of utilization is calculated for each group using Equation 6.7. From the optimal or minimum execution window size of each group, the optimal or maximum number (m) of FTPs in each group (with equal coverage F), that can be executed during T_t, is calculated such that the overall utilization is maximized.

Let us say, the deadline of τ_{t_f} is same as its time period. So, from Equation 6.1, the utilization of τ_{t_f} is understood as:

$$U(\tau_{t_f}) = E_{\tau_{t_f}} / T_{\tau_{t_f}}, \tag{6.8}$$

The minimum value of the time period ($T_{\tau_{t_f}}$) of τ_{t_f} is equivalent to the execution window w of τ_{t_f}. So, the value of w for each group could be evaluated from the maximum possible value of $U(\tau_{t_f})$ and the execution time ($E_{\tau_{t_f}}$) of τ_{t_f} as:

$$w = min(T_{\tau_{t_f}}) = E_{\tau_{t_f}} / max(U(\tau_{t_f})), \tag{6.9}$$

where the maximum possible value of $U(\tau_{t_f})$ is determined from Equation 6.7 and the execution time ($E_{\tau_{t_f}}$) of the sub-task τ_{t_f} is evaluated experimentally.

6.3.3 Scheduling of FTPs

To schedule the FTPs, self-test period is split into equal execution windows. In each execution window, a self-test sub-task is selected and executed with a *pseudo-release time* and a *psuedo-deadline*. Suppose m self-test sub-tasks $\{\tau_{t_1}, \tau_{t_2}, \ldots, \tau_{t_m}\}$ could be executed during test period T_t. The value of m could be evaluated from the optimal execution window size w. So, the optimal number m of FTPs in the group of self-test sub-task τ_{t_j} with permissible execution window size w_j is:

$$m = \frac{T_t}{w_j} \tag{6.10}$$

The psuedo-release time r_j of the self-test sub-task τ_{t_j} is:

$$r_j = \frac{(j-1).T_t}{m}, \tag{6.11}$$

where T_t is the period of the self-test task τ_t. The psuedo-deadline d_j of each τ_{t_j} is:

$$d_j = \frac{j.T_t}{m} \tag{6.12}$$

Using Equation 6.10, the optimal number of FTPs for each group is calculated using the test period T_t and the execution window size w_j calculated in Equation 6.9 discussed in Subsection 6.3.2. In a test period, each of these sub-tasks is executed exactly once, where the DM scheduler assigns the highest priority for the sub-task with the shortest pseudo-deadline. Finally, the testable intermittent faults of the processor are considered to be completely tested only after the execution of all FTPs are carried out.

Although smaller self-test codes have lesser fault detection latency, such tiny code snippets are barely reliable due to lower coverage. Using a reliability analysis, we could identify the FTPs with optimal fault coverages. In the next subsection, we analyze the optimality of FTPs with respect to the fault coverage and fault detection latency.

6.3.4 Reliability Enhancement Analysis of FTPs

After applying any of the proposed self-adjustment schemes suggested in [98] for an increased CPU utilization, a self-test task τ_t is executed periodically between the mission tasks. To reduce the fault detection latency, the self-test task τ_t is replaced with m smaller but coverage-efficient self-test tasks τ_{t1}, τ_{t2}, τ_{t3}, $\ldots$, τ_{tm} with an equal execution window size w.

To evaluate the reliability of self-test sub-tasks, we assume a self-test sub-task with adequate coverage F executing with a test execution window size w along with the set of mission tasks. For a system integrated with rollback-only recovery technique, if a fault is detected during the self-test sub-task execution, the mission tasks would be rolled back to the previous checkpoint and re-executed. This recovery mechanism leads to

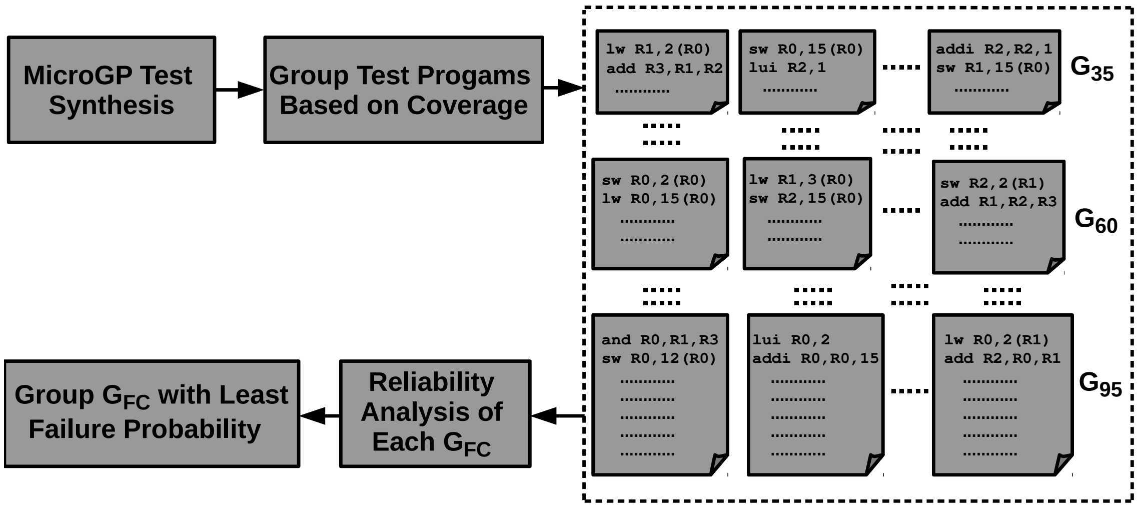


Figure 6.5: *Overall Self-test Code Fragment Synthesis*

the minimization of effective system failure rate from β to $\beta(1 - \frac{F}{100})$. Therefore, the reliability of mission tasks executing along with a set of self-test sub-tasks of coverage F with a test execution window would be $e^{-\beta(1-\frac{F}{100})w}$ as defined in Equation 6.4.

The reliability of the actual, synthesized, self-test code S, with 96.3% coverage, is evaluated using Equation 6.4. If S is executed periodically during a time period w, which is the test execution window, the effective failure rate is reduced from β to $\beta(1 - \frac{96.3}{100})$. So, the system reliability would be increased from $e^{-\beta w}$ to $e^{-0.037\beta w}$. However, due to a large test execution window w of the self-test code S, the reliability would still be very small.

The reliability value in Equation 6.4 is dependant on two parameters; Fault coverage (F) of an FTP and its execution window w, which is proportional to the maximum fault detection delay. For the self-test codes that have adequate coverage and small execution window, reliability will be higher. Further, the failure probability $FP(w) = 1 - Rel(w)$ for each group of FTPs is evaluated from the reliability parameter ($Rel(w)$) in Equation 6.4. We select the group of FTPs with least failure probability as the self-test codes for online processor testing.

6.3.5 Overall Synthesis of Self-test Fragments

The proposed FTP synthesis is shown in Algorithm 6. In Steps 2 to 6 of Algorithm 6, each group is subjected to reliability evaluation to select the group with minimum failure probability. In Step 2, we select the FTP of a group G_F with the longest execution time.

ALGORITHM 6: Reliable Self-test Fragment Synthesis

```
Input: A self-test task $\tau_t$ executing self-test program $S$ with period $T_t$;
       Groups $G_{75}, G_{76}, G_{77}, \cdots, G_{95}$ of FTPs, where $G_F$ is a set of all FTPs
       with fault coverage $F$;
Output: Group $G$ with maximum system reliability;
1 for each group $G_F$ do
2     Select the FTP $S_i$ in $G_F$ with longest execution time;
3     Calculate optimal test period $w$ for the execution of $S_i$ with the help of
        Equation 6.7, Equation 6.8, and Equation 6.9;
4     Apply the same test period $w$ as the execution window for all FTPs in $G_F$;
5     Calculate the optimal number $(m)$ of FTPs of $G_F$ with execution window $w$
        that can be executed in a test period $T_t$ using Equation 6.10;
6     Calculate Reliability value $Rel(w)$ of group $G_F$, where $\beta$ is the failure rate for
        a time interval $w$, as shown in Equation 6.4;
7 end
8 Select group $G$ with minimum failure probability $FP(w)$, where
    $FP(w) = 1 - Rel(w)$;
9 Schedule $m$ FTPs of the selected group G as the self-test sub-tasks with
    pseudo-release time and pseudo-deadline discussed in Equation 6.11 and
    Equation 6.12, respectively.
```

Now, the optimal test period (execution window) w of this FTP is determined using the least upper bound inequality of overall utilization as discussed in Step 3. In Step 4, every FTP of group G_F is assigned with the same execution window w to execute in the same test period. Finally, the optimal number of FTPs m of group G_F (Step 5) and the reliability value $Rel(w)$ of group G_F (Step 6) are evaluated using the optimal size of execution window (w) and the failure rate (β) for the intermittent faults. In Step 8, we select the group G that yields maximum reliability and subsequently, schedule the FTPs of the group G with a pseudo-release time and pseudo-deadline (Step 9).

As shown in Fig. 6.5, the SBST programs are developed and stored with the help of a μGP test synthesis procedure. Now, each of these FTPs is grouped based on the fault coverage. In these conditions, the schedulability and the response time constraints are investigated for each group of FTPs. Later, these groups of FTPs are subjected to system reliability analysis to identify the group G_F with minimal failure probability. In

Table 6.1: *Mission Task Workloads on 100 MHz MIPS Processor Model*

Task	**Exec. Time E_i** (Cycles)	**Deadline D_i** (Cycles)	**Period T_i** (Cycles)
T1	29,000	400,000	500,000
T2	80,000	1,100,000	1,400,000
T3	200,000	1,300,000	1,500,000
T4	87,000	1,600,000	2,000,000
T5	13,500	1,700,000	2,100,000
T6	130,000	1,700,000	2,200,000
T7	21,000	2,500,000	3,200,000
T8	135,000	2,850,000	3,000,000
T9	450,000	3,200,000	3,500,000
T10	30,000	3,500,000	4,000,000
T11	80,000	3,600,000	4,000,000
T12	30,000	3,700,000	4,200,000

the next section, self-test experiments are performed on a 32-bit MIPS processor model and the schedulability and reliability are analyzed to corroborate the proposed fragment synthesis strategy.

6.4 Experimental Results

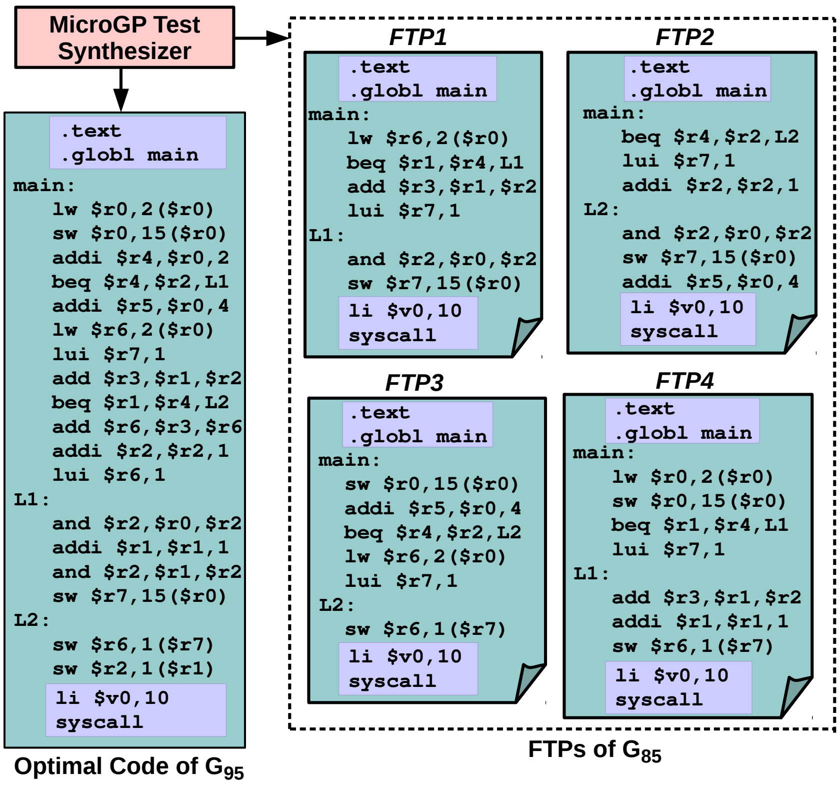
In our approach, we model the intermittent faults as stuck-at faults in the behavioral level and our self-test programs are simulated for identifying these high-level faults. For the self-test generation, we have used references of MIPS instruction set for a 32-bit MIPS processor model. The self-test codes are evaluated using the behavioral fault model shown in Table 3.1 in Chapter 3 with 10 different HDL fault representations. The synthesized test programs are simulated on a ModelSim 10.5b simulator platform for the MIPS processor. Later, the responses are extracted and compared for the fault-injected and non-faulty processor models and subsequently, the self-test codes with low memory footprint and maximum coverage are selected.

In Fig. 6.6, an example of fragment synthesis is shown for the self-test programs of the branch functionality of a MIPS processor. The self-test code belongs to the group G_{95} covers 95% of similar testable behavioral faults of branch instruction execution. However, the detection of intermittent faults is barely ensured by such larger self-test codes as the maximum fault detection latency is very high. To reduce the fault detection latency, G_{95} is replaced by FTPs with lesser coverages (e.g., 85%) that are executed frequently. Each FTP, shown in Fig. 6.6, consists of at least one *beq* instruction and therefore, could detect 85% of the behavioral faults modeled for the branch instruction execution. So, the maximum fault detection latency of branch functionality could be reduced nearly 4 times by replacing the optimal self-test code of group G_{95} with 4 FTPs of the group G_{85}, as shown in Fig. 6.6. Also, the overall coverage of these 4 FTPs should be nearly equal to 95%, i.e., the fragmentation must not be lossy. However, these FTPs could replace the larger self-test code, only if the modified overall utilization is lesser than the least upper bound of utilization and the modified response times of each FTP meet their corresponding deadline. Likewise, each group of FTPs is selected from the population of the μGP evolutionary test synthesis and is subjected to reliability analysis. Thereupon, the most reliable group of FTPs is selected for the self-testing, and these FTPs are employed as self-test sub-tasks, which are scheduled with equal execution windows.

6.4.1 A Case Study of Reliable Synthesis of FTPs

The operational frequency of synthesized versions of the MIPS processor model is 100 MHz. A set Γ of 12 mission task workloads is shown in Table 6.1. The normal overall utilization for U is 0.673155, whereas the schedulability for 12 tasks has the upper bound of 0.713557. If a self-test task is integrated with the mission task set, the upper bound for 13 tasks would reduce to 0.711959, i.e., the maximum utilization (0.711959 - 0.673155) permissible for the new self-test task would be 0.038804. So, we introduce a self-test task τ_t with $E_t = 25000$ cycles, $T_t = 800000$ cycles (8 ms), and 96.3% coverage. The utilization $U_t = 0.035714$ and the overall utilization becomes $U = 0.708870 \leq 0.711959$. So, the schedulability of the workloads along with τ_t is assured. So, the sub-tasks of τ_t are restricted to an overall utilization bound $U_t (= 0.035714)$. Using this overall utilization, we extract the minimum permissible execution window and the maximum number of FTPs possible for every group G_F of fault coverage F.

The relation between the maximum number of sub-tasks and execution window size of a single sub-task is depicted in Equation 6.10. As the number of FTPs in a test period increases, the execution window size will reduce. In Fig. 6.7, the utilization of



Optimal Code of G_{95}:

```
 .text
 .globl main
main:
    lw $r0,2($r0)
    sw $r0,15($r0)
    addi $r4,$r0,2
    beq $r4,$r2,L1
    addi $r5,$r0,4
    lw $r6,2($r0)
    lui $r7,1
    add $r3,$r1,$r2
    beq $r1,$r4,L2
    add $r6,$r3,$r6
    addi $r2,$r2,1
    lui $r6,1
L1:
    and $r2,$r0,$r2
    addi $r1,$r1,1
    and $r2,$r1,$r2
    sw $r7,15($r0)
L2:
    sw $r6,1($r7)
    sw $r2,1($r1)
    li $v0,10
    syscall
```

FTP1:

```
 .text
 .globl main
main:
    lw $r6,2($r0)
    beq $r1,$r4,L1
    add $r3,$r1,$r2
    lui $r7,1
L1:
    and $r2,$r0,$r2
    sw $r7,15($r0)
    li $v0,10
    syscall
```

FTP2:

```
 .text
 .globl main
main:
    beq $r4,$r2,L2
    lui $r7,1
    addi $r2,$r2,1
L2:
    and $r2,$r0,$r2
    sw $r7,15($r0)
    addi $r5,$r0,4
    li $v0,10
    syscall
```

FTP3:

```
 .text
 .globl main
main:
    sw $r0,15($r0)
    addi $r5,$r0,4
    beq $r4,$r2,L2
    lw $r6,2($r0)
    lui $r7,1
L2:
    sw $r6,1($r7)
    li $v0,10
    syscall
```

FTP4:

```
 .text
 .globl main
main:
    lw $r0,2($r0)
    sw $r0,15($r0)
    beq $r1,$r4,L1
    lui $r7,1
L1:
    add $r3,$r1,$r2
    addi $r1,$r1,1
    sw $r6,1($r7)
    li $v0,10
    syscall
```

Figure 6.6: *Replacement of a Test Program of Group G_{95} by 4 Smaller Test Programs (FTPs) of Group G_{85} for testing the Branch Functionality of a MIPS Processor*

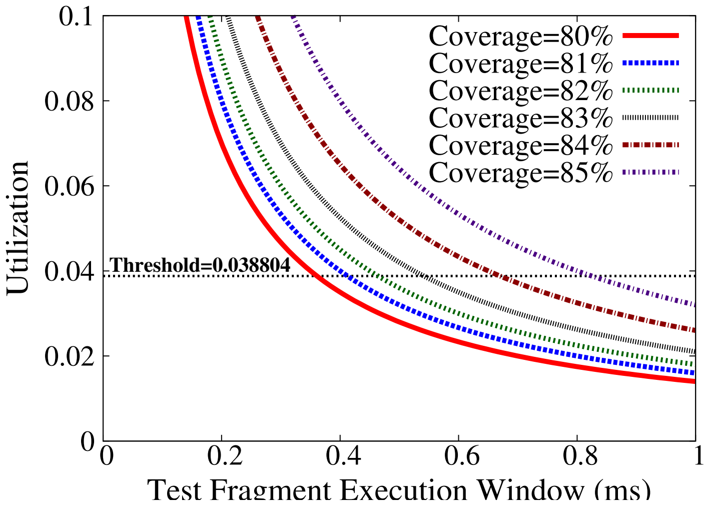


Figure 6.7: *Utilization of Different Test Execution Windows on 100 MHz MIPS Processor Model*

each group of FTPs is depicted for different execution windows, i.e., for different number of FTPs. In this case study, the maximum possible utilization for a set of sub-tasks in a group G_F is 0.038804. So, if the number of sub-tasks is high, the execution window will

Table 6.2: *Test Program Characteristics on 100 MHz MIPS Processor Model*

Fault Coverage (%)	Avg. Test Execution Time (ms)	Avg. FTP Size (KB)	Lower Bound w for Test Execution Window (ms)	Optimal Number of FTPs m	Overall Coverage of FTPs (%)
80	0.014	1.72	0.40	20	96.0
81	0.016	1.87	0.45	17	95.8
82	0.018	1.92	0.52	15	95.5
83	0.021	1.98	0.61	13	95.9
84	0.026	2.10	0.75	10	96.0
85	0.032	2.31	0.90	8	95.6

be small and the overall utilization increases and may cross the schedulability bound. If the number of sub-tasks is lesser, the execution window will be expanded to satisfy the schedulability threshold.

As shown in Table 6.2, the FTPs in the group G_{80} has maximum utilization when the execution window size is 0.4 ms, the group G_{81} has execution window size of 0.45 ms, and so on. Using Equation 6.10, the minimum execution window size of FTPs is deduced from the number of schedulable FTPs for each group. Also, the execution window size increases as the coverage of the FTP increases and vice versa. For example, 10 FTPs of the group G_{84} could be executed during a test period T_t with an execution window size 0.75 ms whereas for the group G_{85}, only 8 FTPs could be executed during T_t bearing a longer execution window of 0.9 ms. However, the overall coverage of each set of fragments is near to 96%, which results in a lossless fragmentation as the coverage of the actual self-test task τ_t is 96.3%.

From Table 6.2, it may be noted that for the low-coverage FTPs, fault detection latency is lesser. However, if the fault coverage decreases beyond a limit, intermittent faults might go undetected and reliability would be less. The failure probability of mission tasks (with no self-testing) calculated using the reliability Equation 6.4 for different durations of time intervals is shown in Fig. 6.8. In this case, the failure probability increases as the time interval and failure rate (β) increase.

Now, we consider the inclusion of self-test sub-tasks and observe the variation in the failure probability for different groups of FTPs as shown in Fig. 6.9. For the group G_{80}, the failure probability seems to be the lowest of all groups. When the coverage increases or decreases from 80%, the probability of failure is observed to be increasing

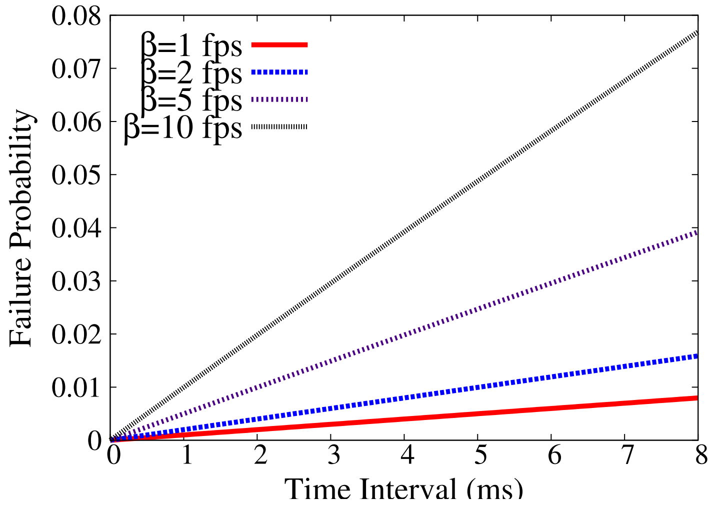


Figure 6.8: *Failure Probability of the Workloads on 100 MHz MIPS Processor Model with No Testing*

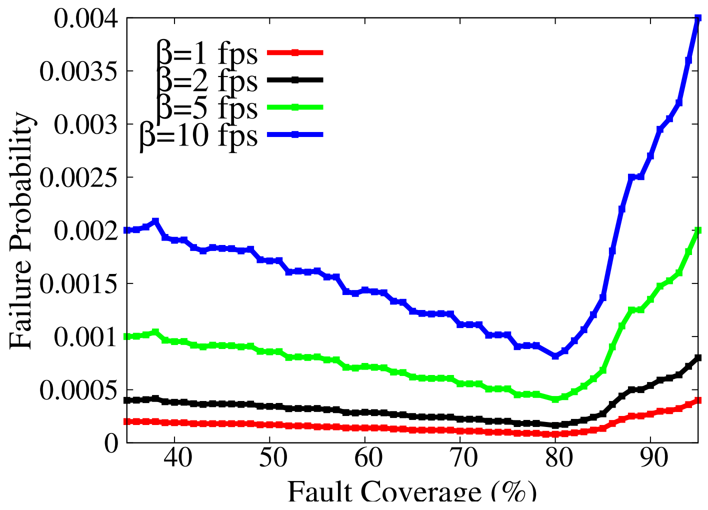


Figure 6.9: *Failure Probability of Workloads along with Self-test Sub-tasks on 100 MHz MIPS Processor Model*

for different failure rates ($\beta = 1, 2, 5, 10$ faults per second (fps)). So, to achieve efficient fault detection latency and test quality, we adopt the optimal number of FTPs from G_{80} estimated using Equation 6.7. These FTPs are executed with the pseudo-release time and pseudo-deadline as defined in Equation 6.11 and Equation 6.12, respectively.

Following the reliability evaluation of each group, FTPs of groups with minimum failure probability is selected for online self-testing. To accomplish fault tolerance in online self-testing, FTPs of G_{80} is chosen due to the lowest failure probability of all groups. Finally, 20 FTPs of G_{80} are executed with an execution window of 0.4 ms in a single test period, where the pseudo-release time and pseudo-deadline $(r(i), d(i))$ are (0 ms, 0.4 ms) for the first self-test sub-task, (0.4 ms, 0.8 ms) for the second self-test sub-task, (0.8 ms, 1.2 ms) for the third self-test sub-task, and so on. These 20 FTPs, altogether, ensures coverage of 96%, which is nearly equal to the coverage of the unfragmented self-test task (96.3%). This distributed self-testing approach could realize the development of high coverage self-test codes with minimal fault detection latency for the tolerance of intermittent faults.

6.5 Summary

In this work, a reliable set of shorter SBST code fragments were discovered to replace the large SBST codes in online processor testing. These set of fragments, developed with the help of enhanced reliability analysis, would have adequate test quality and minimal fault detection delay. The schedulability and the overall coverage efficiency of these self-test sub-tasks, executed along with a set of 12 mission tasks, are verified on a 32-bit MIPS processor model. To effectively trace the intermittent faults on a MIPS processor circuitry, the actual self-test code with 96.3% coverage and a test period of 8 ms could be replaced with a set of 20 fragments of test tasks with 80% coverage and a test period of 0.4 ms. To ensure high-quality fragmentation, adequate overall coverage of 96% is achieved by these 20 code fragments. Currently, we are working on the development of a neural network-based fault estimation model using test signatures of power and temperature consumptions, to predict the occurrence of intermittent faults.

Chapter 7

Conclusions and Future Perspectives

7.1 Summary

As the sophistication of processor circuits are increasing exponentially every year, test performance has also become crucial. In this thesis, we have addressed four major challenges of SBST test procedure for processors. Each of these challenges leads to performance degradation in the execution of different phases of SBST testing. Our proposals attempt to deal with the critical challenges in SBST testing domain towards a high-performance, high-quality SBST test procedure. These proposed techniques are validated on a configurable 32-bit MIPS processor and a full-fledged 7-stage pipeline SPARC V8 Leon3 soft processor and can be extended towards potentially developing a meaningful fault discovering technique for more complex multicore NoCs and SoCs.

In the first contribution, an enhanced SBST synthesis for the processor cores is employed by integrating the greedy coverage and the testability features into the traditional evolutionary core module of μGP for the detection of the hard faults. From the experimental results, we could conclude that our strategy synthesizes test solutions that could detect 40% of the hard faults which affirms the development of high-quality test programs. A comprehensive behavioral fault model is used to capture the possible faults in the processors. Also, the scalability of the proposed scheme is demonstrated by analyzing and comparing the effort in testing the behavioral faults for a MIPS processor and a Leon3 processor. As the number of behavioral faults does not change considerably with the processor size, the test program generation time is nearly equivalent for both processors, which explains the scalability of our scheme.

The second contribution extends the existing SBST automation techniques and also, the technique proposed in our first contribution with a faster SBST synthesis called RSBST for processors. In this work, an accelerated greedy-based evolutionary method is employed to detect the hard-to-test faults. To realize this, we search for identical test programs in terms of observability and reuse their simulation responses. This affirms more than 80% reuse of intermediate test programs (chromosomes in terms of the genetic algorithm) for both MIPS processor and Leon3 processor.

To compress larger test programs developed for complex processors, a two-stage algorithm is proposed and validated in our third contribution. These stages are enhancements of existing instruction removal and instruction restoration techniques. The large test programs are compacted effectively and effortlessly using our proposed technique. The first stage of compaction easily eliminates the loosely coupled independent instruction groups. In the second stage, we have used a top to bottom restoration-based compaction for the remaining instructions that reduce the test code size by 19%, consuming 142.33 fault simulations, thereby allowing low-cost compaction.

In our fourth contribution, a set of shorter SBST code fragments with enhanced reliability were discovered to replace the large SBST codes in online processor testing. These set of fragments, synthesized during the evolutionary process and discovered with the help of enhanced reliability analysis, have adequate test quality and minimal fault detection delay in the detection of temporary faults. The schedulability and the overall coverage efficiency of these self-test sub-tasks, executed along with a set of real-time mission tasks, are verified on a 32-bit MIPS processor model. It is observed that the intermittent faults on a MIPS processor circuitry could be effectively traced if the actual self-test code of 96.3% could be replaced with a set of 20 fragments of test tasks with 80% coverage.

In all our works, we have demonstrated the validation of algorithms with the help of elaborate experimental results and further analysis. Altogether, we conclude our thesis works, comprising different techniques and corresponding results for the development of a high-quality, accelerated, and low-cost SBST test process with enhanced reliability.

7.2 Future Works

The work presented in this thesis leaves several open directions and there is ample scope for future research in the domain of SBST for processors. In this section, we present four such future perspectives.

- **Advanced fault discovery using greedy SBST synthesis:** The evolutionary and greedy concepts of automated SBST synthesis can be extended towards potentially developing a meaningful and profound fault discovering technique for more complex multicore NoCs and SoCs. The computational components of complex SoCs can be functionally tested using SBST programs. In Leon3 processor, the floating point components and the coprocessor could also be tested for the completeness of the test process.

- **Fragment-wise reusability and fault equivalence techniques:** A faster and profound test synthesis could be developed using the fragment-wise reusability of test programs. Even if the observability of two test programs are different, the identical, and data-independent code fragments (chunks) could be extracted from these test programs and reused. Also, the fault equivalence techniques could be used for reducing the volume of simulations and the test generation time. Two faults are equivalent if the output function of a processor module is the same for both of them. So, we classify the equivalent faults into the same group. Also, if a test program could detect one of these faults, it can detect all its equivalent faults. So, all faults in a group of equivalent faults could be tested using a single fault simulation, which will accelerate the test synthesis.

- **Instruction reordering technique could be integrated for further optimization:** Our work on removal-restoration techniques could be extended by integrating instruction reordering techniques for further optimization of the test program. We could heuristically search for the optimal ordering of instruction groups with minimum execution time while maintaining the fault coverage. However, this search process must be conducted keeping a threshold on the computational cost.

- **Test code optimization by removing common instruction in test sets:** The module-by-module test programs could have instructions that cover common faults. The instructions of a test program for a module may also cover some of the faults of other modules. So, we can extend our test program compaction work by identifying and removing instructions which cover faults that are already covered by test programs of other modules. To realize this, test programs with the largest excess fault coverage are to be considered first and redundant instructions are to be removed from other test programs. Also, a comparison study for the monolithic test programs and module-by-module test programs, in terms of execution time and size, can be conducted to determine the most efficient online testing approach.

- **Scheduling of FTP execution to enhance intermittent fault coverage:** FTP execution time is significantly smaller ($< 10\%$) when compared with the mission critical task execution. So, it may generate less heat as compared to mission critical applications but it can detect intermittent faults and raise a flag for fault tolerance. However, differnet execution sequences of FTPs executed in a test period lead to different power consumption and heat generation, i.e., the execution ordering of FTPs have direct impact on power consumption and temperature of the processor. So, each FTP must be assigned a priority and based on this priority, they must be scheduled during test period interleaving mission application. Therefore, a study of FTP execution scheduling can be conducted to maximize the intermittent fault coverage.
- **SBST techniques for memory consistency testing and validation:** Memory consistency models significantly impact the ease of programming a multi-processor system, as well as the set of hardware and compiler optimizations. Commercial architectures support a variety of memory models, such as Sequential Consistency(SC), Total Store Order(TSO) and Release Consistency(RC). As several complex elements are involved in the memory hierarchy design, memory model testing is a major challenge for memory architects today. There are several approaches for checking the correctness of shared-memory multiprocessor implementations focusing on the memory subsystem. SBST approaches for multiprocessors are able to test the designs with programs whose results can be reasoned about a priori or are precomputable. So, a study on the memory consistency testing can be conducted using various SBST approaches.
- **SBST techniques for post-silicon validation and manufacturing testing:** After the processor chip design passes from the pre-silicon verification stage, few chip prototypes are fabricated and these prototypes are used as test objects in the

post-silicon validation stage. Post-silicon validation is in the orders of magnitude faster than simulation-based pre-silicon verification tests. SBST programs may be utilized to enhance the controllability and observability of functional verification tests to develop an effective the post-silicon validation. Production/manufacturing testing screens manufactured chips for physical faults or defects before the chip is released into the market. This testing procedure must be conducted in the actual speed of the processor hardware which is in GHz. As SBST ensures at-speed testing of circuits, it is an emerging solution for production test/manufacturing test. Currently, SBST is used for the at-speed testing of processors in BIST and online/concurrent testing methods. So, effective SBST soultions for both production test/manufacturing test and post-silicon validation can also be developed.

- **SBST techniques to test unencrypted firmware or software of a complex SoC design:** Firmware is low-level software which can directly access the physical memory space of its interacting hardware devices. This hardware-specific nature distinguishes it from higher-level software such as the operating system (OS) or application code which is device independent. This higher-level software communicates with the hardware via firmware. The firmware address space is kernel accessible but not user accessible and the software address space is assigned separately for each user. Correct functionality of firmware is critical and its malfunction while accessing critical physical memory can crash the OS or even the entire system. For example, bugs in device drivers were considered to be the cause of 85% of the failures of the Windows XP Operating System. This component of the system is increasing in scale and importance, and thus firmware validation is a critical part of system validation. SBST techniques can be developed to validate the firmware and software by testing their concurrency with the interacting SoC hardware modules.

- **SBST techniques to test encrypted firmware of a complex SoC design:** The encrypted firmware of complex SoC designs can be validated by developing SBST programs for the encryption hardware for the firmware. When SoC componets are accessed, all memory addresses that are issued by software are virtual. These memory addresses can be passed to the Memory Management Unit (MMU), which can check the TLBs for a recently used cached translation. If the MMU does not find a recently cached translation, the table walk unit reads the appropriate table entry from memory. As MMU ensures authorized access to firmware or software in memory, the integrity of the circuit could be compromised if any faults/defect occur in MMU hardware. To validate encrypted firmware, both encryption hardware and MMU hardware can be tested using dedicated SBST programs.

- **SBST techniques for validation of trust zones:** Trust Zones are embedded security technology that starts at the hardware level by creating two environments that can run simultaneously on a single core: a secure world and a not-as-secure world (non-secure world). *ARM TrustZone* Trust Execution Environment (TEE) is an implementation of the TEE standard, which offers a execution space of high-level application security. To validate these trust zones, SBST programs can be developed separately based on the level of testability of secure and non-secure environments.

Publications

- **Vasudevan Madampu Suryasarman**, Santosh Biswas and Aryabartta Sahu, **Automation of Test Program Synthesis for Processor Post-silicon Validation**, Journal of Electronic Testing, vol.34, no.1, pp.83-103, Feb 2018.
- **Vasudevan Madampu Suryasarman**, Santosh Biswas and Aryabartta Sahu, **RSBST: An Accelerated Automated Software-based Self-test Synthesis for Processor Testing**, Journal of Electronic Testing. vol. 35, 695714 (2019).
- **Vasudevan M S**, Santosh Biswas and Aryabartta Sahu, **RSBST: A Rapid Software-based Self-test Methodology for Processor Testing**, The 32nd International Conference on VLSI Design 2019.
- **Vasudevan Madampu Suryasarman**, Santosh Biswas and Aryabartta Sahu, "**Fragmented SBST Technique for Online Intermittent Fault Detection in Processors**". (Provisionally accepted with major revisions in IET Computers & Digital Techniques)

References

[1] David A Patterson and John L Hennessy. *Computer organization and design MIPS edition: the hardware/software interface*. Newnes, 2013. [Pg.xvii], [Pg.42], [Pg.43]

[2] Giovanni Squillero. Microgpan evolutionary assembly program generator. *Genetic Programming and Evolvable Machines*, 6(3):247–263, 2005. [Pg.xvii], [Pg.xviii], [Pg.xxiii], [Pg.9], [Pg.13], [Pg.16], [Pg.27], [Pg.29], [Pg.30], [Pg.39], [Pg.42], [Pg.43], [Pg.53], [Pg.54], [Pg.55], [Pg.58], [Pg.59], [Pg.63], [Pg.68], [Pg.69], [Pg.71], [Pg.83], [Pg.84], [Pg.85], [Pg.86], [Pg.87], [Pg.99]

[3] Martin Danek, Leos Kafka, Luks Kohout, Jaroslav Skora, and Roman Bartosinski. Utleon3: Exploring fine-grain multi-threading in fpgas. 2014. ISBN: 978-1-4614-2410-9. [Pg.xviii], [Pg.64], [Pg.65]

[4] Marco Gaudesi, Irith Pomeranz, Matteo Sonza Reorda, and Giovanni Squillero. New techniques to reduce the execution time of functional test programs. *IEEE Transactions on Computers*, 66(7):1268–1273, 2016. [Pg.xviii], [Pg.16], [Pg.19], [Pg.31], [Pg.32], [Pg.33], [Pg.89], [Pg.94], [Pg.95], [Pg.99], [Pg.100], [Pg.101], [Pg.102]

[5] Chien-In Henry Chen. Behavioral test generation/fault simulation. *IEEE Potentials*, 22(1):27–32, 2003. [Pg.xxiii], [Pg.12], [Pg.28], [Pg.39], [Pg.40], [Pg.41], [Pg.53], [Pg.54]

[6] Bernd Hoefflinger. *ITRS: The International Technology Roadmap for Semiconductors*, pages 161–174. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. [Pg.1]

[7] Peter J Ashenden, Jean Mermet, and Ralf Seepold. *System-on-Chip Methodologies & Design Languages*. Springer Science & Business Media, 2013. [Pg.1]

[8] Steve B Furber. *ARM system Architecture*. Addison-Wesley Longman Publishing Co., Inc., 1996. [Pg.1]

[9] Santosh Biswas and Jatindra Kumar Deka. Vlsi design verification and test. *[online]. Available:http://www.nptel.courses.in/106103016*. [Pg.1], [Pg.2], [Pg.24]

[10] Valeria Bertacco. Post-silicon debugging for multi-core designs. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, pages 255–258. IEEE Press, 2010. [Pg.1]

[11] Brian T Murray and John P Hayes. Testing ics: Getting to the core of the problem. *Computer*, (11):32–38, 1996. [Pg.1], [Pg.2]

[12] Michele Favalli and Cecilia Metra. Online testing approach for very deep-submicron ics. *IEEE Design & Test of Computers*, 19(2):16–23, 2002. [Pg.1]

[13] Paul KK Yeung, Alan D Howard, James W Hoo, and James L Pennock. Automatic test equipment for integrated circuits, August 9 1988. US Patent 4,763,066. [Pg.2]

[14] Raghuram S Tupuri and Jacob A Abraham. A novel functional test generation method for processors using commercial atpg. In *Proceedings International Test Conference 1997*, pages 743–752. IEEE, 1997. [Pg.3], [Pg.24], [Pg.26]

[15] Paolo Prinetto, Maurizio Rebaudengo, and M Sonza Reorda. An automatic test pattern generator for large sequential circuits based on genetic algorithms. In *Proceedings., International Test Conference*, pages 240–249. IEEE, 1994. [Pg.3], [Pg.24], [Pg.26]

[16] Antonis Paschalis, Dimitris Gizopoulos, Nektarios Kranitis, Mihalis Psarakis, and Yervant Zorian. Deterministic software-based self-testing of embedded processor cores. In *Proceedings of the conference on Design, automation and test in Europe*, pages 92–96. IEEE Press, 2001. [Pg.3], [Pg.4], [Pg.5], [Pg.14], [Pg.24], [Pg.26], [Pg.27], [Pg.33], [Pg.34]

[17] Adrian Carbine and Derek Feltham. Pentium (r) pro processor design for test and debug. In *Proceedings International Test Conference 1997*, pages 294–303. IEEE, 1997. [Pg.3], [Pg.24]

[18] Indradeep Ghosh, Anand Raghunathan, and Niraj K Jha. Hierarchical test generation and design for testability methods for aspps and asips. *IEEE transactions*

on computer-aided design of integrated circuits and systems, 18(3):357–370, 1999. [Pg.3], [Pg.24]

[19] Kwanghyun Kim, Dong Sam Ha, and Joseph G Tront. On using signature registers as pseudorandom pattern generators in built-in self-testing. *IEEE transactions on computer-aided design of integrated circuits and systems*, 7(8):919–928, 1988. [Pg.4], [Pg.25], [Pg.26]

[20] Kenneth D. Wagner, Cary K. Chin, and Edward J. McCluskey. Pseudorandom testing. *IEEE transactions on computers*, (3):332–343, 1987. [Pg.4], [Pg.25], [Pg.26]

[21] F. Corno, M. Sonza Reorda, G. Squillero, and M. Violante. On the test of microprocessor ip cores. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 209–213, 2001. [Pg.4], [Pg.25]

[22] Nektarios Kranitis, Antonis Paschalis, Dimitris Gizopoulos, and Yervant Zorian. Instruction-based self-testing of processor cores. *Journal of Electronic Testing*, 19(2):103–112, 2003. [Pg.4], [Pg.5], [Pg.25], [Pg.27]

[23] S. Gurumurthy, S. Vasudevan, and J. A. Abraham. Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor. In *2006 IEEE International Test Conference*, pages 1–9, 2006. [Pg.4], [Pg.25]

[24] Loganathan Lingappan and Niraj K Jha. Satisfiability-based automatic test program generation and design for testability for microprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(5):518–530, 2007. [Pg.4], [Pg.25]

[25] Ying Zhang, Ahmed Rezine, Petru Eles, and Zebo Peng. Automatic test program generation for out-of-order superscalar processors. In *2012 IEEE 21st Asian Test Symposium*, pages 338–343. IEEE, 2012. [Pg.4], [Pg.6], [Pg.25], [Pg.26], [Pg.27]

[26] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001. [Pg.4], [Pg.25], [Pg.26]

[27] F Corno, G Cumani, M Sonza Reorda, and Giovanni Squillero. Efficient machine-code test-program induction. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, volume 2, pages 1486–1491. IEEE, 2002. [Pg.4], [Pg.25], [Pg.26], [Pg.27]

[28] Dimitris Gizopoulos, Antonis Paschalis, and Yervant Zorian. *Embedded processor-based self-test*, volume 28. Springer Science & Business Media, 2013. ISBN: 978-1-4020-2801-4. [Pg.4], [Pg.6], [Pg.12], [Pg.34]

[29] Edward J McCluskey. Built-in self-test techniques. *IEEE Design & Test of Computers*, 2(2):21–28, 1985. [Pg.4]

[30] Charles E Stroud. *A designers guide to built-in self-test*, volume 19. Springer Science & Business Media, 2006. [Pg.4]

[31] Li Chen, Srivaths Ravi, Anand Raghunathan, and Sujit Dey. A scalable software-based self-test methodology for programmable processors. In *Proceedings of the 40th annual Design Automation Conference*, pages 548–553. ACM, 2003. [Pg.4], [Pg.5], [Pg.14], [Pg.26], [Pg.27], [Pg.33], [Pg.34]

[32] Li Chen and Sujit Dey. Software-based self-testing methodology for processor cores. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(3):369–380, 2001. [Pg.4], [Pg.5], [Pg.14], [Pg.27], [Pg.33], [Pg.34]

[33] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis. Software-based self-testing of embedded processors. *IEEE Transactions on Computers*, 54(4):461–475, April 2005. [Pg.5], [Pg.6], [Pg.25], [Pg.26], [Pg.27], [Pg.34]

[34] Nektarios Kranitis, George Xenoulis, A Paschalis, Dimitris Gizopoulos, and Yervant Zorian. Application and analysis of rt-level software-based self-testing for embedded processor cores. In *Test Conference, 2003. Proceedings. ITC 2003. International*, volume 1, pages 431–440. IEEE. [Pg.6], [Pg.14], [Pg.33], [Pg.34]

[35] Dimitris Gizopoulos, Mihalis Psarakis, Miltiadis Hatzimihail, Michail Maniatakos, Antonis Paschalis, Anand Raghunathan, and Srivaths Ravi. Systematic software-based self-test for pipelined processors. *IEEE Transactions on Very Large Scale Integration Systems*, 16(11):1441–1453, 2008. [Pg.6], [Pg.25], [Pg.26], [Pg.28]

[36] Mihalis Psarakis, Dimitris Gizopoulos, Ernesto Sanchez, and Matteo Sonza Reorda. Microprocessor software-based self-testing. *IEEE Design & Test of Computers*, 27(3):4–19, 2010. [Pg.6], [Pg.25]

[37] Mauricio De Carvalho, Paolo Bernardi, Ernesto Sánchez, M Sonza Reorda, and Oscar Ballan. Increasing the fault coverage of processor devices during the oper-

ational phase functional test. *Journal of Electronic Testing*, 30(3):317–328, 2014. [Pg.6], [Pg.25]

[38] Andreas Riefert, Riccardo Cantoro, Matthias Sauer, Matteo Sonza Reorda, and Bernd Becker. On the automatic generation of sbst test programs for in-field test. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1186–1191. EDA Consortium, 2015. [Pg.6], [Pg.26], [Pg.27]

[39] Andreas Riefert, Riccardo Cantoro, Matthias Sauer, Matteo Sonza Reorda, and Bernd Becker. A flexible framework for the automatic generation of sbst programs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(10):3055–3066, 2016. [Pg.6], [Pg.26], [Pg.27]

[40] Ieee standard vhdl language reference manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages c1–626, Jan 2009. [Pg.8]

[41] Fulvio Corno, Ernesto Sánchez, Matteo Sonza Reorda, and Giovanni Squillero. Automatic test program generation: A case study. *IEEE Design & Test of Computers*, 21(2):102–109, 2004. [Pg.9], [Pg.13], [Pg.16], [Pg.27], [Pg.29], [Pg.30], [Pg.68], [Pg.69], [Pg.71], [Pg.99]

[42] Kenyon CY Mei. Bridging and stuck-at faults. *IEEE Transactions on Computers*, 100(7):720–727, 1974. [Pg.11]

[43] Kyoung Youn Cho, Subhasish Mitra, and Edward J McCluskey. Gate exhaustive testing. In *IEEE International Conference on Test, 2005.*, pages 7–pp. IEEE, 2005. [Pg.11]

[44] Xinyue Fan, Will Moore, Camelia Hora, Mario Konijnenburg, and Guido Gronthoud. A gate-level method for transistor-level bridging fault diagnosis. In *24th IEEE VLSI Test Symposium*, pages 6–pp. IEEE, 2006. [Pg.11]

[45] Irith Pomeranz and Sudhakar M Reddy. Functional test generation for delay faults in combinational circuits. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 687–694. IEEE Computer Society, 1995. [Pg.11]

[46] Juergen Alt and Udo Mahlstedt. Simulation of non-classical faults on the gate level-fault modeling. In *Digest of Papers Eleventh Annual 1993 IEEE VLSI Test Symposium*, pages 351–354. IEEE, 1993. [Pg.11]

[47] Kazutoshi Wakabayashi and Takumi Okamoto. C-based soc design flow and eda tools: An asic and system vendor perspective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1507–1522, 2000. [Pg.12]

[48] R. Leveugle and K. Hadjiat. Multi-level fault injections in vhdl descriptions: Alternative approaches and experiments. *Journal of Electronic Testing*, 19(5):559–575, Oct 2003. [Pg.12], [Pg.28], [Pg.39]

[49] Anton Karputkin and Jaan Raik. A synthesis-agnostic behavioral fault model for high gate-level fault coverage. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 1124–1127. IEEE, 2016. [Pg.12], [Pg.28], [Pg.29], [Pg.40], [Pg.53], [Pg.60]

[50] Fulvio Corno, Gianluca Cumani, M Sonza Reorda, and Giovanni Squillero. An rt-level fault model with high gate level correlation. In *High-Level Design Validation and Test Workshop, 2000. Proceedings. IEEE International*, pages 3–8. IEEE, 2000. [Pg.12], [Pg.28], [Pg.29], [Pg.40], [Pg.60]

[51] M Karunaratne, A Sagahayroon, and S Prodhuturi. Rtl fault modeling. In *Circuits and Systems, 2005. 48th Midwest Symposium on*, pages 1717–1720. IEEE, 2005. [Pg.12], [Pg.28], [Pg.29], [Pg.40], [Pg.60]

[52] Fulvio Corno, Ernesto Sánchez, and Giovanni Squillero. Evolving assembly programs: how games help microprocessor validation. *IEEE Transactions on Evolutionary Computation*, 9(6):695–706, 2005. [Pg.13], [Pg.28], [Pg.29], [Pg.30], [Pg.68], [Pg.69], [Pg.71]

[53] Ernesto Sánchez, Matteo Sonza Reorda, and Giovanni Squillero. Efficient techniques for automatic verification-oriented test set optimization. *International Journal of Parallel Programming*, 34(1):93–109, 2006. [Pg.13], [Pg.28], [Pg.29], [Pg.30], [Pg.68], [Pg.69], [Pg.71], [Pg.99]

[54] Andreas Apostolakis, Mihalis Psarakis, Dimitris Gizopoulos, and Antonis Paschalis. Functional processor-based testing of communication peripherals in systems-on-chip. *IEEE transactions on very large scale integration (VLSI) systems*, 15(8):971–975, 2007. [Pg.14], [Pg.33], [Pg.34]

[55] Ying Zhang and Krishnendu Chakrabarty. Fault recovery based on checkpointing for hard real-time embedded systems. In *Proceedings 18th IEEE Symposium on*

Defect and Fault Tolerance in VLSI Systems, pages 320–327, Nov 2003. [Pg.14], [Pg.33]

[56] Ying Zhang and Krishnendu Chakrabarty. A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(1):111–125, 2005. [Pg.14], [Pg.33]

[57] George Xenoulis, Dimitris Gizopoulos, Mihalis Psarakis, and Antonis Paschalis. Instruction-based online periodic self-testing of microprocessors with floating-point units. *IEEE Transactions on Dependable and Secure Computing*, 6(2):124–134, 2008. [Pg.14], [Pg.33]

[58] P. Pop, V. Izosimov, P. Eles, and Z. Peng. Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(3):389–402, March 2009. [Pg.14], [Pg.33]

[59] O. Heron, J. Guilhemsang, N. Ventroux, and A. Giulieri. Analysis of on-line self-testing policies for real-time embedded multiprocessors in dsm technologies. In *2010 IEEE 16th International On-Line Testing Symposium*, pages 49–55, July 2010. [Pg.14], [Pg.33]

[60] Paolo Bernardi, Riccardo Cantoro, L Ciganda, Ernesto Sánchez, M Sonza Reorda, Sergio De Luca, Renato Meregalli, and Alessandro Sansonetti. On the in-field functional testing of decode units in pipelined risc processors. In *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 299–304. IEEE, 2014. [Pg.14], [Pg.33]

[61] Boyang Du, Ernesto Sánchez, M Sonza Reorda, J Perez Acle, and Anton Tsertov. Fpga-controlled pcba power-on self-test using processor's debug features. In *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 1–6. IEEE, 2016. [Pg.14], [Pg.33]

[62] Riccardo Cantoro, Davide Piumatti, Paolo Bernardi, Sergio De Luca, and Alessandro Sansonetti. In-field functional test programs development flow for embedded fpus. In *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 107–110. IEEE, 2016. [Pg.14], [Pg.33]

[63] Nektarios Kranitis, George Xenoulis, Dimitris Gizopoulos, A Paschalis, and Yervant Zorian. Low-cost software-based self-testing of risc processor cores. *IEE Proceedings-Computers and Digital Techniques*, 150(5):355–360, 2003. [Pg.16], [Pg.31], [Pg.89]

[64] Dimitris Gizopoulos. Low-cost, on-line self-testing of processor cores based on embedded software routines. *Microelectronics journal*, 35(5):443–449, 2004. [Pg.16], [Pg.31], [Pg.34]

[65] R Cantoro, E Cetrulo, E Sanchez, M Sonza Reorda, and A Voza. Automated test program reordering for efficient sbst. In *2017 32nd Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6. IEEE, 2017. [Pg.16], [Pg.31], [Pg.32], [Pg.89]

[66] Prakash Rashinkar, Peter Paterson, and Leena Singh. *System-on-a-chip verification: methodology and techniques.* Springer Science & Business Media, 2007. [Pg.23], [Pg.24]

[67] Luc Séméria and Abhijit Ghosh. Methodology for hardware/software co-verification in c/c++. In *Proceedings 2000. Design Automation Conference.(IEEE Cat. No. 00CH37106)*, pages 405–408. IEEE, 2000. [Pg.23]

[68] Prakash Rashinkar, Peter Paterson, and Leena Singh. System-level verification. *System-on-a-Chip Verification: Methodology and Techniques*, pages 45–66, 2002. [Pg.24]

[69] Vishnu A Patankar, Alok Jain, and Randal E Bryant. Formal verification of an arm processor. In *Proceedings Twelfth International Conference on VLSI Design.(Cat. No. PR00013)*, pages 282–287. IEEE, 1999. [Pg.24]

[70] Subhasish Mitra, Sanjit A Seshia, and Nicola Nicolici. Post-silicon validation opportunities, challenges and recent advances. In *Proceedings of the 47th Design Automation Conference*, pages 12–17. ACM, 2010. [Pg.24]

[71] CH-P Wen, Li-C Wang, and Kwang-Ting Cheng. Simulation-based functional test generation for embedded processors. *IEEE Transactions on Computers*, 55(11):1335–1343, 2006. [Pg.26]

[72] Jian Shen and Jacob A Abraham. Native mode functional test generation for processors with applications to self test and design validation. In *Proceedings*

International Test Conference 1998 (IEEE Cat. No. 98CH36270), pages 990–999. IEEE, 1998. [Pg.26]

[73] Ismet Bayraktaroglu, Jim Hunt, and Daniel Watkins. Cache resident functional microprocessor testing: Avoiding high speed io issues. In *2006 IEEE International Test Conference*, pages 1–7. IEEE, 2006. [Pg.26]

[74] Praveen Parvathala, Kaila Maneparambil, and William Lindsay. Frits-a microprocessor functional bist method. In *Proceedings. International Test Conference*, pages 590–598. IEEE, 2002. [Pg.26]

[75] Fulvio Corno, Gianluca Cumani, M Sonza Reorda, and Giovanni Squillero. Evolutionary test program induction for microprocessor design verification. In *Proceedings of the 11th Asian Test Symposium, 2002.(ATS'02).*, pages 368–373. IEEE, 2002. [Pg.26], [Pg.27]

[76] N. Kranitis, G. Xenoulis, A. Paschalis, D. Gizopoulos, and Y. Zorian. Application and analysis of rt-level software-based self-testing for embedded processor cores. In *International Test Conference, 2003. Proceedings. ITC 2003.*, volume 1, pages 431–440, Sep. 2003. [Pg.27]

[77] Paolo Bernardi, Edgar Ernesto SÁnchez Sanchez, Massimiliano Schillaci, Giovanni Squillero, and Matteo Sonza Reorda. An effective technique for the automatic generation of diagnosis-oriented programs for processor cores. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):570–574, 2008. [Pg.27]

[78] Sankaranarayanan Gurumurthy. *Automatic generation of instruction sequences for software-based self-test of processors and systems-on-a-chip*. The University of Texas at Austin, 2008. [Pg.27]

[79] V Mani and Nikil Sathish. A new approach for generation of test program for detection of hardware fault in vliw processor. 2016. [Pg.27]

[80] F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero. Fully automatic test program generation for microprocessor cores. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 1006–1011, March 2003. [Pg.27]

[81] Fawnizu Azmadi Hussin, Nor Hisham Hamid, Noohul Basheer Zain Ali, et al. Automatic generation of test instructions for structural faults in processor cores

using satisfiability. In *2013 International SoC Design Conference (ISOCC)*, pages 388–391. IEEE, 2013. [Pg.27]

[82] F Corno, E Sanchez, M Sonza Reorda, and G Squillero. Automatic test generation for verifying microprocessors. *IEEE Potentials*, 24(1):34–37, 2005. [Pg.27]

[83] Paolo Bernardi, Ernesto Sánchez, Massimiliano Schillaci, Giovanni Squillero, and Matteo Sonza Reorda. An evolutionary methodology to enhance processor software-based diagnosis. In *2006 IEEE International Conference on Evolutionary Computation*, pages 859–864. IEEE, 2006. [Pg.27]

[84] Ying Zhang, Huawei Li, and Xiaowei Li. Software-based self-testing of processors using expanded instructions. In *2010 19th IEEE Asian Test Symposium*, pages 415–420. IEEE, 2010. [Pg.27]

[85] Artjom Jasnetski, Raimund Ubar, and Anton Tsertov. On automatic software-based self-test program generation based on high-level decision diagrams. In *2016 17th Latin-American Test Symposium (LATS)*, pages 177–177. IEEE, 2016. [Pg.27]

[86] Fulvio Corno, M Sonza Reorda, Giovanni Squillero, and Massimo Violante. On the test of microprocessor ip cores. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 209–213. IEEE, 2001. [Pg.27]

[87] Ernesto Sánchez, M Reorda Reorda, Giovanni Squillero, and Massimo Violante. Automatic generation of test sets for sbst of microprocessor ip cores. In *Proceedings of the 18th annual symposium on Integrated circuits and system design*, pages 74–79. ACM, 2005. [Pg.27]

[88] Davide Sabena, Luca Sterpone, and Matteo Sonza Reorda. On the automatic generation of software-based self-test programs for functional test and diagnosis of vliw processors. In *IFIP/IEEE International Conference on Very Large Scale Integration-System on a Chip*, pages 162–180. Springer, 2012. [Pg.27]

[89] M. Schlzel, T. Koal, S. Rder, and H. T. Vierhaus. Towards an automatic generation of diagnostic in-field sbst for processor components. In *2013 14th Latin American Test Workshop - LATW*, pages 1–6, April 2013. [Pg.27]

[90] Tai-Hua Lu, Chung-Ho Chen, and Kuen-Jong Lee. Effective hybrid test program development for software-based self-testing of pipeline processor cores. *IEEE*

Transactions on Very Large Scale Integration (VLSI) Systems, 19(3):516–520, 2011. [Pg.27], [Pg.30]

[91] Ján Hudec and Elena Gramatová. An efficient functional test generation method for processors using genetic algorithms. *Journal of Electrical Engineering*, 66(4):185–193, 2015. [Pg.28], [Pg.30], [Pg.69]

[92] Nektarios Kranitis, Andreas Merentitis, George Theodorou, Antonis Paschalis, and Dimitris Gizopoulos. Hybrid-sbst methodology for efficient testing of processor cores. *IEEE Design & Test of Computers*, 25(1):64–75, 2008. [Pg.30]

[93] Aymen Touati, Alberto Bosio, Patrick Girard, Arnaud Virazel, Paolo Bernardi, and M Sonza Reorda. An effective approach for functional test programs compaction. In *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 1–6. IEEE, 2016. [Pg.31], [Pg.32]

[94] R Cantora, E Sanchez, M Sonza Reorda, Giovanni Squillero, and E Valea. On the optimization of sbst test program compaction. In *2017 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–4. IEEE, 2017. [Pg.31], [Pg.32]

[95] Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. Intermittent hardware errors recovery: Modeling and evaluation. In *2012 Ninth International Conference on Quantitative Evaluation of Systems*, pages 220–229. IEEE, 2012. [Pg.33], [Pg.34], [Pg.35], [Pg.110]

[96] Michael Nicolaidis and Yervant Zorian. On-line testing for vlsia compendium of approaches. *Journal of Electronic Testing*, 12(1-2):7–20, 1998. [Pg.34]

[97] Nektarios Kranitis, Andreas Merentitis, N Laoutaris, George Theodorou, A Paschalis, Dimitris Gizopoulos, and Constantin Halatsis. Optimal periodic testing of intermittent faults in embedded pipelined processor applications. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 65–70. European Design and Automation Association, 2006. [Pg.34]

[98] Dimitris Gizopoulos. Online periodic self-test scheduling for real-time processor-based systems dependability enhancement. *IEEE Transactions on Dependable and Secure Computing*, (2):152–158, 2009. [Pg.34], [Pg.35], [Pg.36], [Pg.107], [Pg.117]

[99] Antonis Paschalis and Dimitris Gizopoulos. Effective software-based self-test strategies for on-line periodic testing of embedded processors. *IEEE Transactions on Computer-aided design of integrated circuits and systems*, 24(1):88–99, 2005. [Pg.34]

[100] George Xenoulis, Dimitris Gizopoulos, Mihalis Psarakis, and Antonis Paschalis. Instruction-based online periodic self-testing of microprocessors with floating-point units. *IEEE Transactions on Dependable and Secure Computing*, 6(2):124–134, 2009. [Pg.34]

[101] Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. Characterizing the impact of intermittent hardware faults on programs. *IEEE Transactions on Reliability*, 64(1):297–310, 2015. [Pg.34], [Pg.35]

[102] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011. [Pg.109]

[103] Miloš Stanisavljević, Alexandre Schmid, and Yusuf Leblebici. *Reliability of nanoscale circuits and systems: methodologies and circuit architectures.* Springer Science & Business Media, 2010. [Pg.110]

[104] Ying Zhang and Krishnendu Chakrabarty. A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(1):111–125, 2006. [Pg.111]