SCALABLE FUNCTIONAL VALIDATION OF NEXT GENERATION SoCs

BY

DEBJIT PAL

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Doctoral Committee:

Associate Professor Shobha Vasudevan, Chair
Professor Wen-mei Hwu
Professor Deming Chen
Professor Sarita V. Adve
Dr. Avi Ziv, IBM Research, Haifa

ABSTRACT

System-on-Chips (SoCs) constitutes the primary backbone of modern embedded computing devices including many safety-critical applications *e.g.*, autonomous vehicles, health care systems. The presence of any undetected bugs in these systems would have aberrant cost both in terms of safety and reliability and can cause loss of property or life. Hence, SoC validation is a crucial task to ensure the functional correctness of an SoC. The sheer size, presence of hundreds of concurrently executing heterogeneous IPs, vertical integration of SoC components *e.g.*, hardware/firmware/software to realize multiple functionality, and application-level relevance of components present a new spectrum of validation challenges that have rendered the traditional microprocessor validation paradigm moot in the context of SoC validation. The challenges include observability enhancement and debug and diagnosis under the constraint of vertical integrations, identifying high-quality verification artifacts among others. In industrial practice, SoC validation is a manual, unsystematic, and ad hoc process that heavily relies on the expertise and the creativity of the validator. Consequently, there is an urgent need to develop scalable and efficient algorithms of industrial relevance to address this massive ongoing challenge of SoC validation.

This dissertation makes contributions to both post-silicon and pre-silicon validation of SoCs, with highly impactful contributions to next-generation post-silicon SoC validation. We use top-down analysis, a higher level of abstraction, and application relevance as the key ideas to automate post-silicon observability enhancement for industrial scale SoCs and scale observability to design that is more than 300× the size of designs that have been presented in the academic literature so far. Our observability enhancement solution can be applied at the netlist-level, behavioral level, and at the system-wide application level to select high-quality signals that are most beneficial for post-silicon debug and diagnosis. We apply a feature engineering based ma-

chine learning technique on the observed signal data to develop an automatic, scalable, and efficient post-silicon debug and diagnosis solution. The key idea is to learn the correct and erroneous design behavior automatically from trace data without prior design knowledge. We believe our debugging solution can automate post-silicon debug and diagnosis, where manual debugging is the norm. The quality of SoC verification and validation heavily depends on the quality of verification artifacts *e.g.*, assertions. To automate and expedite identification of high-functional coverage assertions that are useful for regression analysis, localization, etc., we have also developed a comprehensive ranking scheme for assertions. The key idea is to identify assertions that capture important design behaviors by analyzing the design source code.

Our SoC validation solutions are scalable and efficient. We consistently show orders of magnitude speedup improvements over the state-of-the-art while objectively improving quality of results. We have shown that going forward application-level analysis is the key to scale post-silicon validation to industrial scale SoCs. Our proposed validation solutions can plug into the existing industrial validation process to introduce automation in the current unsystematic, ad hoc, manual settings with multiple order of magnitudes of benefit.

To my parents, for their love and support, to my best friend Sai, for her advice and unwavering support, to all MagentaMinions, for connecting me to my family, and to Prof. Shobha Vasudevan, for being my friend, philosopher, and guide.

ACKNOWLEDGMENTS

As I start to write my doctoral dissertation, I look back over my academic journey at the University of Illinois at Urbana-Champaign (UIUC). Although only my name appears on the top of this dissertation, in reality, it is a culmination of advice, encouragement, and continuous support of many people at and outside of UIUC. I would like to use this space to express my gratitude to all those people who extended their unwavering support and kind help.

First and foremost, I would like to express my sincere gratitude to my advisor Professor Shobha Vasudevan. I feel very fortunate and privileged to have her as my doctoral advisor. With her patience, continuous encouragement, motivation, technical guidance, and demand for thoroughness, I have seen my professional growth and achievements in the past few years.

Since, I met Professor Vasudevan on November 14, 2012, she generously devoted time and extended her technical expertise to help me to excel in my studies and research activities. Her endless quest for perfection and attention to even small details helped me to achieve better results and to demonstrate the impact of my research both in industry and academia. Whenever I lacked the confidence to solve a problem, her reassuring voice always told me "Debjit, you can do better." Her encouragement reenergized me to work harder and exceed my own expectations. During my doctoral study, I made many mistakes. Instead of scolding me, Professor Vasudevan offered her guidance and provided me the necessary freedom and time to learn from my own mistakes. Any word will fall short to express my humble gratitude to her. Yet, thank you, Professor Vasudevan, for all your lessons and help. I will forever be your student and cherish this advisor-advisee relationship.

Second, I would like to express my sincere gratitude to my mentor Professor Sandip Ray of the University of Florida, Gainesville. He proposed two exciting problems for me to explore. During that collaboration, he not only offered his industry knowledge and technical expertise, but numerous tech-

nical discussions with him helped me to grasp the crux of the problem and the shortcomings of existing state-of-the-art solutions. His timely feedback and constructive comments inspired me and also made my research more practical and meaningful at the industrial scale. Over the past few years, apart from research, we have become very good friends.

My sincere thanks to Dr. Flavio M. de Paula of IBM Austin for numerous insightful discussions on one of my research problems. I would also like to thank Farhan Rahman and Dr. Sankar Gurumurthy for providing a year-long internship opportunity at AMD Austin. I would also like to thank various university officials including Jennifer at the ECE graduate office, Jenny at the CSL editorial office, and Carol in CSL. Without their precious support, it would have been impossible to conduct this research.

I would also like to thank my distinguished doctoral dissertation committee members Professor Wen-mei Hwu, Professor Deming Chen, Professor Sarita V. Adve, and Dr. Avi Ziv from IBM Research, Haifa for their insightful comments and suggestions on my research work.

I would also like to thank my best friend Sai for offering her unwavering support, advice, and motivation. She is the oasis in this otherwise corn desert of Illinois. We first met in January 2013 in a class and then she joined my research group. In no time, we became very close friends. Our endless chit-chats, detailed discussion over SnowStorm at Jarlings are possibly the best memories that I will cherish forever. Even today, when I am feeling low, when I am stuck, I know Sai is just a text away to uplift my spirit. Without her, memories of UIUC go bleak. I do not think a word "thank you" can do justice to our friendship, yet thank you, Sai for all your help, support, and encouragement.

I would also like to thank my fellow labmates Rui, Tian, Abhishek, Spencer, Lingyi, Sam, Greg, Subho, Saurav, and Arjun for the stimulating discussions on study, research, graduate life, and for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last couple of years. I also would like to thank my two close friends – Mousumi Guha and Rashi Ahuja for listening to me and standing by me.

Last but not least, I would like to thank my parents for their unconditional love and endless support throughout my academic career. Without their encouragement and sacrifice, I would have never been able to finish my doctoral study.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 SoC verification and validation

The ubiquitous role of System-on-Chips (SoCs) in modern societies, as well as the increasing reliance on SoCs in safety critical applications like autonomous vehicles and health care has unprecedented implications for their safety and reliability. The cost of an undetected bug in these systems is much higher than in traditional processor systems – it may not simply mean an erroneous result or reduced performance; it could mean the loss of property or life. Even the benign effects of a functional bug in say a navigation system, an IoT device or a smart phone, could be very disruptive and inconvenient.

Verification and validation, or the process of ensuring functional correctness, therefore, is more critical to the SoC life cycle now than ever before. There are two phases in SoC validation (c.f., Figure 1.1 and Figure 1.2). One is the pre-silicon verification phase, and the other is the post-silicon validation phase. Pre-silicon verification, as is well known, is critically important to the functionality of the SoC and ensures the absence of design bugs. Post-silicon validation of SoCs, is the "gating stage" before a decision is made to continue mass fabrication or discard the SoC. The importance of both types of verification in our society is significant,[1] due to the impact it can directly have on our lifestyle and productivity.

Both these phases of verification have always been massively challenging. In addition, the SoC design paradigm presents a new spectrum of pre-silicon and post-silicon validation challenges. This includes checking of the communication fabric between IPs, communication protocols among the hundreds of Intellectual Property (IP) blocks, concurrency related violations (like dead-

[1] As a thought experiment, if the Samsung Note line of devices had been discarded due to a bug trend in post-silicon, how would that have affected the popularity of the Android devices?

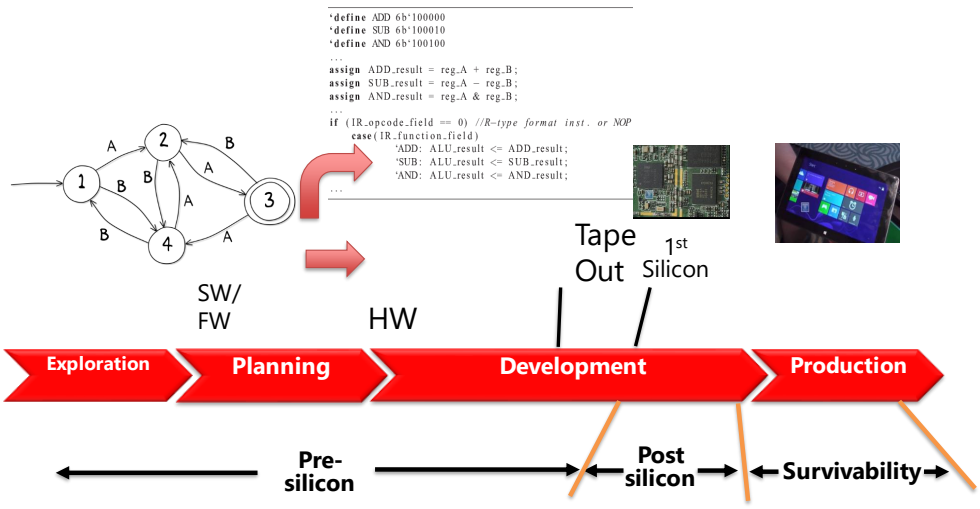


Figure 1.1: High-level categorization of different components of SoC design life cycle. Tape out refers to the time point when the pre-silicon design is mature enough for first silicon [1].

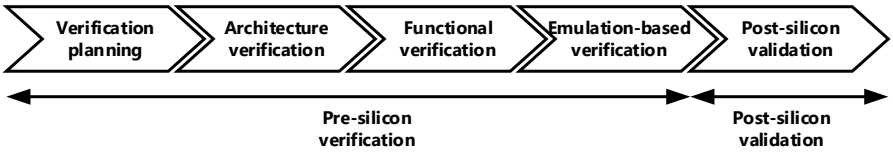


Figure 1.2: SoC validation life cycle [2].

locks), and application-level relevance of components.

An SoC (c.f., Figure 1.3) consists of billions of transistors (*e.g.*, Qualcomm SnapDragon 855 contains more than 6.9 billion transistors [3], Samsung Exynos 9820 contains more than 7 billion transistors [4]) and more than a hundred of pre-verified hardware functional blocks called IPs (*e.g.*, Qualcomm SnapDragon 855 contains more than 150 IP blocks [3]) to realize tens of different functionality. Concurrent execution of different IP blocks creates a massive design state space consisting of the order of 10^{80} states which is impossible to explore exhaustively as required for verification. Due to the sheer size of the state space, integration of multiple types of design functionality, and rapid shrinking of time-to-market (less than one year), SoC validation is an extremely difficult and challenging task.

In spite of decades-long maturity of the hardware verification research and

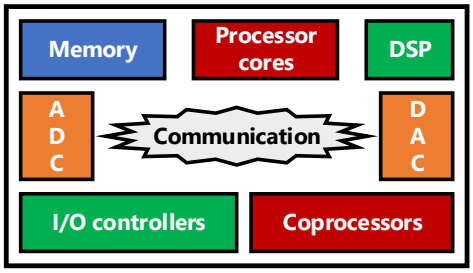


Figure 1.3: An SoC integrates hundreds of IPs on a chip that includes one or more processor cores, digital signal processors (DSPs), multiple co-processors and accelerators, I/O controllers, analog-to-digital (ADC) and digital-to-analog (DAC) converters that are connected via a communication fabric [1].

associated electronic design automation (EDA) tools, scaling verification to the needs of modern SoCs is still a formidable challenge [2]. We discuss a few key challenges of contemporary SoC validation. While some of the challenges are driven by complexity *e.g.*, tool scalability, other are driven by the needs of the rapidly changing design paradigm and the underlying technology.

Shrinking verification time: The disproportionate growth in number (of the order of a billion devices) of *connected* devices has resulted in a massive shrinkage in the system development life cycle, leaving low to no room for customized verification efforts.

Limited tool scalability: Scalability remains a crucial problem in effective application of verification technology, especially for formal verification techniques such as SAT checking and SAT modulo theories [5]. Simulation-based verification cost is also increasing due to explosive growth in the design state space of modern SoCs. Random simulation covers a tiny fraction of design state space whereas developing coverage-specific directed test is prohibitively costly.

Power management challenges: Power efficiency and low-power requirements for integrated circuits have been the main focus of modern SoC designs. Several well-researched technologies, *e.g.*, clock gating, power gating have been developed to address this problem. Addition of these features significantly convolutes verification activities. Tens of power domains and hundreds of power modes create a colossal verification challenge (of ensuring that design is functional for all possible power modes) for both formal

verification and simulation-based verification.

Security and functional safety: Security and privacy have become critical requirements for electronic devices in the modern era. Unfortunately poor specification and understanding leave many security holes. Often, one sorts to *hackathons* or directed targeted hacking of the device to identify security threats.

Hardware/software co-verification: In the era of microprocessors and application software, it was easy to sperate concerns between hardware and software verification activities. Recently, with increasing trend of defining critical functionality in software, it is difficult, often impossible to define a coherent specification of the hardware without the associated firmware or software running. Before SoC era, hardware and software were traditionally developed independently. In contrast, the strong coupling between software and hardware makes it inevitable that we develop and validate them concurrently. This requirement essentially makes contemporary SoC validation a hardware/software co-verification problem.

Over the last few years several industrial studies by Foster [6, 7] identify following critical broad trends in SoC verification.

1. SoC verification represents bulk of the effort in the SoC design cycle, incurring a cost of *up to 80%* (*on average about 57%*) of the total project time.

2. On an average *two* silicon spins are needed before an SoC is productized. Note that for a hardware/software vertically integrated system such as modern SoC, this entails to one spin for catching hardware problems and another spin catching hardware/software interaction issues. This underlines the critical role of pre-silicon verification to ensure that here is not critical gating issues during post-silicon validation.

This dissertation makes contributions to both pre-silicon verification and post-silicon validation of SoCs, with highly impactful contributions to post-silicon SoC validation.

1.2 Phases of SoC validation

We now describe the two phases of SoC validation.[2]

1.2.1 Pre-silicon SoC verification

Pre-silicon SoC verification is performed on behavioral models written in hardware description languages (HDLs) *e.g.*, Verilog [8], VHDL [9], SystemVerilog [10]. The primary objective of the pre-silicon verification is *logic and functional verification*, *timing verification*, etc. The principal advantage of pre-silicon verification is that it is a *white-box validation* method where the validator has complete observability and controllability of the internal design signals. Hence, during design simulation, any internal design signals can be monitored as needed for verification. On the other hand, simulation is extremely slow, often of the order of *a few hundred cycles per second*. In addition, heterogeneity and concurrent execution of multiple different IPs prevent the execution of real-world use cases on top of a behavioral model *e.g.*, booting an operating system using a register transfer level (RTL) model will take *approximately two to three years of CPU time*. This causes many hard to detect deep state space bugs to escape pre-silicon verification [1, 11, 12, 13].

1.2.2 Post-silicon SoC validation

Post-silicon SoC validation refers to the validation that is done after the first silicon is available. Post-silicon allows execution at the target clock speed making it approximately a billion times ($10^9\times$) faster than the pre-silicon simulation. Hence, real-world use cases *e.g.*, booting an operating system, can be executed which allows deep design state space exploration. Consequently, the primary objective of the post-silicon validation is *to detect and diagnose hard to detect deep state space bugs* such that those bugs do not escape to the final product. Post-silicon validation acts as the final gateway before mass production for a system is committed. Furthermore, due to the physical nature of the validation vehicle (*i.e.*, actual silicon rather

[2]In this dissertation, we call both pre-silicon verification and pot-silicon validation as validation.

than a computer model), it becomes possible to validate the artifact for non-functional characteristics such power consumption, temperature tolerance, and electrical noise margin. On the other hand, it is considerably more complex to control and/or observe the execution of silicon than that of an RTL simulator. In RTL simulator, virtually any internal design signal is observable. In silicon one can only observe a few hundred among millions. Additionally, in pre-silicon platform, changing observability or controllability to facilitate more control would require a compilation (which can take hours but a feasible option) whereas for silicon, it requires a silicon respin, which is often infeasible.

1.3 Challenges in post-silicon and pre-silicon validation

Validation is the most resource and time intensive phase in the life cycle of modern software, hardware or embedded systems, requiring 70% of the time and teams that are three times the size of design teams [6, 7, 11, 12, 13]. We touch upon some of the challenges in SoC validation.

1.3.1 Challenges in post-silicon validation

Post-silicon validation is often termed as a *black art* in industry. This is due to i) the inherently difficult, "black box" verification it entails, ii) the lack of on-chip observability and controllability due to a perennial economy in area, iii) lack of principled methods to plan, utilize, and channelize available resources, iv) inadequate a priori planning and top-down vertical communication across higher and lower levels of design. This is over and above the fundamental malaise of all verification/validation problems, a *battle of scale* against massive, complex next-generation SoC designs. Some challenges are specific to an industrial environment, while others are more common across the board.

In post-silicon validation, *limited observability* and *controllability* are key obstacles that seriously hinders observation of various internal design signals during post-silicon execution. Hence, to observe the internal design signals during post-silicon execution, important internal design signals need to be

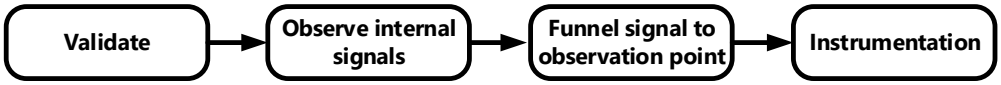


Figure 1.4: Motivation of hardware tracing.

Figure 1.5: Post-silicon validation problem.

funneled at an observation point *e.g.*, debug pins (c.f., Figure 1.4). Consequently, the important design signals need to be instrumented before the first silicon is available as any change post first silicon would need a costly respin, which is often infeasible.

The above necessity to observe and instrument internal design signals begs the question of *what part of the chip should be observed?* as shown in Figure 1.5. In industry, this problem is commonly known as *hardware tracing*. Due to extremely limited availability of on-chip storage (typically less than 10% of the total die area), and limited availability of external debug pins (typically of the order of 100 pins), only a few hundred among millions of internal giga-hertz signals can be traced. Selection of a few hundred signals among millions makes hardware tracing a *colossal optimization* problem.

Once a post-silicon execution fails (*e.g.*, hangs, crash) and a bug is detected (c.f., Figure 1.5), the validator needs to investigate the traced signal values to diagnose the potential root causes of the failure. This begs the question *what went wrong in the execution?* as shown in Figure 1.5. This problem is called post-silicon debug and diagnosis.

Diagnosing an SoC post-silicon failure is extremely difficult due to the following reasons. i) Multiple instances of communication protocols execute *concurrently* [14, 15, 16] among different IPs which results in a mammoth interleaved design state space.[3] Investigating such mammoth design state space manually is practically impossible. ii) Post-silicon execution traces

[3]As a thought experiment, if two instances of each of the three different protocols each of which has four states execute concurrently, it will result in a design state space consisting of $4^6 = 4096$ states.

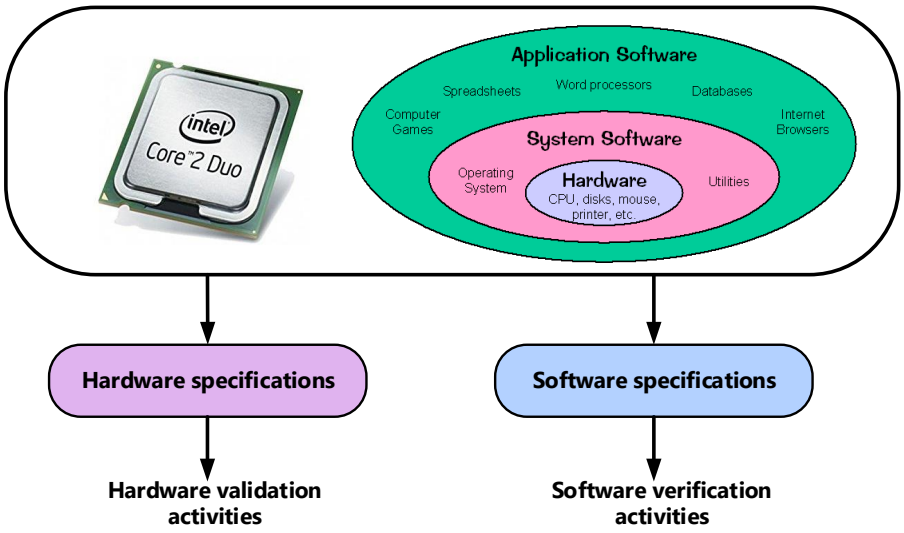


Figure 1.6: Traditional microprocessor validation paradigm where hardware and softwares are designed, developed, and verified independently.

usually span over *millions of clock cycles* and consists of *hundreds of protocol messages interleaved*. These make *temporal* and *spatial localization* of post-silicon failures manually a tedious and error-prone task. iii) Unlike pre-silicon diagnosis, the traditional notion of *error sequentiality* [1] does not hold good for post-silicon diagnosis, *i.e.*, one cannot use a **detect** $\rightarrow$ **diagnose** $\rightarrow$ **fix** cycle per post-silicon bug as it would require costly respin which is often infeasible.

In current industrial practice [11, 17], post-silicon debugging is unsystematic, ad hoc, and heavily relies on the acumen and expertise of the designer and the creativity of the validator. Diagnosing a post-silicon failure (*e.g.*, deadlock, hangs, crash) can take validation engineers *a few weeks* to *two months*, which often increases the time-to-market of the SoC.

1.3.2 Unique challenges in post-silicon validation of SoCs

Post-silicon validation has been well studied and researched for microprocessors [18, 19, 20, 21, 22]. The microprocessor is one among many hundreds of different IP components that make an SoC. The traditional processor validation paradigms (c.f., Figure 1.6) are inadequate to address the new spectrum of validation challenges that SoCs bring. In our "app" (application) driven

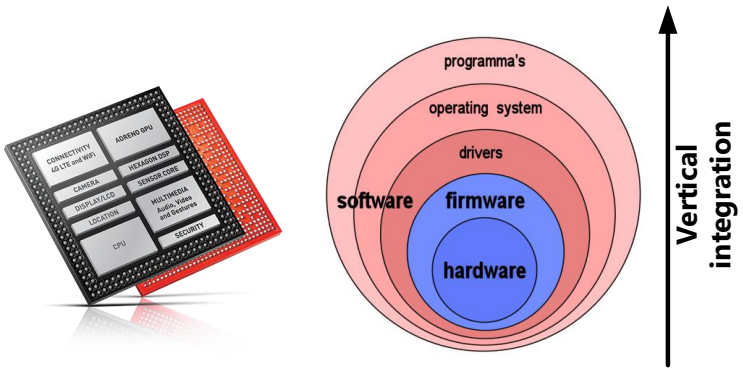


Figure 1.7: SoC post-silicon validation is a complex co-validation problem encompassing hardware, software, firmware, and peripherals.

societies where the principal role of the ubiquitous mobile device is to run diverse applications, the verification/validation problem is compounded [1]. Lines blur between hardware and software, as most functions can be implemented in both. Most applications use a particular combination of hardware, software and peripherals. This obscures the traditional notion of functional validation of software, hardware and peripherals as distinct entities. The vertical integration (c.f., Figure 1.7) of components on the basis of applications also presents a challenge for controlling and observing components in silicon, since the importance of a component may not be uniform across applications.

1.3.3 Challenges in pre-silicon verification

Pre-silicon verification of SoCs comprises a wide spectrum of activities *e.g.*, formal verification [23, 24, 25, 26, 27], simulation-based verification [28, 29, 30, 31], emulation-based verification [32, 33, 34], test generation [35, 36, 37, 38], transaction-level verification [39, 40], assertion-based verification [6, 29, 41, 42, 43, 44] among others.

The principle issue in most of these activities is the inherently complex nature of verification – the lack of computational capacity to search through and check the entire state space of a modern system. This issue of scalability manifests as a hindrance to most activities in verification.

Along with scalability, an important issue in verification is the necessity to

express specifications/properties/assertions about the system that need to be checked during the design and implementation of that system. Assertions are artifacts used to validate hardware designs throughout their life cycle. They are applied in formal verification, dynamic validation, runtime monitoring and coverage analysis. Assertion-based verification heavily depends on the quality of the assertions used.

To write good assertions, a verification engineer needs creativity and deep understanding of a design's functionality. Traditionally, writing good assertions has been known to be a very hard problem [45, 46, 47, 48, 49, 50]. Recent industrial studies [6, 7, 51] report that even after decades of research on assertion-based verification, writing good assertions is very challenging. Consequently, in an industrial setup, it requires manual inspection to identify high-functional coverage assertions that are useful for regression analysis, bug detection, and localization, etc.

1.4 Contributions of this dissertation to post-silicon validation

1.4.1 Value added to SoC validation

In our solutions, we have managed to scale current post-silicon observability technologies like hardware tracing to *more than* 300× the size of what has been presented in academic literature so far. We have scaled hardware tracing from small ISCAS89 benchmarks to the OpenSPARC T2 SoC (c.f., Figure 1.8), an industry scale SoC.

In post-silicon debug and diagnosis, where manual debugging is the norm, we present a completely automated, efficient solution. We could isolate *66%* more bugs and take *up to* 847× less time than manual debugging for the OpenSPARC T2.

We outline some insights that have emerged from our work, that can be used as principles and resources for next-generation post-silicon validation solutions.

1. **Functional context awareness**: The focus of our solution is on providing a methodology that minimizes the effort of debugging and is

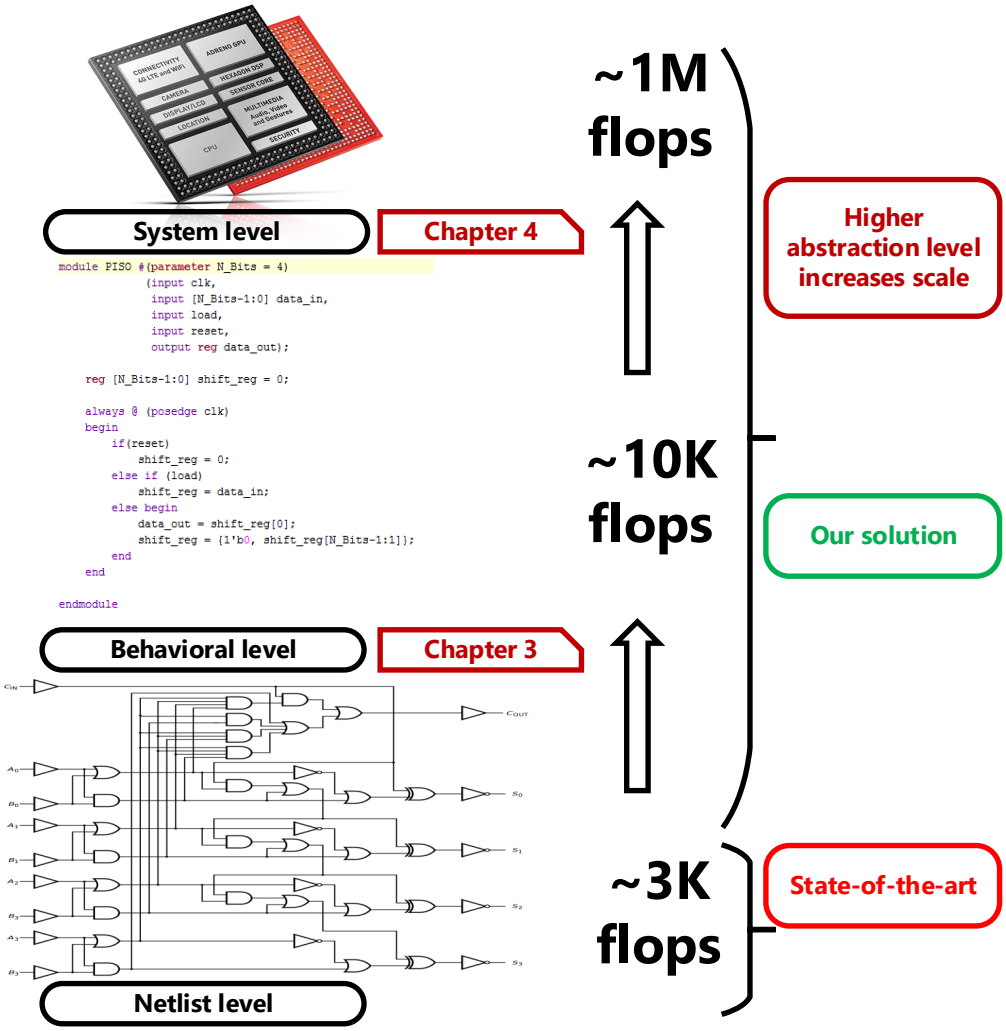


Figure 1.8: State-of-the-art solutions for hardware tracing select signals at the gate-level netlist. Our solution uses *abstraction* as the key idea to scale hardware tracing. We apply hardware tracing at the behavioral level and at the application level to scale hardware tracing to OpenSPARC T2 SoC which is 333× bigger than the designs on which state-of-the-art solutions work.

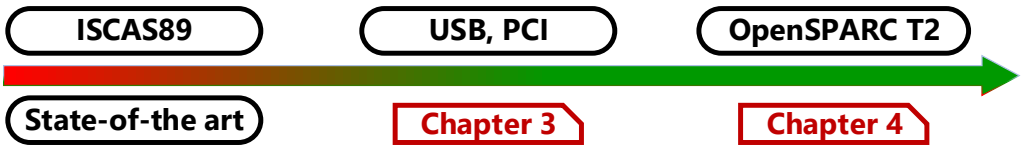


Figure 1.9: Scaling hardware tracing from ISCAS89 benchmarks to OpenSPARC T2 SoC via abstraction.

aware of the high level functional context. As such, we introduce a top-down methodology, where we model and analyze user scenarios or applications, and cut across abstraction levels to identify relevant observation artifacts.

2. **Scalability**: In addition to functional context, we make scalability an objective of our post-silicon debug solution. In doing so we depart from the prior art and "zoom out" to behavioral level (RTL) and the application level. Operating at the higher level of abstraction allows to scale the observability selection to industrial scale SoC that is many orders of magnitude bigger than the design on which state-of-the-art solutions work. We also demonstrate that this scale is beyond the capacity of current tracing approaches.

3. **Vertically integrated solutions**: An integrated picture of the failure in the presence of a detected bug is most valuable to debugging. Most traditional methods have focused on in-depth analysis at one layer (usually netlist or RTL), tending to over optimize for observability only at that level. This lacks big picture context, and is ineffective for debugging. The relevant components of observation that can present an integrated picture are not necessarily available from one layer, but need to be culled across different layers of abstraction. We present a post-silicon validation methodology that cuts across various layers of abstraction. Towards this, we model and analyze interacting components at the application, RTL and netlist levels. We believe that this cross-cutting approach aids the scalability in our solutions. In future research, the abstraction, if raised to firmware and software levels, can be made to scale further and be more robust.

4. **Feature engineering for learning buggy behavior**: One of our innovations is in the atypical use of machine learning in the automated diagnosis and debug solution. We define our diagnosis task as identifying buggy traces as "outliers" and bug-free traces as "normal" behavior, for which we seek to use unsupervised learning algorithms for outlier detection. Typical use of machine learning for outlier detection would involve the direct application of classification or clustering algorithms over trace data using the signals as raw features. Instead, we use the

approach of *feature engineering*, or the transformation of raw features into more sophisticated features by using domain specific operations. The engineered features are highly relevant to the diagnosis task, resulting in the classifiers identifying buggy traces accurately as outliers. They are also generic, *i.e.* they are transformations that can be applied to any hardware design. Our unsupervised approach is free of the labor associated with training. It is also able to detect bugs that could not be manually identified faster by orders of magnitude, as compared to manual debug. With more research that identifies more distinguishing features, the diagnosis can improve further in precision and be widely applied to system designs.

5. **Benchmark creation**: In our research and industrial collaborations, we have found that there is a widening gap between the state-of-the-art in academic research in post-silicon validation and the state-of-the-practice in industry [1, 12, 13]. This gap is probably due to the extensive and elaborate infrastructure that is required for post-silicon validation in industry. We believe that the innovative solutions from research community can have higher impact and adoption if an experimental testbed of industrial scale is used. Toward this goal, we have released our current post-silicon observability framework [52, 53, 54] for OpenSPARC T2 SoC which includes signal selection framework, synthesized netlist of different T2 design modules, constrained random testbenches, and signal-to-message conversion framework. We believe that this framework will help the academic post-silicon validation research to move beyond ISCAS89 benchmarks and will motivate and inspire academic researchers to propose scalable and efficient solutions of industrial relevance for post-silicon validation.

6. **Comprehensive evaluation and ranking for assertions**: A contribution that is of high value to pre-silicon verification is providing a methodology for comprehensively evaluating and ranking assertions. A big deterrent in the effective use of assertions by non-experts and designers in contemporary industry is in the lack of a figure of merit that captures the notion "how good is/are my assertion(s)?" We have developed a figure of merit for assertions in RTL designs that captures the importance of an assertion within a design and the extent of cover-

age achieved by the assertion. Such a comprehensive figure of merit is the first of its kind, resulting in an evaluation/ranking that is very close to human assessment. It is also computationally efficient and scalable to large designs.

We outline the specific technical problems addressed by this dissertation, as well as the impact of our solutions for each problem.

• **Hardware tracing at different abstractions**

Given the severity of the impact of missing necessary observability, there has been significant research in the "signal selection problem", *i.e.*, disciplined identification of traceable signals that can maximize the design visibility as necessary for post-silicon debugging, under observability restrictions.

Academic techniques for hardware tracing [55, 56, 57, 58, 59, 60, 61, 62] work at the gate-level netlist (c.f., state-of-the-art in Figure 1.8 and Figure 1.9), treat all signals equally, and use an irrelevant metric to select profitable signals for tracing. Consequently, they i) suffer from scalability issues and ii) select low-quality signals irrelevant for debugging. While there are significant differences in the specific approaches proposed, virtually all related work optimizes the same metric, called the *State Restoration Ratio* (SRR)[4] which takes a *myopic* view of the design and tends to lose critical information about functional relevance of the signals. In spite of its wide use as a de facto standard in signal selection research, SRR is a poor metric [63] for determining the quality of post-silicon trace signals. This casts serious doubts on the practical applicability of all related signal selection algorithms that are based on optimizing SRR.

We present the first hardware tracing solution (Chapter 3) that is applicable across different abstraction levels of the design. We repurposed Google's PageRank [64] algorithm for signal selection and exploit design structure and connectedness to guide signal selection. Our solution works at the netlist and at the behavioral level of hardware designs and scaled hardware tracing from designs containing approximately 3,000 flip-flops to designs containing more than 10,000 flip-flops. Our hardware tracing solution is highly scalable and computationally efficient which finished signal selection for designs containing up to 14,000 flip-flops and 76,000 logic elements with a runtime of only

[4]SRR measures the number of design states reconstructed from the signals observed: a set S of signals is considered superior to another set S' if more design states can be inferred from observing S than S'.

13 seconds and peak memory usage of 1.5GB. Further, the hardware tracing solution selected high-quality trace signals which achieved up to 50.94% more behavioral coverage and up to 7.3× more state restorability as compared to the signals selected by the state-of-the-art methods.

• **Application-level hardware tracing**

An expensive component of post-silicon SoC validation is application level use-case validation (c.f., Figure 1.8). In this activity, a validator exercises various target usage scenarios of the system (*e.g.*, for a smartphone, playing videos or surfing the Web, while receiving a phone call) and monitors for failures (*e.g.*, hangs, crashes, deadlocks, overflows, etc.). Use-case validation forms a key part of compatibility validation [1] and often takes weeks to months of validation time. Consequently, it is crucial to determine techniques to streamline this activity.

Each usage scenario involves interleaved execution of several protocols among IPs in the SoC design, *e.g.*, a usage scenario that entails receiving a phone call in a smartphone when the phone is asleep may constitute protocols among the antenna, power management unit, CPU, etc. To debug such a scenario, the validator typically needs to observe and comprehend the messages being sent by the constituent IPs. An effective way to do that is to use *hardware tracing*.

In response to current technology trends, application-level analysis dominates many research areas in hardware like specialized architectures, alternate computation models, etc. For verification/validation, application relevance could serve to ease the increasingly daunting challenges of scale. The primary reason for this is that the modern applications are very closely integrated with hardware, software, and peripherals. Without sacrificing verification of any part of the hardware, we argue for the modeling and analysis at the application level, and a top-down approach to post-silicon validation as opposed to a bottom-up approach as seen in the literature so far. Given that post-silicon validation tends to be a maze, we are arguing for navigating the maze with a divide-and-conquer approach, using the application space as a starting point.

We present the first hardware tracing solution (Chapter 4) that specifically targets use-case validation. We exploit available architectural collateral such as messages, transaction flows etc., to develop a targeted message selection

for hardware tracing. To make scalability an objective of the post-silicon debug solution, we operate at a higher-level of abstraction (application-level) that allows to scale hardware tracing to industrial-scale SoC containing multiple heterogeneous IPs that is bigger by a factor of 333× *as compared to the state-of-the-art published designs. Our traced signals are of high-quality that achieved up to 99% flow specification coverage, pruned up to 89% of candidate root causes in post-silicon failures, focused debugging to only 55% of participating IP-pairs, and localized failures to no more than 0.31% of paths.*

• **Automated debugging of post-silicon failures**

The SoC post-silicon debug and diagnosis problem is convoluted by the heterogeneity of the constituent IPs and the vertical integration of hardware/software/firmware components. Due to the concurrent execution of multiple flows in the different usage scenarios, extremely long execution traces (potentially spanning over millions of clock cycles), lack of bug reproducibility (due to on-chip asynchronous events, electrical effects), and lack of error sequentiality lead to an extremely time consuming, if not unachievable, post-silicon debug and diagnosis effort.

In current industrial practice [11, 12, 13], post-silicon debug and diagnosis is a manual, unsystematic, ad hoc process that primarily relies on the creativity of the validator. Beginning with first silicon, SoCs are executed at the target clock speed using various applications and a set of IP interface signals are traced. When execution fails with a hang, crash etc., manual debugging begins. During post-silicon debugging, the focus is on hard to detect deep state space functional bugs that escapes pre-silicon verification.

We present a scalable and efficient post-silicon bug diagnosis solution (Chapter 5) using machine learning and feature engineering. Our bug diagnosis solution can automatically diagnose a post-silicon failure by analyzing intrinsic characteristics of input data without requiring a prior design knowledge. We use feature engineering to transform input data in the machine learning space such that normal behaviors are close to each other and densely distributed whereas buggy behaviors are distant from normal behaviors and sparsely distributed. Our diagnosis solution diagnosed 66.7% more bugs and took up to 847× less diagnosis time as compared to the manual debugging to debug subtle and complex bugs on OpenSPARC T2 SoC. The diagnosing solution is highly effective that achieved a diagnosis precision of up to 0.769

with only up to 63 seconds of runtime and 508 MB of peak memory usage.

1.5 Contributions of this dissertation to pre-silicon verification

- **Assertion ranking**

Assertions are used in a wide spectrum of hardware design validation tasks, *e.g.*, formal verification, dynamic simulation-based verification, runtime monitoring, and emulation-based verification [6, 44, 45]. Identifying "good" assertions is the key to ensure high-quality assertion-based verification. A validation engineer needs a deep understanding of the design functionality to write good assertions. Even after decades of research on assertion-based verification, writing good assertions remains a formidable challenge both in academia [45, 46, 47, 48, 49, 50] and in the industry [6, 7, 51]. Consequently, in the current industrial setup, tedious and error-prone manual inspection is used to identify high-functional coverage assertions for regression analysis, bug detection, and diagnosis.

GoldMine [41, 43, 65] and other tools [42] automatically generate succinct assertions but do not provide a quantitative metric to evaluate the *goodness* of the assertions in terms of an assertion's design functionality coverage. Further, automatic methods [41, 42, 43, 65, 66, 67] often generate more assertions than can practically be examined by a human. Ranking of the most important assertions is essential if this technology is to be practicable.

The use cases of an assertion-ranking approach comprise situations where assertions are used and need to be examined by a human in the loop, *e.g.*, assertion ranking can be used to save and prioritize *debugging efforts* both in simulation-based and formal verification, to prevent a *formal verifier* from running into capacity constraints by including top-ranked assertions with high-behavioral coverage, identifying *a few high-behavioral coverage assertions* for simulation and emulation. In general, an assertion ranking technique can inform designers of any missing design behaviors and the *quality of the assertions* that they have written.

We present the first solution (Chapter 6) for assertion ranking using systematic RTL source code analysis. We model dependencies among design variables as a directed graph called a variable dependency graph. We define

assertion importance and assertion complexity metrics and use the dependency graph to algorithmically compute those two metrics. Our assertion ranking solution can identify presence of important design variables both in combinational and temporal assertions. The ranking solution ranks assertions higher that contain such important variables and cover critical design functionality paths. Prioritizing presence of important design variables helps our assertion ranking solution to rank assertions with good bug detectability at the top of the ranked list. Our analysis shows that top-ranked assertions from our assertion ranking solution can detect up to $1.5\times$ *more bugs per assertion as compared to a baseline algorithm [29].*

• **Functional debug of hardware designs**

With increasing complexity and versatility, verification, in particular debug and diagnosis have become the biggest bottleneck in the hardware design cycle. Debugging even a single bug can take several weeks to months [6, 7, 51]. During simulation of massive industrial-scale designs (with thousands of lines of RTL source code), a tremendous amount of simulation data (often in the order of several GBs) is generated. Hence localizing the root cause is tantamount to finding a needle in the haystack. Consequently, localization of the bug to any extent is valuable and can significantly slash debugging costs and efforts. Although state-of-the-art academic research [68, 69, 70, 71, 72, 73, 74, 75] and industrial debugging tools [76] aid "what-if" scenarios with visualizations, they fail to provide any localization of the root cause.

We present the first solution (Chapter 7) for assertion-based bug localization for RTL functional debugging. Our solution leverages the massive volumes of simulation trace data that is generated in typical verification environments to mine accurate symptoms of buggy behavior. Our solution identifies statistically relevant common symptoms across failing simulation traces and mapping these symptoms back to the corresponding design execution paths in the RTL source code. Our solution achieved precise localization to less than 5% of RTL source code and localized to simulation traces smaller by 80% as compared to the original failure trace. Our solution localized to small, focused, and functionally coherent high-importance code zones with importance of up to 0.857.

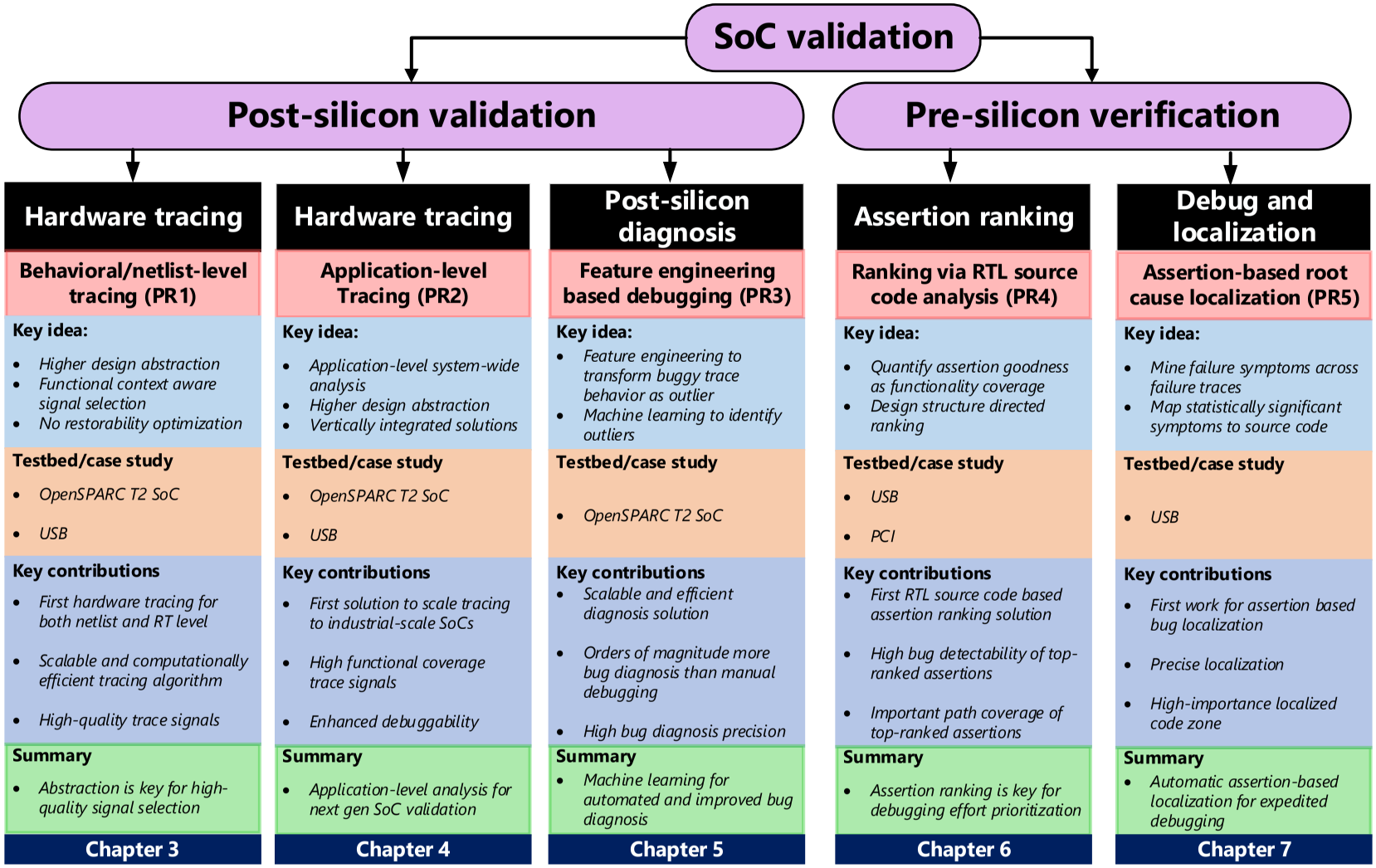


Figure 1.10: This dissertation addresses key problems in both SoC post-silicon validation and SoC pre-silicon verification, with an emphasis on SoC post silicon validation. PR1, PR2, PR3, PR4, and PR5 are the problem IDs that we will refer in the next chapters.

1.6 Dissertation outline

The remainder of this dissertation is organized as follows. Figure 1.10 shows a flowchart of the different key problems of SoC validation addressed in this dissertation.

In Chapter 2, we present the previous works that are closely related to the contributions of this dissertation.

In Chapter 3, we present a scalable and computationally efficient post-silicon trace signal selection technique [63, 77] that can be applied both at the netlist-level and the at behavioral-level (c.f., Figure 1.8) RTL.

In Chapter 4, we present a method for selecting trace messages at the application level (c.f., Figure 1.8) for diverse post-silicon use-case validation [78].

In Chapter 5, we present a scalable and computationally efficient method for diagnosing candidate root causes for post-silicon failures [79].

In Chapter 6, we present a systematic and efficient ranking method to quantify the goodness of an assertion [80, 81] using RTL source code analysis.

In Chapter 7, we present an automatic and scalable assertion-based statistical bug localization technique for pre-silicon debugging [82].

In Chapter 8, we summarize the work and conclude this dissertation.

In Chapter 9, we detail the tools that are developed as a part of this dissertation.

CHAPTER 2

RELATIONSHIP TO EXISTING WORK

In this chapter, we outline different activities and the goal of post-silicon validation followed by a detail survey of existing state-of-the-art techniques for both post-silicon validation and pre-silicon verification of SoCs. Then we study the principles of GoldMine that we will use for our pre-silicon solutions.

2.1 Post-silicon validation primer

2.1.1 Post-silicon validation activities

Post-silicon validation encompasses a diverse set of activities that include validation of both functional and timing behavior as well as non-functional requirements [1].

1. *Power-on-debug* is one of the first activities performed when a pre-production silicon arrives at the post-silicon validation lab. It includes a significant brainstorming component to come up with a *bare-bone system configuration* (by removing most of the complex features, *e.g.*, power management, security) such that the first silicon reliably powers on with the help of a custom debug board. A stable power-on can take a few days to a week. Once the power-on process is stabilized, a number of more complex validation and debug activities can be performed.

2. *Basic hardware logic validation* follows power-on and ensures that the hardware design works correctly and exercises specific features of constituent IPs in the SoC design. This is typically done by subjecting the silicon to a suite of random and constrained-random special purpose tests.

3. *Compatibility validation* refers to the activities to ensure that the first silicon works with various versions of the system, application software, and peripherals. This validation accounts for various target use cases of the systems, the platforms in which the SoC is to be included, etc. Compatibility validation also includes validation of system with add-on hardwares, various operating systems and applications including games, and various network protocols and communication infrastructures. A key challenge in compatibility validation is the large number of potential combinations (of configurations of hardware, software, peripheral, and use cases) that need to be tested; typically it includes over a dozen operating systems, more than a hundred peripherals, and over 500 applications.

4. *Electrical validation* exercises electrical characteristics of the system to ensure adequate electrical margin under worst-case operating conditions. The electrical characteristics include input–output, power delivery, clock, etc. The validation is done with respect to various specification and platform requirements. As with compatibility validation, a key challenge here is the size of the parameter space:[1] for system quality and reliability targets, the validation must cover the entire spectrum of operating conditions for millions of parts.

5. *Speed-path validation* identifies frequency-limiting design paths in the first silicon due to the variation in the switching performance of the different transistors. Since circuit speed is constrained by the slowest path in the design, identifying such slow paths is of paramount importance to optimize design performance.

2.1.2 Post-silicon validation goals

The primary goal of post-silicon validation is to identify errors by exploiting post-silicon as a *giga-hertz order simulator*. The goal is *not* to completely diagnose or root-cause a bug, rather to narrow down from a post-silicon failure to an error scenario that can be effectively and efficiently investigated in the pre-silicon environment. Since silicon is involved in the validation process,

[1]Note, here the parameters are real valued variables *e.g.*, voltage, current, resistance.

the path from an observed failure (*e.g.*, system crash) to a resolution of the root cause for the failure is not straightforward. It includes following four steps [1] – i) test execution, ii) pre-sighting analysis, iii) sighting disposition, and iv) bug resolution.

1. *Test execution* involves setting up the test environment and platform, running the test, and performing sanity checks if a test fails. If the problem fails to resolve, then it is typically referred to as a pre-sighting.

2. *Pre-sighting analysis* aims to make the failure repeatable. This is a highly non-trivial task as most post-silicon failures occur under highly subtle coordinated execution of different IP blocks. Once a stable recipe for failure is discovered, the failure is referred to as sighting.

3. *Sighting disposition* involves developing a plan to track, address, and create turnarounds for the failure and calls for collaboration among architects, designers, and validators.

4. *Bug resolution* includes both finding a workaround for the failure to enable exploration of other potential bugs, and triaging and identifying root causes for the bug. Triaging and root-cause diagnosis of bugs are the two most complex challenges in post-silicon validation. Identifying the bug as a logic error, recreating the error in pre-silicon platform,[2] different observable failures for different tests for the same bug, and aggressive validation schedule make bug resolution a highly non-trivial exercise.

2.2 Established techniques for post-silicon SoC validation

2.2.1 Post-silicon observability enhancement

To facilitate post-silicon debugging and validation, modern SoC designs include a significant amount of on-chip instrumentation hardware, called design-

[2]The exact post-silicon scenario cannot be exercised in pre-silicon platform; one second of silicon execution would take several days to weeks on pre-silicon simulators. A scenario needs to be created that exhibits the same behavior as the original post-silicon failure but involves execution small enough to be replayable in pre-silicon platforms.

for-debug (DfD) [1]. In some cases, the DfD estimates to 20% or more silicon real estate. Two critical DfDs are i) *scan chain* [83] and ii) *trace buffer* [12]. In addition to these two architectures, there are also instrumentation to transport internal register values off-chip, quickly access large memory arrays, etc. These architectures can get highly complex. For example, in modern SoC designs, data transport mechanisms may repurpose some of the communication mechanisms already present in the system *e.g.*, universal serial bus (USB) port.

Scan-chain based observability enhancement: In [84, 85] the authors have proposed a technique to combine scan chains and trace buffers for enhanced in-field real-time debug data acquisition to maximize the observability of internal circuit states. In [86, 87], the authors propose a fine-grained architecture that uses various scan chains with different dumping periods. The authors also propose an efficient algorithm to select beneficial signals based on this architecture. In [88], the authors propose an efficient algorithm to select a profitable combination of trace and scan signals to maximize the overall signal restoration performance.

The primary drawback of scan chain-based observability enhancement methods is that the design execution needs to be stopped to offload the data from scan chains. This has two primary consequences – i) design execution data cannot be acquired in real-time and ii) halting design execution may prevent manifestation of subtle logic bugs that need continuous execution for thousands of clock cycles. Consequently, continuous data acquisition methods such as *trace buffer-based* techniques are favored for post-silicon validation.

Trace-buffer based observability enhancement: There are two distinct paradigms of trace buffer-based signal selection techniques – i) dynamic signal selection and ii) static signal selection.

In [89] the authors propose an enhanced algorithm for dynamic trace signal selection that can calculate state restorability values accurately by considering both local and global connections of the gate-level states. Also the proposed algorithms select trace signals dynamically to always guarantee high restoration ratio regardless of the input test patterns. Basu et al. [90] propose an efficient signal selection algorithm and associated trace controller design that would enable verification engineers to dynamically trace different set of signals for improved error detection. The authors propose a region-aware sig-

nal selection algorithm that selects useful signals during design time (using static analysis) based on the knowledge of functional regions and associated error zones and develop a low-overhead dynamic signal tracing hardware to enable designers to trace different set of signals during execution based on active (relevant) functional regions. In [91], the authors leverage pre-silicon information to enhance post-silicon trace signal selection in modern processors. In addition to that, the authors have developed a novel architecture for dynamic per-cycle selection of signals based on the present instruction. In pre-silicon phase, first, a set of controlling signals and their corresponding rules are extracted manually. Based on these rules, a set of data is extracted using an automatic formal method, which determines which signals should be traced at post-silicon. In [92, 93] the authors have proposed a trace signal selection technique based on error transmission, taking into account the topology of the design. The proposed signal selection methodology can be effectively applied to trace as well as a combination of trace and scan based observability techniques.

Static trace signal selection techniques can broadly be classified into *partial-restorability based selection* and *complete-restorability-based selection*. In [94], the authors have proposed a method based on partial-restorability using probabilistic analysis. They also enhance the technique of forward restorability and backward restorability [95, 96] to restore many more missing gate-level states.

Complete-restorability based signal selection techniques use a wide variety of methodologies including simulation and machine learning to select post-silicon trace signals. In [55], the authors have proposed a signal selection technique based on complete-restorability using a structural analysis of the gate-level netlist. This technique can guarantee better restoration compared to partial restorability and can provide both higher gate-level signal restoration ratio and significantly lower signal selection time. In [56], the authors show that a more accurate metric for state restoration capability of a set of signals can be obtained by actually simulating the restoration process on the circuit over a small number of cycles, and measuring the corresponding restoration ratio. They also propose a novel signal selection method guided by this metric. Komari and Vemuri [97] modeled the trace signal selection problem as a bi-partitioning problem, the set of flip-flops being tapped onto the trace buffer is one partition and remaining flip-flops form the other

partition. They use a simulated annealing heuristic to select trace signals. In [60], the authors combine structural analysis and simulation of gate-level netlist to propose a hybrid analysis-based trace signal selection technique. In [57, 58, 98], the authors propose an efficient signal selection technique using machine learning and take advantage of simulation-based signal selection while significantly reducing the simulation overhead. The approach uses bounded mock simulations to generate training vectors set for the machine learning technique followed by an elimination approach to identify the most profitable signals set. Later, the authors augmented this machine learning technique with integer linear programming (ILP) and propose an ILP-based algorithm for refining trace signal selection over multiple mock simulation runs of the gate-level netlist. Assertion coverage-aware trace signal selection was proposed in [61, 62]. In [99, 100, 101] the authors leverage information from RTL to select trace signals from gate-level netlist and to design on-chip debug hardware.

In our solution [63, 77, 78], we depart from prior art and apply hardware tracing at a higher-level of abstraction. First we apply hardware tracing for signal selection at the behavioral level (RTL) and then we raise the abstraction further and apply hardware tracing at the application level. Higher abstraction allows hardware tracing in a specific functional context, increasing signals' relevance in design understanding and debugging.

2.2.2 Post-silicon debug and diagnosis

IFRA [102, 103, 104] is primarily aimed to localize electrical bugs in multicore processors in a system setup. IFRA consists of a special design and analysis techniques required to bridge a major gap between system-level and circuit-level debug. Special hardware recorders, called footprint recording structures record semantic information about data and control flows of instructions passing through various design blocks of a processor. This information is recorded concurrently during normal operation of a processor in a post-silicon system validation setup. Upon detection of a problem, the recorded information is scanned out and analyzed for bug localization. Special program analysis techniques, together with the binary of the application executed during post-silicon validation, are used for the analysis.

Although IFRA does not require full system-level reproduction of bugs or system-level simulation but applying IFRA to a new processor microarchitectures can be challenging due to the manual effort required to implement special micro-architecture-dependent analysis techniques for bug localization. BLoG [105] automates the manual effort that is required to implement special micro-architecture-dependent analysis techniques of IFRA to a new processor micro-architecture.

BackSpace and BackSpaceL [106, 107] introduce a new paradigm for post-silicon debugging using formal analysis, augmented with some on-chip hardware support. These methods allow the chip to run at full speed, yet provide the ability to *backspace* hundreds, perhaps thousands, of cycles from a crash state or a programmable breakpoint, to derive an error trace that led to the crash, which can then be replayed in a simulator or waveform viewer to help understand the bug. Although the on-chip overhead was reasonable in BackSpaceL, Paula et al. leverage existing in-silicon debug logic *e.g.*, trace buffers, to propose TAB-BackSpace [108]. TAB-BackSpace has no additional hardware cost. Virtually, TAB-BackSpace achieves the effect of extending the trace buffer arbitrarily far back in time, *i.e.*, an effectively unlimited length trace buffer. In nuTAB-BackSpace [109], the authors exploit an observation that BackSpaceL needs to repeatedly trigger the bug via the exact same execution. In practice, non-determinism of post-silicon execution makes such exact repetition extremely unlikely. Instead, what typically arises is an intuitively *equivalent* trace that triggers the same bug, but is not cycle-by-cycle identical. In nuTAB-BackSpace, a user provides rewrite rules to specify which traces should be considered equivalent, and nuTAB-BackSpace uses these rules to make progress in trace computation even in the absence of exact trace matches. The authors prove that under reasonable assumptions about the rewrite rules, the abstract trace computed by nuTAB-BackSpace is concretizable – *i.e.*, it corresponds to a possible, real chip execution with low possibility of error.

BiPED [110] leverages the vast body of design knowledge that is available during pre-silicon verification to identify the exact time and location of post-silicon bugs. BiPED learns the correct design behavior of a design's communication patterns during pre-silicon verification. In post-silicon validation, this knowledge is used to detect errors by means of a reconfigurable hardware unit. On detection of an error, bug reproduction is not necessary: a diag-

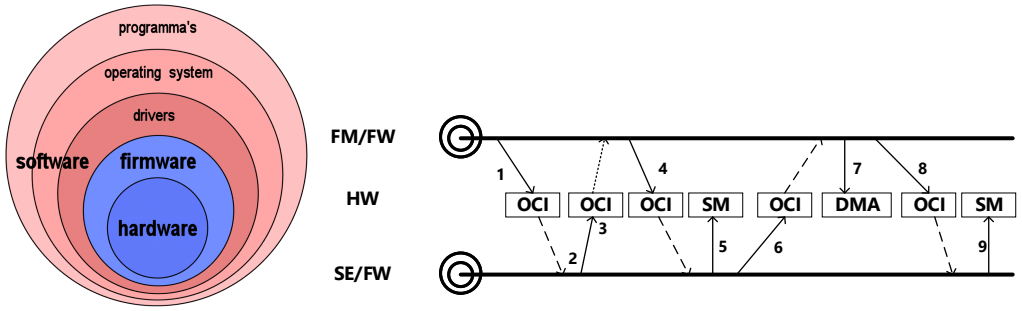
nosis software algorithm analyzes information stored in the hardware unit to provide a wide range of debugging information. Bug positioning system (BPS) [111] proposes a novel technique for automatic diagnosis of difficult electrical bugs during post-silicon validation. Lightweight BPS hardware logs a compact encoding of observed signal activity over multiple executions of the same test: some passing, some failing. Leveraging a novel post-analysis algorithm, BPS uses the logged activity to diagnose the bug, identifying the approximate manifestation time and critical design signals.

In [112, 113, 114, 115, 116], the authors present the Quick Error Detection (QED) technique for systematically creating families of post-silicon validation tests that quickly detect bugs inside processor cores and uncore components *e.g.*, cache controllers, memory controllers, and on-chip interconnection networks of multi-core SoCs. Such quick detection is essential because long error detection latency, the time elapsed between the occurrence of an error due to a bug and its manifestation as an observable failure, severely limits the effectiveness of traditional post-silicon validation approaches. QED can be implemented completely in software, without any hardware modification. Hence, it is readily applicable to existing designs. QED shortens error detection latencies and increases bug coverage. In Hybrid-QED (H-QED), Campbell et al. [117] leverage high-level synthesis (HLS) techniques to overcome post-silicon validation and debugging challenges for hardware accelerators. In [118], the authors present E-QED, a new approach that automatically localizes electrical bugs during post-silicon validation.

In contrast, our post-silicon debug and diagnosis solution [79] does not need any additional hardware and plugs into the current industrial post-silicon validation process with multiple orders of magnitude of benefits. Our solution uses trace signals obtained during post-silicon execution and employs the power of machine learning and feature engineering to automatically diagnose the root-cause of a post-silicon failure.

2.2.3 Comparative discussion of our post-silicon debug solution and QED

In contemporary SoCs, most of the system-level functionality are realized by guiding hardware via a firmware and vertically integrated components



(a) (b)

Figure 2.1: (a) shows vertical integration of SoC components. (b) shows a simplified secure boot flow [1, 2]. **FM/FW**: Functional module firmware. **SE/FW**: Security engine firmware. **HW**: Hwardware. **OCI**: On-chip interconnect. **SM**: Secure memory. **DMA**: Direct memory access. **1**: Request authentication. **2**: Fetch, lock, and authenticate. **3**: Status query. **4**: Status reply. **5**: Configure secure memory access. **6**: Image info. **7**: Data transfer. **8**: Transfer done. $\rightarrow$: MMIO message. $\dashrightarrow$: Polling messages. $\cdots\rightarrow$: Interrupt messages.

(c.f., Figure 2.1a), *e.g.*, boot image authentication, AES computation via AES firmware core. Consequently, failures such as deadlocks, hangs, crashes, blue screen of death etc. happen due to the bugs sensitized at the HW/SW or HW/FW communication interface. For example, in Figure 2.1b, a bug in any of the communication steps 1-9 would eventually cause a failure of boot image authentication that will symptomatize as *boot failure* or *blue screen of death*.

This example points to some key aspects of our post-silicon debug solution and QED. *Firstly*, focusing on hardware alone (like QED) *would not allow to debug failures* such as hangs, crashes etc. that occurs due to communications of vertically integrated components (c.f., Figure 2.1a). This is because neither hardware or firmware alone realizes the functionality nor it sensitizes the bug. It is the HW/FW communication that sensitizes and symptomatizes the bug. The application-level analysis (like our solution) models the different communications among HW/SW, HW/FW interfaces making it a potential candidate to target such failures. *Secondly*, the scope of QED is limited to only logical/electrical bugs that may occur in the behavioral model or in the fabricated silicon. On the other hand the scope of application-level analysis is much broader encompassing communication bugs at HW/FW, HW/SW, and SW/FW interfaces, concurrency bugs between HW/FW and between

IP blocks, and logical bugs. *Finally*, since QED accepts a behavioral model (such as a RTL of the design) for its computation, it cannot be applied to diagnose and localize bugs at the communication interfaces of HW/SW or HW/FW. This is primarily due to the fact that it is almost impossible to run any firmware of a reasonable size and complexity on a RTL design due to its extremely slow simulation speed (typically less than 100 cycles per second).

Scalability of a formal verification engine: Formal verification engine is the primary backbone of the symbolic QED [119] to localize the culprit set of hardware IPs. Symbolic QED uses a technique called *partial instantiation* (similar to *slicing*) to address the *scalability* issues. Using partial instantiation, QED creates a smaller set of *connected* hardware IPs that fits into a formal verification engine. While this method works when considering only hardware IPs, in the context of failures caused by communication between vertically integrated components, it has the following limitations.

1. A software verification engine like CBMC [120] cannot be used since the *hardware functions are not captured by program semantics*.

2. Formally verifying HW/FW components together using cycle-accurate models does not scale for multiple heterogeneous IPs.

3. Although recent works [121] have addressed reasoning about HW/FW concurrency, verification of concurrently executing HW/FW on multiple heterogeneous IPs is still an unaddressed problem. This is important in a heterogeneous environment where IPs potentially have different micro-controllers, different communication handling, and synchronization mechanisms.

4. Firmware often uses bit-wise operations, *e.g.*, shifting, masking, to access hardware architectural states stored in hardware registers. Formally checking such operations requires bit-precise reasoning, *e.g.*, bit-blasting. For multiple heterogeneous IPs such operations are highly expensive and do not scale.

In contrast, since application-level modeling and analysis work at higher abstraction and uses highly efficient and scalable machine learning techniques, scalability issues such as mentioned above do not affect it.

2.3 Established techniques for pre-silicon SoC verification

2.3.1 Assertion ranking

Vasudevan et al. [41, 43, 65] present GoldMine, a scalable automatic assertion generator for both sequential and combinational hardware designs in RTL. GoldMine integrates two solution spaces, statistical, dynamic techniques (data mining) and deterministic, static techniques (lightweight static analysis and formal verification), to provide a solution to the assertion generation problem. In [39, 67], the authors raised the abstraction and applied GoldMine to generate system-level assertions. Candidate assertions are generated in the form of frequent patterns from dynamic simulation trace data for both cycle-accurate and transaction-level designs [40]. In [66], the authors have proposed a word-level assertion generation method to have higher expressiveness and readability than their corresponding bit-level assertions using weakest-precondition computation [122]. In [123, 124], the authors present a coverage-guided mining approach for mining assertions from simulation traces using a combination of association-rule mining, greedy set covering, and formal verification. The authors use a coverage feedback to prevent both exhaustive rule generation of association-rule mining and assertions being over-constrained.

Recently, some efforts have been put to reduce the amount of required simulation trace data for high-quality assertion generation. In [42], the authors propose an assertion generation technique using dynamic dependency graphs. They extract relations between design signals and use significantly less number of simulation traces to generate more expressive properties. They do not use any expression template to establish relations between signals. The authors abstract from concrete use cases by inserting symbolic values by merging similar conditions in time.

In our solution [80, 81], we provide a systematic and efficient assertion ranking method to quantify the quality of an assertion (both automatically generated and manually written) based on the assertion's functional coverage of the design.

2.3.2 Pre-silicon debug and diagnosis

Debugging and bug localization have had a long history in software programs [125]. Machine learning and statistical techniques were applied in [126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137] for debugging software programs.

Veneris et al. [68, 69, 70] present an automated method for RTL failure triage. Failure triage is the task of analyzing large sets of failures followed by grouping those failures together that are likely to be caused by the same design error. The proposed framework instruments techniques from the machine-learning domain combined with the root-cause analysis power of modern SAT-based debugging tools in order to exploit information from error traces and group the corresponding failures using clustering algorithms. In [74, 138], the authors propose an automated failure triage framework for RTL debugging that unifies three critical aspects of the problem: the approximation of the general location of root-cause(s) in the design under verification, the binning of all related failures generated by regression runs, and the distribution of these binned failures to the proper engineer(s) for detailed analysis. The proposed triage engine entails two novel methodologies – i) a classification framework that mines information from SAT-based debugging and simulation to probabilistically reason about the relation of root-causes with their respective failing verification traces and ii) a formulation of failure binning as exemplar-based clustering for grouping and distributing failing traces to the proper engineering team(s).

For RTL debugging, a simulation-based technique was proposed in [139] by capturing all possible faulty behaviors that can be generated from specific sets of design nodes. In [140], the authors propose a BDD-based multiple design error diagnosis and correction technique via implicit enumeration of erroneous lines. A novel formulation of the debugging problem using MaxSAT to improve performance and applicability of automated debuggers was proposed in [141]. The technique identifies the errors in the design and indicates when the bug is excited in the error trace. In [73, 75], the authors propose a directed SAT-based debugging algorithm which prioritizes examining design locations that are more likely to be suspects. This prioritization is learned from historical debug data to predict the suspect locations. Berryhill et al. [142, 143, 144] propose a novel approach that considers only

a portion of the RTL design locations but still finds the complete solution set to the problem. The presented approach proceeds through a series of iterations, each considering a strategically-chosen subset of the design locations (a suspect set) to determine if they are root causes. The results of each iteration inform the choice of suspect set for the next iteration. Becker et al. [71] propose FudgeFactor, a RTL debugging technique that provides semantically-meaningful RTL source code corrections. This method starts with a buggy design, at least one failing and several correct test vectors, and a list of possible suspect bug locations. Using this list and a library of rules empirically characterizing typical source code mistakes, the authors instrument the buggy design to allow each potential error location to either be left unchanged, or replaced with a set of possible corrections. FudgeFactor then combines the instrumented design with the test vectors and solves a 2QBF-SAT problem to find the minimum number of source-level changes from the original code which correct the bug. In [72], the authors introduce a performance-driven debugging methodology for pinpointing the root-cause of memory-locked errors. The technique models only a sliding time window and a final time window explicitly at any one time, while interstitial time-frames are linked with a lightweight memory model. In [145], the authors propose an iterative algorithm in which a high coverage rule is discovered using association rule mining to differentiate state vectors in a failure cycle from the passing cycle of a simulation run. This method does not localize to suspicious code zones in the design. In symbolic QED method [146, 147, 148], the authors employ bounded model checking, software transformations, including redundant execution and control flow checking of the applied quick error detection tests [112, 113]. Symbolic QED combines these error-detecting QED transformations with bounded model checking-based formal analysis to generate minimal-length bug activation traces that detect and localize any logic bugs in the pre-silicon RTL design.

In contrast, our RTL debugging solution [82] is based on identifying statistically relevant common symptoms across failing simulation traces through mining, and mapping these back to the corresponding execution paths in the RTL source code. Our solution does not need historical debug data, possible suspect bug locations, or a library of rules. Our localized code zones are small, focused, functionally coherent, and executable.

2.4 GoldMine for automatic assertion generation

GoldMine [41, 65] incorporates two diverse solution spaces, statistical, dynamic techniques (data mining and machine learning) and deterministic, static techniques (lightweight static analysis and formal verification), to provide a scalable and automated solution to the assertion generation problem. Static analysis of designs (including formal verification) can make excellent generalization and capture domain/design specific information, but often suffers from scalability and computational complexity issues. Data mining and machine learning, on the other hand, are extremely computationally efficient, but depend on domain knowledge guidance for deriving relevant knowledge from a system. Together, these two technologies offset each other's disadvantages. The data mining when guided by the design information gathered via static analysis of the design, gives rise to useful and succinct design knowledge *i.e.*, assertions.

GoldMine can generate both propositional and temporal assertions. The generated assertions are of the form $\mathbf{P}$: $\mathcal{G}(A \rightarrow C)$ or $\mathbf{P}$: $\mathcal{G}(A \Rightarrow C)$ where A is the antecedent and C is the consequent of the assertion P. A is a conjunction of a propositions defined in terms of input and/or register variables and C is proposition defined in terms of a given register and/or output variable. Each proposition in A or in C is a signal-value pair. The variable in C is called a target variable. GoldMine generates assertions of bounded length. Hence, GoldMine cannot generate unbounded liveness properties. We use linear temporal logic (LTL) [149] notation to express GoldMine assertions. GoldMine generates assertions both at the module level [41] and at the system level [67]. GoldMine can also generate assertions for bit-level target variable [41] and word-level target variable [66].

Figure 2.2 shows the architecture of GoldMine which is composed of *data generator*, *static analyzer*, *assertion miner*, *formal verifier*, and *assertion evaluator* components.

2.4.1 Static analyzer

The static analyzer analyzes the design source code and extracts design-specific information and passes it to the other GoldMine components such as assertion miner, formal verifier etc. The static analyzer determines basic

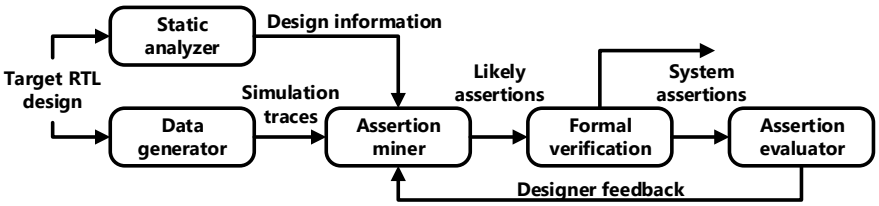


Figure 2.2: GoldMine architecture.

design information such as discerning the top module in the design hierarchy, identifying clock and reset signals, and selecting a set of target variables.

Static analyzer also selects a set of feature variables per target variable by using the bounded cone of influence. The bounded cone of influence uses a design's dependency graph to transitively compute the variables that can affect the target variable within a bounded number of temporal frames.

2.4.2 Data generator

The data generator generates data for the assertion miner algorithms. For a given RTL design, the data is obtained via dynamic simulation. The design is simulated for a fixed number of cycles (10,000 cycles) using random input stimuli. Regression and directed test, if available, can also be used to generate the data.

The data generator parses entire simulation trace data and summarizes it by retaining only those data that coincides with the clock edge. Next, the data generator unrolls the data for the specified number of temporal frames and discards any duplicate frames. The assertion miner heavily relies on this data preprocessing to simply mining task.

2.4.3 Assertion miner

The assertion miner uses the design information from the static analyzer to constrain mining on the signals that are in the bounded cone of influence of a target variable. This limits the search space of the mining algorithm from all possible design signals to the relevant signals in the design for a target variable.

The assertion miner of GoldMine uses various mining algorithms such as decision-tree based miner and best-gain decision forest miner [41], coverage closure-based miner and counterexample-guided miner [123, 124, 150], and PRISM [151]. The data miner searches for causal relationships between feature variables and the target variable in the simulation data. If the data miner finds a relationship with 100% confidence, it generates an assertion.

2.4.4 Formal verifier

It is often very difficult, if not impossible, to simulate every possible design functionality. Therefore, the simulation trace data is *incomplete* and captures only a subset of design functionality. Consequently, it cannot be guaranteed that the generated assertions are true system invariants. Therefore, the formal verifier uses Cadence Incisive Formal Verifier (IFV) [152] to verify the generated assertions. The formal verifier reports assertions that pass formal verification as true system invariants. If an assertion fails formal verification, then the assertion miner can use the assertion's counterexample to refine it [150].

CHAPTER 3

EMPHASIZING FUNCTIONAL RELEVANCE OVER STATE RESTORATION IN POST-SILICON SIGNAL TRACING

3.1 Introduction

In post-silicon validation, limited observability is a key obstacle that seriously hinders observation of various internal design signals during execution. Hence important and functionally relevant internal design signals need to be instrumented at an observation point (*e.g.*, trace buffers) before first silicon is available. State-of-the-art methods fail to select signals that are functionally relevant and most beneficial for design understanding and debugging.

In this chapter, we endeavor to increase the functional relevance of selected signals by departing from the SRR optimizing strategy of prior art. Instead, our approach was to let the design structure indicate importance of signals. Our algorithm is based on the Google's PageRank algorithm [64, 153] (c.f., Section 3.2.1), as applied to the circuit behavioral design (RTL) and circuit netlist (c.f., Problem PR1 of Figure 1.10). At the gate level, we applied PageRank to the structural netlist. For RTL, we applied it to the variable dependency graph (c.f., Definition 1). The reason for applying PageRank in these two modes is to study the relative benefits, if any, of signal selection in an RTL data structure over gate level: applying the same algorithm at both levels would prevent the variability in analysis due to algorithmic differences. The algorithm gives us a rank ordering among important signals for tracing. We compared the signals selected by our method with the signals from SRR-based techniques. We used pre-silicon simulation coverage metrics to establish functional relevance of selected signals. We performed an initial set of signal selection experiments on a USB design [154], which has substantially more complex behavior than ISCAS89 benchmarks used in the literature. Our results showed that compared to SRR-based methods, our method selected signals with high functional relevance. Further, we plot-

ted SRR against the behavioral coverage achieved by the signals selected by SRR optimizing methods *e.g.*, [55] and our method. We found that high SRR values do not correlate to high behavioral coverage.

Scalability is an important concern in automatic post-silicon trace signal selection as its methods work on a fine-grained netlist level. A modern SoC contains hundreds of different IP blocks [3, 4, 155] with millions of logic elements such as flip-flops. SRR-based methods [55, 56, 58, 60] update the rate of restorability of each flip-flop in the design in each iteration based on the currently selected trace signals. For a large-scale design such as a modern SoC, this iterative update is computationally expensive and has a chance to run out of time and/or memory. This considerably limits the scalability of the state-of-the-art algorithms. On the other hand, our PageRank based algorithm as applied to netlist (PRoN) avoids the restorability computation altogether, relying on design structure and connectedness as the guideline for signal selection. This is a cheaper operation.

In our hardware tracing solution, we also focus on scalability, and demonstrate experiments at an industrial scale. We show results on the publicly available multi-core SoC design, OpenSPARC T2 [156, 157]. OpenSPARC T2 contains several heterogeneous IPs and reflects many of the complex features of an industrial SoC design. We selected several large and complex modules from OpenSPARC T2 that contain *up to 14,000 flip-flops* and *up to 74,000 logic elements* for our experiment. The scale and complexity of these design modules are several orders of magnitude greater than those of the traditional ISCAS89 benchmarks used in signal selection literature. This added complexity helps to illustrate the divergence between gate-level state restorability and functional behavior.

Our experiments on OpenSPARC T2 design modules showed that state-of-the-art signal selection techniques [55, 56, 58, 60] could not finish signal selection for designs consisting of *no more than 2,800* flip-flops due to timeout and large peak memory usage (*up to 30 GB*). Our PRoN algorithm was able to select trace signals for designs containing *approximately 14,000* flip-flops *within 13 seconds* with a peak memory usage of *up to 1.5 GB*. Our results showed that PRoN has much better scalability than other state-of-the-art signal selection algorithms for industrial-scale designs.

While the original PageRank was sufficiently accurate for our ISCAS89 and USB experiments, application of this algorithm to the large-scale OpenSPARC

T2 introduced problems that needed to be addressed at scale. Complex interconnections such as feedback and feed-forward loop structures among flip-flops and deep hierarchical signal connections from instantiated modules to the top module exposed issues in the original PageRank algorithm. The density and complexity of connections in the real design caused PageRank to infer that the outputs were the most important. For our purpose, tracing an output signal does not add any value, since it is observable anyway. In this chapter, therefore, we further modify PageRank to correctly rank internal signals for large, complex designs. We will refer to this modified PageRank as PageRank on Netlist (PRoN) hereafter.

In this chapter, we also provide a more comprehensive experimental study to compare the quality of the selected trace signals in terms of behavioral coverage by using total restorability-based [55, 63], hybrid-analysis-based [60], ILP-based [57, 58] and simulation-based [56] signal selection algorithms.

Our experimental results, when we applied the algorithms to OpenSPARC T2, were in conformance with the results presented with USB design. We compared our PRoN method with the only two SRR-based techniques that could finish for at least some of the OpenSPARC T2 design modules. The *behavioral coverage of the signals selected by our PRoN method consistently outperformed* (*up to 50.94% more*) the signals selected by SRR optimizing methods [55, 60]. Further, we showed that signals selected by PRoN executed *up to 4.59%* more design paths than did signals selected by SRR-optimizing methods on large-scale designs. PRoN achieves higher path coverage for the signals than selected by the SRR-optimizing methods due to enhanced PageRank metric as it prefers flip-flops that are highly connected and part of many design paths.

For completeness, we determined the extent of restorability achieved by all the algorithms, including PRoN. Interestingly, the signals selected by PRoN although not optimized for SRR, often achieve *up to 7.3*× (*on an average 3.15*×) higher restorability on large-scale designs compared to signals selected by SRR-optimizing methods.

Our contributions are as follows.

- We show through empirical evidence and analysis that SRR is severely limiting as a general metric for post-silicon signal selection. We argue that a different metric is necessary that directly correlates with the

extent of coverage of the execution flow of the design.

- We propose a new scalable signal selection algorithm that performs significantly better than algorithms designed to maximize SRR in achieving functional coverage. Our algorithm is adapted from Google's [64, 153] PageRank algorithm. It ranks some signals as more important than the others based on the connectivity in the structural netlist or the RTL variable dependency graph. Higher ranked signals are better candidates for tracing. It also avoids inclusion of entire arrays, and selects relevant signals instead. Finally, it typically selects the signals and their operating conditions together due to their high co-occurrence and consequent similar ranking.

- We demonstrate the scalability and viability of our PRoN signal selection algorithm on the OpenSPARC T2 SoC design modules containing *up to 14,000* flip-flops and *up to 74,000* logic elements. To the best of our knowledge, this is the largest-scale application of netlist- level signal selection approaches demonstrated in the literature.

- Finally, we provide a comprehensive comparison of our PRoN technique with all the signal selection based techniques (and tools) available in the public domain in terms of behavioral coverage. This provides conclusive empirical evidence for the functional superiority of the signals selected by our method as compared to the state-of-the-art SRR-based methods.

3.2 Preliminaries

3.2.1 PageRank algorithm

Google PageRank algorithm [64, 153] ranks a web page as important if it is hyperlinked from many important web pages. This ensures that not all hyperlinks have equal weights. PageRank computes an importance score for each web page based on its incoming hyperlinks. Let p denote a web page. Let $B(p)$ denote the set of pages that have an outgoing link to p, and let $F(p)$ denote the set of pages to which p has outgoing links. Let ϵ be a constant

between 0 and 1.0 and let n be the number of web pages. The PageRank $PR(p)$ of p is defined as:

$$PR(p) = (1 - \epsilon) \sum_{p_i \in B(p)} \frac{PR(p_i)}{|F(p_i)|} + \frac{\epsilon}{n} \tag{3.1}$$

The first term in the Equation 3.1 represents the probability that a random surfer will navigate to a web page. If the surfer is caught in a cycle of web pages, then it is unlikely that he or she will continue in the cycle forever. The second term accounts for the surfer's eventual departure from the cycle and navigation to a random web page.

3.2.2 Trace buffer parameters

Hardware tracing is one among many different DfD architectures (c.f., Section 2.2.1) that are used to address the observability limitation during post-silicon debugging. A trace buffer has two parameters, i) *width i.e.* the number of bits of signals that can be traced simultaneously, ii) *depth i.e.* the number of cycles for which signals values can be traced.

3.2.3 Variable dependency graph

We define a variable dependency graph for an RTL design based on the semantics of the Verilog hardware description language [8]. An *expression* is a function defined over variables and operators. A *left reference* refers to a variable that appears on the left side of a Verilog assignment expression. A *right reference* refers to all variables that are not left references. Let v_i and v_j be two Verilog variables. A variable v_i depends on v_j if there exists a Verilog assignment to v_i that will execute only if a right reference to v_j is evaluated.

Definition 1 *A **variable dependency graph** (VDG) is defined as the weighted directed graph* $\mathbf{G} = (\mathbf{V}, \mathbf{E}, \mathbf{W})$ *with vertices* $\mathbf{V}$*, directed edges* $\mathbf{E}$*, and edge weights* $\mathbf{W}$*. Let each vertex* $v_i \in \mathbf{V}$ *denotes a Verilog variable. Let each directed edge* $(v_i, v_j) \in \mathbf{E}$ *denotes a dependence between* v_i *and* v_j*. If* $e_{ij} = (v_i, v_j) \in \mathbf{E}$*, then* v_j *depends on* v_i*. Let* $w_{ij} \in \mathbf{W}$ *be the weight of*

```
module arb2(input clk, rst, req1, req2,
    output gnt1, gnt2);
reg gnt_, gnt1, gnt2;
always @(posedge clk or posedge rst)
 if(rst)
   gnt_ <= 0;
 else
   gnt_ <= gnt1;
always @(*)
 if(gnt_)
   begin
     gnt1 = req1 & ~req2;
     gnt2 = req2;
   end
 else
   begin
     gnt1 = req1;
     gnt2 = req2 & ~req1;
   end
endmodule
```

Figure 3.1: Verilog code of a two-port arbiter design.

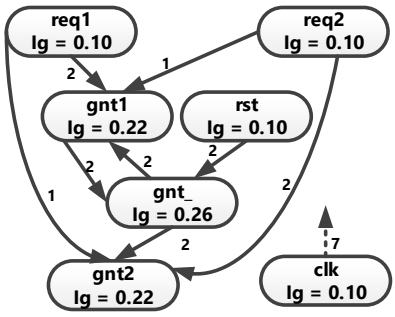
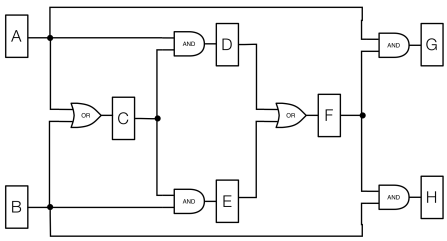


Figure 3.2: Variable dependency graphs (VDG) for the two-port arbiter of Figure 3.1. Here $\mathbf{V} = \{req1, req2, gnt\_, gnt1, gnt2, clk, rst\}$. I_g is the importance score of a node.



(a)

Signal	Cyc1	Cyc2	Cyc3	Cyc4	Cyc5
A	0	0	0	0	1
B	1	0	1	0	X
C	1	1	0	1	0
D	X	0	0	0	0
E	X	1	0	0	0
F	X	X	1	0	0
G	X	0	0	0	0
H	X	X	0	1	0

(b)

Figure 3.3: (a): Example circuit [55]. (b): State restoration for circuit shown in Figure 3.3a applying algorithm of [55].

the edge $e_{ij} \in \mathbf{E}$ *that summarizes the control and data dependencies between* (v_i, v_j) *in a Verilog design.*

We use the two-port arbiter of Figure 3.1 as our running example. Figure 3.2 shows the variable dependency graph **G** of the two-port arbiter of Figure 3.1. The edge weight $w_{(gnt\_,gnt1)} = 2$ summarizes two control dependencies between $gnt\_$ and $gnt1$ at line 12 and line 17 of Figure 3.1.

3.2.4 Signal reconstruction and SRR calculation

State Restoration Ratio (SRR) [55] *is defined as the sum total of the number of signals traced and the number of signals restored expressed as a fraction of the number of signals traced, i.e.* SRR = (total number of signals traced + total number of signals restored) / (total number of signals traced).

We calculate SRR for the simple circuit shown in Figure 3.3a. Let us assume that the trace buffer can record values of two signals. The restored values of the other signal states that use the method of [55] are shown in Figure 3.3b. The signals that are chosen via total restorability computations are *A* and *C*. The selected signals are shown in grey. Since *ten* signal values are traced and *22* values are restored, the SRR with this selection is *3.2*.

3.2.5 Simulation based coverage metrics

In this section, we define several simulation-based coverage metrics that are used in pre-silicon simulation environment to quantify the design behavior covered by a testbench.

Definition 2 *The* **line coverage** *is defined as the fraction of total design statements (like* blocking, non-blocking*, and* assign *statements) that are executed in a design simulation.*

Definition 3 *The* **branch coverage** *is defined as the fraction of total branches (like* If-else*,* Case statements*) that are executed in a design simulation.*

Definition 4 *The* **condition coverage** *is defined as the fraction of conditions of all branches that are executed in a design simulation.*

Definition 5 *The* **path coverage** *is defined as the fraction of total design paths that are executed in a design simulation.*

Definition 6 *The* **toggle coverage** *is defined as the fraction of total bits of a wire/register that change from a value of zero (1'b0) to one (1'b1) and back from one (1'b1) to zero (1'b0) in a design simulation. A bit is said to be fully covered when it toggles back and forth at least once.*

3.3 Inadequacy of SRR as a metric

3.3.1 A motivating example

In this example we provide a comparison of selected signals corresponding to interesting high-level behavior between SRR based signal selection methods and our proposed PRoN method. We show that using LC3B [158], a 16-bit academic processor in which we attempt to reconstruct the micro-architectural state.

We applied the SRR based signal selection technique SigSeT_1 [55] that is designed to maximize the SRR [55]. It selects the complete ISDU FSM (functional state machine of LC3B) state registers, some bits of the program counter (PC), and some bits of the instruction register (IR) at the top of the list. With this set of signals, we can recreate a few control states, but not the rest of the processor state. Without the complete PC and IR, it is not possible to determine which instruction will be processed and fetched from memory next.

As a point of contrast with the above results, consider the performance of our PRoN algorithm that does not seek to maximize SRR (c.f., Section 3.4)

on the same example. PageRank selects all of the ISDU FSM state registers, all 16 bits of PC and IR as complete words, and NZP branching registers. This is sufficient to check the sequence of states in the design, the opcode and operands fetched, all transitions in the control state machine, and branching behavior. PRoN ranks all of the control signals with high priority, while ranking eight 16-bit data registers lower. This helps in reconstructing the micro-architectural state of LC3B.

The above example suggests a key problem with the utility of SRR as a metric: it treats all gate-level design states as "equals". Reconstructing any specific design state is not considered more valuable than reconstructing any other state. However, practical debugging experience suggests that some signals are inherently more valuable for validation and debug than others. Also, some signals can provide useful state information only in the presence of some other signals as well. For example, reconstructing only the lower-order bit of a program counter (PC) provides little information on program behavior or execution flow, while reconstructing all bits of the PC can provide significant insight. Consequently, signals selected to optimize SRR do not necessarily facilitate debugging.

3.3.2 Deconstructing SRR inadequacies

In particular, SRR is not useful for signal selection for designs with the following features.

Large arrays: In such designs, individual array elements are typically less valuable for debugging than are control signals that affect reads and writes to the arrays. Methods that optimize SRR, on the other hand, would tend to reconstruct individual array values.

On-chip instrumentation: Modern IC designs include a significant amount of on-chip hardware instrumentation that do not contribute to functionality, including Design-for-Test (DFT) features, instrumentation for security, and, indeed, hardware to enable post-silicon debug and control. Since SRR is agnostic to design intent, selection based on SRR typically includes a sampling of signals for different functionalities as well as different instrumentation features. The result is that the traced signals are inadequate for functional debugging while also not providing sufficient design visibility to enable vali-

dation of instrumentation.

Complex protocols: Most multi-core systems and SoC designs include design blocks (referred to as "IP") that coordinate through complex protocols. One of the critical applications of hardware trace is to validate these protocol implementations during post-silicon debugging. This implies that the traced signals include the messages communicated across the IPs during system execution. However, SRR does not account for the relative importance of these signals. Indeed, algorithms that optimize SRR would tend to favor signals in larger IPs with more design states while missing smaller IPs; thus, routers in communication fabrics through which protocol messages are communicated would typically be ignored.

Algorithm 1 Pseudo-code of PRoN algorithm

1: **procedure PRoN($\mathcal{G}$, $\mathcal{G}'$, error, ϵ)**
2: $\mathcal{G} = (\mathcal{V}, \mathcal{E}), \mathcal{G}' = (\mathcal{V}, \mathcal{E}')$ {if $(v_i, v_j) \in \mathcal{E}$, then $(v_j, v_i) \in \mathcal{E}'$}
3: **error**: error bound for rank matrix convergence
4: ϵ: damping factor
5: $prank_1 \leftarrow$ PageRank($\mathcal{G}$, error, ϵ)
6: $prank_2 \leftarrow$ PageRank($\mathcal{G}'$, error, ϵ)
7: **for** v in $\mathcal{G}$ **do**
8: $\quad prank_{hm}(v) \leftarrow$ HM($prank_1(v)$, $prank_2(v)$) {HM: Harmonic Mean}
9: **end for**

3.4 PageRank-based trace signal selection algorithm

3.4.1 PageRank for netlist

We apply the PageRank algorithm [64, 153] to the circuit netlist. Algorithm 1 details the PRoN algorithm. In this section we apply it on the example circuit shown in Figure 3.3a.

Network construction: We parse the synthesized netlist of an RTL design to construct a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ representing the connectivity between different logic elements, where every $v \in \mathcal{V}$ represents a logic element and every directed edge $(v_i, v_j) \in \mathcal{E}$ represents a connection between the logic elements v_i and v_j. Figure 3.4 shows the directed graph for the example circuit in Figure 3.3a.

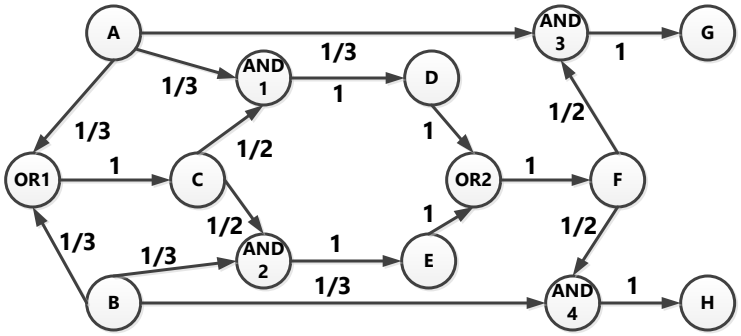


Figure 3.4: Circuit network $\mathcal{G}$ for the example circuit of Figure 3.3a. Each node in $\mathcal{G}$ is a logic element and each edge in $\mathcal{G}$ follows the connectivity in the circuit. Each edge of $\mathcal{G}$ is annotated with importance contributions from each node.

PageRank value calculation: After constructing the directed graph for the circuit, we apply PageRank algorithm to compute the importance of each node. The directed graph in Figure 3.4 has 14 logic elements (eight sequential elements and six logic gates). Each node transfers its importance equally to the nodes to which it links. For example, node A has three out-links, so it will transfer 1/3 of its importance to each of the nodes OR1, AND1, and AND3. In general, if a node has n out-links, it will pass on $1/n$ of its importance to each of the nodes to which it is linked. Following this importance transition rule, we annotate every edge of the graph in Figure 3.4 with the corresponding importance value.

Initially we assume an equal rank for each of the nodes *i.e.* if there are n nodes in the network, every node will have a rank of $1/n$. In Figure 3.4 each node has a rank of 1/14. As each incoming link increases the rank of a node, we update the rank of each node by adding the importance of the incoming links. We continue this until the rank of all of the nodes stabilizes. We use a standard error tolerance value in the PageRank algorithm, which is 1e-6, to check for convergence in the power iteration process. If the PageRank values across two iterations is within this error tolerance, the rank of nodes is assumed to have stabilized and is returned. In the example network, nodes G and H do not have any outgoing links, and PageRank refers to them as *dangling nodes*.

Dangling nodes would cause the final rank of each node to converge to 0, and the importance of these nodes cannot be propagated further. Since

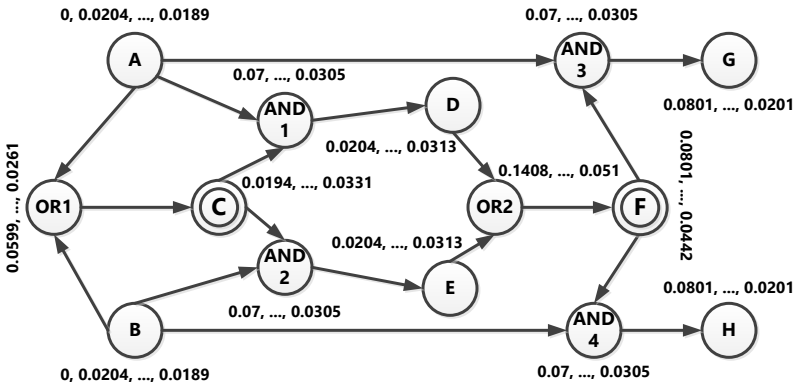


Figure 3.5: Nodes of $\mathcal{G}$ of Figure 3.4 annotated with importance values in successive iterations and the final importance value as calculated by PRoN. PRoN selects flip-flop F and C (shown in double circle) to trace as trace buffer width is 2.

dangling nodes and disconnected components are quite common in the internet as well as in common circuits, a positive constant between 0 and 1.0 (typically 0.15) is introduced, which is the damping factor ϵ [153]. We add a virtual directed edge from G and H to every other node in the network and assign ϵ to every outgoing edge from G and H.

After adjustment of the dangling nodes, we recalculate the rank of each of the nodes in the graph until the PageRank value stabilizes. For our example, the initial value, intermediate value, and final value of the PageRank of each node is shown in Figure 3.5.

Let $0 < \epsilon < 1$ be a constant source of importance. Let $\mathbf{r}^k$ denote $\mathbf{r}$ in the k-th iteration of the rank computation. Let $\mathbf{A}$ be the adjacency matrix of size $n \times n$ where each $A(a_i, a_j)$ is the ratio between the number of right references to variable i in all assignments to variable j to the number of right references to variable i in all assignments. Let $\mathbf{r}_i^0 = \frac{1}{n}$. We compute the importance score of each of the variables according to Equation 3.2.

$$\mathbf{r}^{k+1} = (1 - \epsilon)\mathbf{A}\mathbf{r}^k + \frac{\epsilon}{n} \tag{3.2}$$

3.4.2 PageRank for RTL

To compute the importance score of each of the variable in an RTL design, we apply PageRank algorithm on the variable dependency graph (c.f., Definition 1) of the RTL design.

We represent a variable dependency graph $G_g = (V_g, E_g, W_g)$ by using a $n \times n$ adjacency matrix $\mathbf{A}$ with rows and columns corresponding to the design variables. Let a_{ij} denote the number of right references to variable i in all assignments to variable j, and a_i denote the number of right references to variable i in all assignments. Let $\mathbf{A}_{ij} = a_{ij}/a_i$ if $a_i > 0$ and let $\mathbf{A}_{ij} = 1/n$ otherwise. Intuitively, we see that $\mathbf{A}_{ij}$ is equal to the fraction of right references to variable i that exist in all assignments to variable j. If no references to variable i exist in the RTL, then we assume that a right reference to variable i exists in an assignment to each other variable. Hence, $\mathbf{A}_{ij} = 1/n$ when $a_i = 0$.

The importance computation iteratively computes the importance score of each variable in the design until the score is stabilized. We have found through experimentation that when $\epsilon = 0.5$, the global importance score distribution of the variable agrees with the designer intuition. The equation for computing the rank of variables in the variable dependency graph is the same as Equation 3.2.

Figure 3.2 shows the variable dependency graph of the arbiter of Figure 3.1. Each node in the graph is labeled with its respective variable and the PageRank score. Edge weights denote the number of dependencies between the variables. For example, since `gnt1` depends on `req1` in both lines 12 and 17 of the Verilog, the weight of the edge (`req1, gnt1`) is equal to 2. Any edge without a specified weight has a a weight equal to 1. From the final ranks after convergence, we find that gnt_, which is the arbitration signal is ranked highest, after which gnt1 and gnt2, the two signals receiving the grant are ranked. Other signals are equally (less) important. We select the top 20% of the signals rank sorted by the PageRank algorithm.

3.4.3 Enhancing the ranking metric of selected signals

PageRank algorithm implicitly adjusts for the in-degree of each node. When the same principle applied on a circuit netlist graph with loop structures

among flops, PageRank algorithm tends to select output signals from the IP modules as high-ranking signals. Since output signals are connected to many internal signals, they inherit their importance values from these signals. This gives the PageRank algorithm a *false* sense of importance. For the signal selection application, selecting output signals is not useful, since these are already observable. Our objective in signal selection is to select important internal signals for observation.

To resolve that concern, we enhance the signal-ranking metric of the traditional PageRank algorithm. We calculate a *reverse PageRank* for each of the nodes in the circuit graph. We create a graph $\mathcal{G}' = (\mathcal{V}, \mathcal{E}')$ for the original circuit graph $\mathcal{G}$. For each directed edge $e \in \mathcal{E}$ connecting a pair of nodes (v_i, v_j) in the original circuit netlist graph $\mathcal{G}$, we create a directed edge $e' \in \mathcal{E}'$ connecting the same pair of nodes (v_j, v_i) in the graph $\mathcal{G}'$. Then we calculate a PageRank score for each of the graph nodes in $\mathcal{G}'$. We use $prank_1(v)$ to denote the PageRank of a node in $\mathcal{G}$, and $prank_2(v)$ to denote the PageRank of a node in $\mathcal{G}'$. Intuitively, we see that PageRank algorithm will assign high importance values to the nodes in $\mathcal{G}'$ that have high in-degrees from other important nodes. The in-degree of a node in $\mathcal{G}'$ maps to the out-degree of the same node in the $\mathcal{G}$. To combine $prank_1(v)$ and $prank_2(v)$, we calculate their harmonic mean (HM) following the idea of the *importance* metric in [126]. We use $prank_{hm}$ to denote this metric.

$$prank_{hm}(v) = \frac{1}{\frac{1}{prank_1(v)} + \frac{1}{prank_2(v)}}$$

By virtue of HM, $prank_{hm}$ will assign high ranks to the flip-flops that have high values for both $prank_1$ and $prank_2$. Intuitively, this means that $prank_{hm}$ selects flip-flop nodes that are connected to many other important flip-flop nodes via incoming edges and can propagate their values to many other flip-flops via outgoing edges. This indeed resolves our original concern about PageRank algorithm's selection of output nodes for tracing. For the output nodes of a circuit netlist, the in-degree is very high but the out-degree is very low. On the other hand, for the input nodes of a circuit netlist, the out-degree is very high but the in-degree is zero. Therefore $prank_{hm}$ will not select either outputs or inputs for tracing. Instead, it prefers important internal nodes of the design. In our running example, PRoN selects flip-flop F and C (c.f., Figure 3.5) to trace which are neither outputs nor inputs

of the design rather they are internal to the design. Our SRR results for OpenSPARC T2 IP modules given in Section 3.6.2 supports that conclusion.

3.4.4 Functionally relevant signal selection by PRoN

The PRoN algorithm analyzes the structure of the circuit netlist and selects a signal that is important in the design, based on which other important signals that signal is connected to. If a variable is well-connected to other connected variables, it is highly likely that variable forms an important part of the design function. In a well-designed hardware, design structure should be closely related to functionality, for optimal performance of the design implementation. We believe variable importance is a metric that transitions quite faithfully between the structure and the function of a design, thereby capturing how a design structure, does in fact, correspond to the functionality. Our results in Section 3.6.3 support this intuition.

3.5 Experimental setup

Design testbed: We primarily use the publicly available USB 2.0 [154], ISCAS89 benchmarks, and multi-core OpenSPARC T2 SoC [156, 157] to demonstrate our results. Comparing the testbeds, we observe that the ISCAS89 benchmarks have no more than 1700 flops, and the USB despite being more complex, synthesizes to around 1800 flops. In contrast, the OpenSPARC T2 is a large, industry-scale design with high complexity. We describe the experimental setup with respect to the OpenSPARC T2 in the rest of this section.

OpenSPARC T2[1] is a multi-core SoC containing several heterogeneous IPs and many of the complex design features of an industrial SoC design. Figure 3.6 shows an IP-level block diagram of OpenSPARC T2. For these experiments, we used several larger and complex IP modules of the OpenSPARC T2 design. Details of the ISCAS89 benchmarks, USB 2.0, and several large and complex modules of OpenSPARC T2 in terms of the total number of flip-flops and the logic elements are shown in Figure 3.7. Table 3.1 details

[1]OpenSPARC T2 source: `http://www.oracle.com/technetwork/systems/opensparc/opensparc-t2-page-1446157.html`

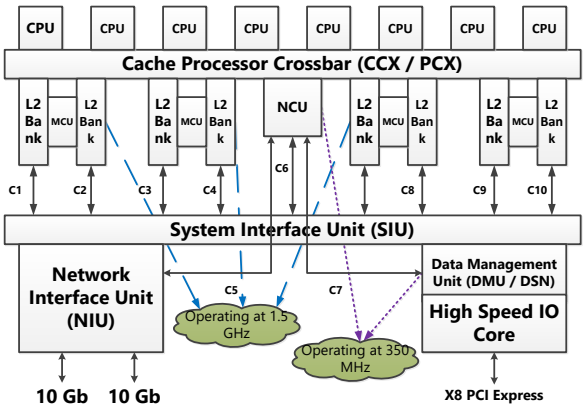


Figure 3.6: Block diagram of OpenSPARC T2 processor. **NCU**: Non-cacheable unit. **MCU**: Memory controller unit [156, 157].

number of sub-modules and the functionality of each of T2 design modules that we used in our experiment. ISCAS89 benchmarks contain up to 1728 flip-flops and 23815 total logic elements whereas the OpenSPARC T2 design modules contain up to 13746 flip-flops and 74350 total logic elements. The presence of a several orders of magnitude more flip-flops and logic elements in the different design modules of OpenSPARC T2 make these modules functionally complex and larger than the three largest designs of the ISCAS89 benchmark and the USB design.

Testbenches: To simulate and collect trace signal values from each of the OpenSPARC T2 design modules, we used our own constrained-random testbenches [159] written in SystemVerilog [10] as per the design specification. We could not use any tests that are included in the OpenSPARC T2 regression suites, since those tests were meant to simulate the whole SoC. We used SystemVerilog monitors during simulation and recorded trace signal values into an output trace file.

Tools used for comparison: We compared the scalability and quality of the selected trace signals of our PRoN method against those of several other state-of-the-art algorithms. We used SigSeT_1 [55], SigSeT_2 [58], HybrSel [60], and AASR [56]. Since SigSeT_1, SigSeT_2, HybrSel, and PRoN accept designs in ISCAS89 format, we converted the OpenSPARC T2 and USB design modules into ISCAS89 netlist format for comparison among these algorithms. We synthesized the OpenSPARC and USB design modules by using the Synopsys Design Compiler with the NanGate 45 nm library [160],

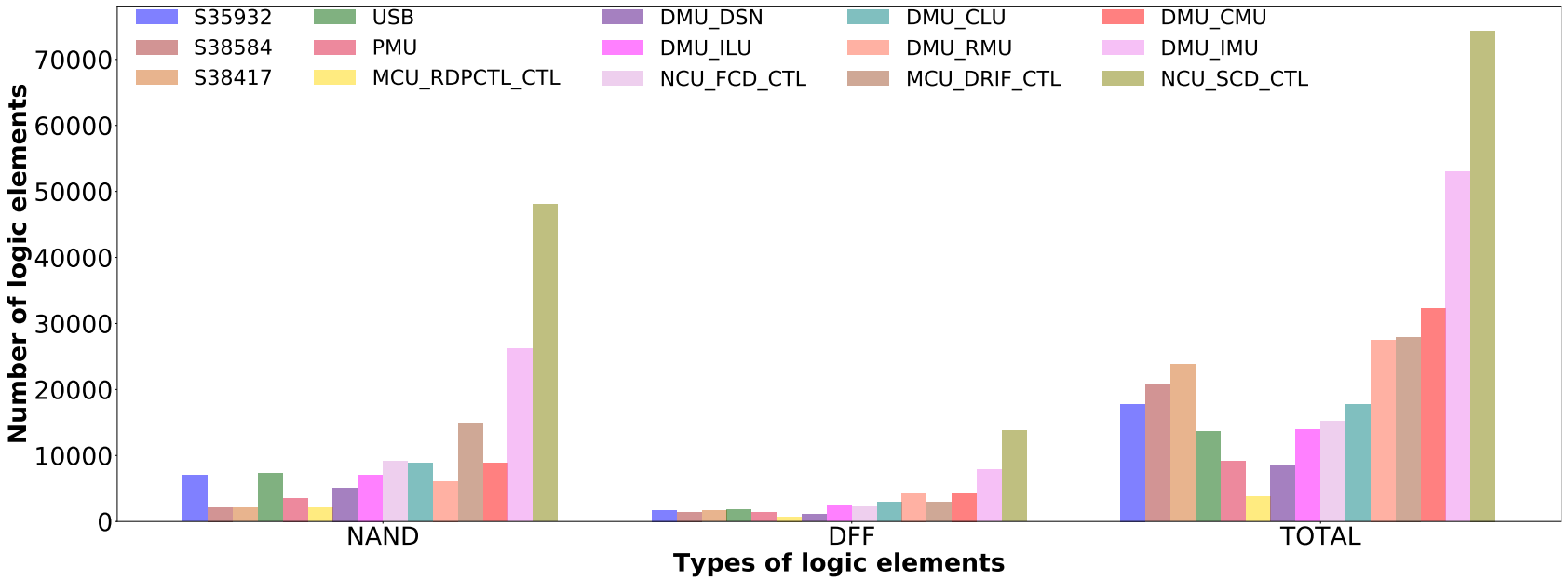


Figure 3.7: Comparison of number of different logic elements in ISCAS89 benchmarks, USB 2.0 and OpenSPARC T2 modules.

Table 3.1: Functional details of each of the OpenSPARC T2 design modules used in our experiment. **NoS**: No. of sub-modules in the design module excluding standard library cells. **LoC**: Total lines of code excluding standard library cells. **DMU**: Data management unit. **CCX**: Cache crossbar. **NCU**: Non-cacheable unit. **SIU**: System interface unit. **NIU**: Network interface unit. **MSI**: Message signal interrupts.

Module Name	**NoS**	**LoC**	**Module Functionality**
mcu_rdpctl_ctl	29	2872	Controlling memory read pointer for the memory controller unit
dmu_dsn	14	3156	Interface IP controlling data flow and interrupt between core side IPs and I/O side IPs
dmu_rmu	43	3376	DMU internal IP responsible for orderly movement of transaction data in and out of DMU pipeline
pmu	32	4008	SPARC core power management unit
dmu_clu	21	5211	DMU internal IP moving data related to memory read, memory write, DMA read, DMA write, and interrupts
dmu_cmu	15	5260	DMU internal IP managing DMU pipelines and serves as the ordering point for transactions in the in/out DMU pipeline
ncu_fcd_ctl	85	8427	Controlling clock-domain crossing data transfer between CCX and NCU at the core clock speed
dmu_ilu	57	10489	Interface IP controlling transaction level data flows between DMU and PCI express unit
mcu_drif_ctl	113	16493	Controls data movement between MCU and DRAM interface unit
ncu_scd_ctl	262	23882	Controlling clock-domain crossing data transfer between SIU, DMU, and NIU at the I/O clock speed
dmu_imu	239	77230	DMU internal IP serving MSIs, PCI Express messages, on-chip and internal interrupts (interrupts generated due to both error and events)

Table 3.2: Runtime and maximum memory usage of SigSeT_1 [55], HybrSel [60], and PRoN during the signal selection phase on ISCAS89 benchmarks and USB. The benchmarks are arranged in increasing order of total number of logic elements. **T**: Runtime in seconds. **Mem**: Peak memory usage in MB.

Bench mark	**Logic elements**		**SiGSeT_1**		**HybrSel**		**PRoN**	
	DFF	Total	T	Mem	T	Mem	T	Mem
USB2.0	1757	13601	181	385	735.36	1204.28	2.01	805.7
s35932	1728	17793	9.52	498	17.6K	389	7.74	275.8
s38584	1452	20705	150	285	8.94K	287	7.86	298.81
s38417	1636	23815	208	702	19.4K	359	9.54	326.9

and constrained the library such that the synthesized DC netlist contained only basic logic gates like AND, OR, NOT, NAND, NOR, and D flip-flop (DFF). We then converted the DC netlist into the ISCAS89 format. For AASR [56], we used the GTECH 180 nm library that is included in the Synopsys Design Compiler package, since AASR can only parse design netlists consisting of GTECH library logic elements.

Execution platform: All experiments on the ISCAS89 designs and USB were run on an AMD Opteron 8-core 22xx processor with 15GB of RAM. All experiments on the OpenSPARC T2 design modules were run on an Intel Xeon CPU E3-1240 8-core processor running at 3.4 GHz with 16 GB RAM. In most of our experiments, we used simulation based coverage metrics for behavioral coverage, including line coverage, condition coverage, branch coverage, toggle coverage, FSM coverage, and path coverage.

3.6 Experimental results

3.6.1 Scalability of different signal selection algorithms

In this experiment we show scalability in terms of runtime and peak memory usage of different signal selection algorithms based on SRR including SigSeT_1 [55], SigSeT_2 [58], HybrSel [60], and AASR [56]. We compare these algorithms to our PRoN algorithm.

For this experiment we use the three biggest designs of the ISCAS89

benchmark (namely s3592, s38417, and s38584), and USB 2.0 design, we compare SigSeT_1, HybrSel and our PRoN algorithm (Table 3.2). On the OpenSPARC T2, we compare many more SRR based signal selection tools with our PRoN algorithm. The tools under comparison are AASR, HybrSel, SigSeT_1, SigSeT_2, and PRoN. We run experiments on 11 large and complex design modules of the OpenSPARC T2 and compare runtime and peak memory usage (c.f., Figure 3.8a, Figure 3.8b, Figure 3.8c, and Table 3.3).

We consider a trace buffer width of 256 bits and trace buffer depth of 512 cycles. To record the maximum memory usage for ISCAS89 benchmarks and USB, we used the Massif tool in Valgrind [161]. For OpenSPARC T2 design module, we use the `datetime` package of Python to measure runtime. We use a virtual memory monitor written in Python to monitor the peak virtual memory usage of each algorithm during signal selection for OpenSPARC T2 design modules. We iterate PageRank until the values of the ranking matrix are stabilized. For each algorithm we set a timeout limit of 7,200 seconds. HybrSel, and AASR iteratively updates the restorability rate of each state element based on the current signal selection, a computationally intensive approach that is time consuming.

Note that in Table 3.2, PRoN uses considerably large peak memory usage for USB due to large fanouts of most of the logic elements (often more than five) causing high outdegree for many nodes in the $\mathcal{G}$ for USB. PageRank of a node thus propagates to many other connecting nodes requiring large number of iterations to converge. This causes high memory usage for PRoN for USB design.

We make the following observations from Table 3.2, Table 3.3 and Figure 3.8a, Figure 3.8b, and Figure 3.8c. From Figure 3.8a, we find that SigSet_1 could not complete signal selection for designs consisting of *more than 2,800* flip-flops because of its large peak memory usage of *30 GB or more*. In Figure 3.8b, we note that HybrSel failed to complete signal selection for any design containing *more than 2,900* flip-flops when the allowed time limit was varied *up to 7,200 seconds in steps of 1,800 seconds*. Hence in Table 3.3, we report the timeout value for HybrSel as *1,800 seconds*. Both SigSeT_2 and AASR failed to complete signal selection for any OpenSPARC T2 designs within the *allowed time limit of 7,200 seconds*. None of the SRR based signal selection algorithms finished for designs *greater than 17,743 logic elements*.

Table 3.3: Runtime and peak memory usage of SigSeT_1 [55], HybrSel [60], AASR [56], SigSeT_2 [58], and PRoN during the signal selection phase on OpenSPARC T2 design modules. OpenSPARC T2 design modules are arranged in increasing order of total number of logic elements. **DFF**: Total number of D flip-flops in a design module. **Total**: Total number of logic elements in a design module. **T**: Runtime in seconds. **Mem**: Peak memory usage in MB. ⊠: Signal selection failure due to excessive peak memory usage (30 GB or more). ⊖: Signal selection failure due to an error. ⋆: Signal selection failure due to timeout (1,800 seconds). ⊘: Signal selection failure due to timeout (7,200 seconds).

Module Name	**Logic Elements**		**SiGSeT_1**		**HybrSel**		**AASR**		**SigSeT_2**		**PRoN**	
	DFF	Total	T	Mem	T	Mem	T	Mem	T	Mem	T	Mem
mcu_rdpctl_ ctl	722	3720	1.13	50.21	107.11	1050.13	⊘	125.85	⊘	113.59	1.01	9.41
pmu	1351	9145	4.07	325.26	320.33	1193.32	⊘	151.8	⊘	113.59	2.01	746.51
dmu_dsn	1108	8409	3.01	351.62	556.20	932.67	⊘	138.22	⊘	113.59	2.01	724.26
dmu_ilu	2517	13948	⊖	⊖	1556.0	1205.31	⊘	174.66	⊘	145.19	2.01	806.0
ncu_fcd_ ctl	2397	15169	20.02	3069.11	771.79	1014.32	⊘	175.12	⊘	152.67	2.01	816.48
dmu_clu	2893	17743	55.06	8271.8	1479.0	1214.27	⊘	180.53	⊘	176.64	3.01	842.3
dmu_rmu	4279	27538	2.02	⊠	⋆	1240.87	⊘	209.87	⊘	297.77	3.01	949.66
mcu_drif_ ctl	2954	27945	272	⊠	⋆	1243.38	⊘	192.58	⊘	251.14	5.01	981.33
dmu_cmu	4246	32234	96.47	⊠	⋆	1240.87	⊘	215.6	⊘	345.49	8.01	1199.83
dmu_imu	7895	52962	245	⊠	⋆	1240.84	⊘	280.9	⊘	462.51	9.01	1249.83
ncu_scd_ ctl	13746	74350	255	⊠	⋆	196.18	⊘	393.64	⊘	645.69	12.01	1389.58

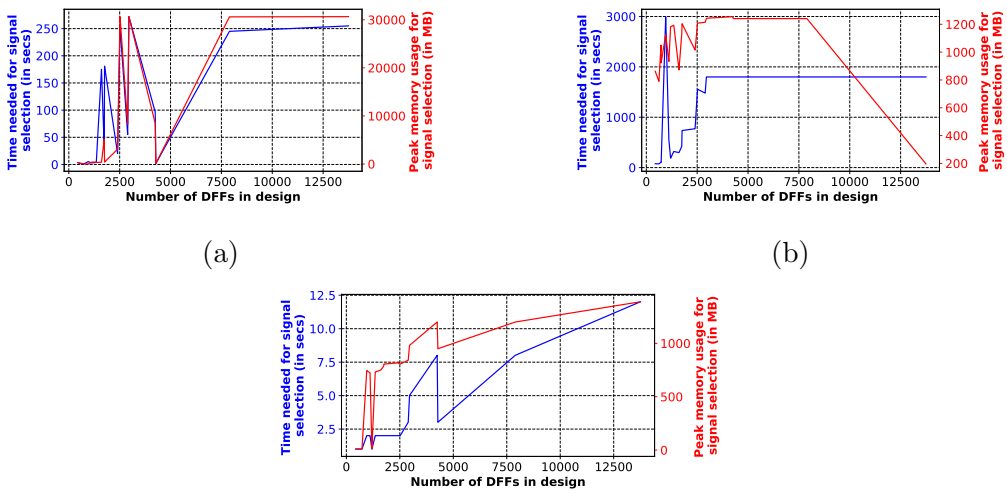


Figure 3.8: Scalability of different signal selection algorithms in terms of runtime (in seconds) and peak memory usage (in MB) for 11 different OpenSPARC T2 design modules using (a) SigSeT_1 [55], (b) HybrSel [60], and (c) PRoN.

In contrast, *our PRoN algorithm was able to complete signal selection within 13 seconds and with a peak memory usage of up to 1.5 GB for the largest OpenSPARC T2 design module consisting of 13,746 flip-flops.*

HybrSel, AASR, and SigSeT_2 update the rate of restorability of each flip-flop in the design in each iteration based on currently selected signals. For a large number of flip-flops of T2 design modules, this iterative update was computationally intensive and took more time and memory, and often failed to complete signal selection in a reasonable amount of time. PRoN is able to scale because it analyzes topography of the design identifying important variables.

Since two of the tools, namely, AASR and SigSeT_2 do not complete signal selection on any of the OpenSPARC T2 design modules, our comparison of selected signals in forthcoming experiments is limited to the two SRR-based tools that completed. Among them, since SigSeT_1 and HybrSel do not complete for any modules larger than the `dmu_clu`, we limit comparisons to the top six modules listed in Table 3.3.

This experiment shows that PRoN signal selection algorithm scales to industry standard large-scale designs compared to the state-of-the-art SRR-based signal selection techniques.

Table 3.4: Comparative analysis of SRR using SigSeT_1 [55], HybrSel [60], and PRoN on ISCAS89 benchmarks and USB 2.0.

Benchmark	**SiGSeT_1**	**HybrSel**	**PRoN**
s35932	4.7	4.7	4.7
s38417	4.0	3.8	3.9
s38584	4.7	4.6	4.7
USB2.0	3.7	3.5	3.5

Table 3.5: Comparative analysis of SRR for signals selected in Table 3.3 on OpenSPARC T2 design modules. **M1**: pmu. **M2**: mcu_rdpctl_ctl. **M3**: dmu_dsn. **M4**: dmu_ilu. **M5**: ncu_fcd_ctl. **M6**: dmu_clu. **Rand**: SRR calculated using trace values obtained from design simulation using random stimulus. **Sim**: SRR calculated using trace values obtained from design simulation using constrained random stimulus. ⊗: No SRR values as SigSeT_1 fails to select signals for dmu_ilu (c.f.,Table 3.3). ⊘: SRR calculation failed using random stimulus for ncu_fcd_ctl. §: Highest SRR achieved using random stimulus. ¶: Highest SRR achieved using constrained random stimulus.

Module	**SiGSeT_1**		**HybrSel**		**PRoN**	
Name	Rand	Sim	Rand	Sim	Rand	Sim
M1	3.83	2.55	14.54§	2.18	8.46	7.62¶
M2	2.97	2.27	4.06	3.32¶	29.65§	1.72
M3	13.61	6.52	14.1	2.92	14.14§	8.03¶
M4	⊗	⊗	33.16§	16.8	29.65	19.55¶
M5	⊘	1.97	⊘	6.71¶	⊘	3.98
M6	37.2	8.98	37.49§	9.83¶	35.99	8.41

3.6.2 Comparison of algorithms with respect to restorability

In this experiment, we compare the restorability (measured by SRR) achieved by algorithms designed to optimize the SRR metric with our PRoN algorithm, that is not designed to optimize this metric. Since SRR is the de facto standard to measure goodness of selected signals, we evaluate our algorithm according to this metric for the sake of completeness.

To calculate SRR values for the ISCAS89 benchmarks and USB 2.0 design, we simulate the designs using randomized testbenches. We use the top 20% of the signals selected by each method for each benchmark and restore signals for 5,000 cycles.

For the OpenSPARC T2 design modules, since we construct SystemVerilog testbenches (c.f., Section 3.5), we could use signal values from simulation

traces in addition to randomized signal values to calculate the SRR. We use a trace buffer width of 256 bits and a trace buffer depth of 512 cycles for both simulation-value-based and randomized-value-based SRR calculations.

Table 3.4 and Table 3.5 show comparative analysis of SRR values for different algorithms on ISCAS89, USB and different OpenSPARC T2 design modules respectively.

SRR calculation involves forward propagation and backward justification [55] for selected trace signals. In several cases, signal restoration tool was not able to compute SRR on the selected signals from Table 3.3. In one case, none of the randomized signal values converged, possibly because the number of signals to be restored overflowed and eventually restoration process ran out of memory.

On the ISCAS89 benchmarks, none of the methods outperformed the others. PRoN's SRR value is lower than that of SigSeT_1 and HybrSel. On OpenSPARC T2 design modules, HybrSel and PRoN *consistently performed better* than SigSeT_1. Table 3.5 shows that PRoN achieves the highest SRR values for three IP modules while using simulation-based trace values (¶ in column 7), and for two IP modules while using randomized trace values (§ in column 6). This is interesting, given that PRoN is not optimized for SRR.

We note that the SRRs of the ISCAS89 benchmarks in Table 3.4 are significantly lower than the values reported in previous papers [55, 56, 60]. SRR (c.f., Section 3.2.4) is a ratio and is defined as (total number of signals restored + total number of signals traced) / (total number of signals traced). Previous work [55, 56, 60] used a fixed-length trace buffer of size 8/16/32, and therefore the denominator is 8/16/32. If the average number of signals restored is 1000, the RR value will be 126/63/32. We select approximately 350 signals in each design, making our denominator very large. So even with 1200 signals restored, the SRR value is small. In Table 3.5, when trace signal values from simulation are used for restoration compared to randomized values, the SRR values of the T2 benchmarks were 4× smaller. The reason is that randomization assigns a concrete binary value of 1'b1 or 1'b0 to every selected trace signal at each cycle, effectively maximizing the SRR value. In a simulation, that is more reminiscent of the real scenario, there are cycles in which a 1'bX (an unknown value) is assigned to a traced signal. An unknown value does not help to restore any other new signal values, effectively reducing the SRR values. A traced signal may have 1'bX if it is part of a

control bus or a data bus. Whenever the control or data bus enable signal is de-asserted, the control bus or the data bus does not have a concrete binary value, causing the trace value of the traced signal to become 1'bX.

Section 3.6.1 concerns the scalability of each of the signal selection methods. AASR and HybrSel simulate the design netlist in each iteration for the specified number of cycles during signal selection to find out the best signals to trace. For these big designs, with a larger number of specified cycles, AASR and HybrSel take much longer time to complete signal selection and often times out. Hence, in order to have a fair comparison of runtime and peak memory usage for signal selection of all the methods on a reasonable number of designs, we used a trace buffer depth of 512 cycles. We found that for any trace buffer depth value of greater than 512 cycles, even HybrSel can only complete signal selection for no more than four designs, thereby reducing the value of this experiment.

This section concerns the signal restoration post tracing using traced signal values. We restore signals with traced signal values using a combination of forward propagation and backward justification. We can afford to restore up to 5000 cycles in this phase, since there is no iterative calculation, unlike in the signal selection phase.

This experiment demonstrates that signals selected by PRoN although not optimized for SRR, often achieve higher restorability.

3.6.3 Comparing behavioral coverage of selected signals

In this experiment we study the behavioral coverage achieved by the selected signals using different tools. In pre-silicon simulation, behavioral coverage metrics are intended to check for important high-level behavioral and functional coverage of the design.

In this experiment, we use USB 2.0 and OpenSPARC T2 design modules. For USB design, we trace values of 355 flip-flops for a simulation duration of 175 ms. Such a long trace is needed since at least 100 ms of simulation is required to activate different important states (such as the high-speed state mode of USB) of the USB line control module. We use the traced value of the selected signals along with five important input control signals as the stimulus in RTL and measure the behavioral coverage by using Synopsys

Table 3.6: Comparative analysis of trace signals from SigSeT_1 [55] and PRoN with respect to simulation-based coverage metrics for different USB modules. **u0**: usbf_utmi_if. **u1**: usbf_pl, **u2**: usbf_mem_arb. **u4**: usbf_rf. **u5**: usbf_wb. **L**: Line coverage. **C**: Condition coverage. **F**: FSM coverage. **B**: Branch coverage. **O**: Overall coverage. ⊘: VCS does not report the coverage value.

Module Name	**SigSeT_1**					**PRoN**				
	O	L	C	F	B	O	L	C	F	B
u0	29.73	35.81	28.89	15.75	38.35	49.61	71.81	38.89	28.75	58.97
u1	17.71	22.94	15.02	7.44	25.43	33.94	52.94	22.02	15.33	45.46
u2	41.44	33.45	⊘	⊘	49.43	68.75	75.00	⊘	⊘	62.50
u4	17.71	30.98	1.67	⊘	20.48	33.70	60.98	2.45	⊘	37.68
u5	23.27	31.90	13.49	15.37	32.33	40.14	56.34	19.23	31.67	53.33

Table 3.7: Comparative analysis of trace signals from PRoN and PageRank on RTL with respect to simulation-based coverage metrics for different USB modules. **u0**: usbf_utmi_if. **u1**: usbf_pl, **u2**: usbf_mem_arb. **u4**: usbf_rf. **u5**: usbf_wb. **L**: Line coverage. **C**: Condition coverage. **F**: FSM coverage. **B**: Branch coverage. **O**: Overall coverage. ⊘: VCS does not report the coverage value.

Module Name	**PageRank on RTL**					**PRoN**				
	O	L	C	F	B	O	L	C	F	B
u0	59.67	76.92	57.78	38.5	65.48	49.61	71.81	38.89	28.75	58.97
u1	43.67	58.91	35.41	24.98	52.97	33.94	52.94	22.02	15.33	45.46
u2	100.0	100.0	⊘	⊘	100.0	68.75	75.00	⊘	⊘	62.50
u4	40.12	69.34	4.91	⊘	46.12	33.70	60.98	2.45	⊘	37.68
u5	53.81	82.35	23.23	39.67	70.00	40.14	56.34	19.23	31.67	53.33

Table 3.8: Comparative analysis of trace signals from SigSeT_1 [55], and PRoN with respect to simulation-based coverage metrics for different OpenSPARC T2 design modules. **M1**: pmu. **M2**: mcu_rdpctl_ctl. **M3**: dmu_dsn. **M4**: dmu_ilu. **M5**: ncu_fcd_ctl. **M6**: dmu_clu. **L**: Line coverage. **C**: Condition coverage. **F**: FSM coverage. **B**: Branch coverage. **T**: Toggle coverage. **P**: Path coverage. **O**: Overall coverage. ⊘: VCS does not report the coverage value. ⊖: Coverage calculation was not possible since SigSeT_1 fails to select signals (Table 3.3). §: Highlighting path coverage.

Module Name	**SigSeT_1**							**PRoN**						
	O	L	C	F	B	T	P	O	L	C	F	B	T	P
M1	17.01	32.44	⊘	⊘	⊘	1.58	⊘	18.93	33.45	⊘	⊘	⊘	1.42	⊘
M2	37.23	65.49	37.50	⊘	38.36	2.59	⊘	37.69	67.21	38.64	⊘	39.36	2.57	⊘
M3	28.29	64.15	31.11	0.0	41.85	8.42	20.20§	28.73	60.95	32.89	0.0	42.49	5.07	23.00§
M4	⊖	⊖	⊖	⊖	⊖	⊖	⊖	30.12	61.24	50.94	0.0	42.72	1.98	23.86§
M5	20.32	31.76	25.69	⊘	23.58	0.23	⊘	16.25	28.91	26.07	⊘	19.58	1.46	⊘
M6	20.38	49.73	11.27	0.0	34.04	1.89	25.38§	25.35	53.86	26.76	0.0	38.65	4.40	28.44§

Table 3.9: Comparative analysis of trace signals from HybrSel [60], and PRoN with respect to simulation-based coverage metrics for different OpenSPARC T2 design modules. **M1**: pmu. **M2**: mcu_rdpctl_ctl. **M3**: dmu_dsn. **M4**: dmu_ilu. **M5**: ncu_fcd_ctl. **M6**: dmu_clu. **L**: Line coverage. **C**: Condition coverage. **F**: FSM coverage. **B**: Branch coverage. **T**: Toggle coverage. **P**: Path coverage. **O**: Overall coverage. ⊘: VCS does not report the coverage value. ¶: Highlighting path coverage.

Module Name	**HybrSel**							**PRoN**						
	O	L	C	F	B	T	P	O	L	C	F	B	T	P
M1	18.24	42.18	⊘	⊘	⊘	0.30	⊘	18.93	33.45	⊘	⊘	⊘	1.42	⊘
M2	29.48	59.34	28.41	⊘	27.40	2.77	⊘	37.69	67.21	38.64	⊘	39.36	2.57	⊘
M3	25.76	58.97	22.78	6.67	39.44	9.89	16.79¶	28.73	60.95	32.89	0.0	42.49	5.07	23.00¶
M4	26.59	55.45	42.95	0.0	39.44	5.07	16.89¶	30.12	61.24	50.94	0.0	42.72	1.98	23.86¶
M5	3.13	8.60	1.45	⊘	2.12	0.33	⊘	16.25	28.91	26.07	⊘	19.58	1.46	⊘
M6	18.93	47.99	8.45	0.0	32.50	0.80	23.85¶	25.35	53.86	26.76	0.0	38.65	4.40	28.44¶

VCS. The behavioral coverage consists of four components, namely *branch coverage*, *line coverage*, *condition coverage* and *FSM coverage*. Table 3.6 and Table 3.7 show the behavioral coverage values reported by VCS. For each of the methods, we do not report the FSM coverage for `u4 (usbf_rf)`, since it did not contain any state machines. Also, we do not report the FSM and conditional coverage for `u2 (usbf_mem_arb)`, as it is a combinational design module. For OpenSPARC T2 design modules, we traced values of 256 flip-flops for 512 cycles. We used the traced values to measure the behavioral coverage by using Synopsys VCS. The behavioral coverage consists of six components namely *line coverage*, *condition coverage*, *branch coverage*, *FSM coverage*, *toggle coverage*, and *path coverage*.[2] Table 3.8, and Table 3.9 show the behavioral coverage values reported by VCS. For SigSeT_1, VCS was able to calculate *path coverage* for two different design modules whereas for HybrSel and PRoN, VCS was able to calculate *path coverage* for three different design modules by using traced signal values. The path coverage values are highlighted with § in Table 3.8 and with ¶ in Table 3.9.

For the USB design, the behavioral coverage of signals selected by PRoN is *up to 42%* (with an *average of 19.6%*) greater than that of the signals selected by SigSeT_1. The signals from PageRank on RTL achieves behavioral coverage *up to 70%* (*average of 30%*) more than the signals from SigSeT_1. This experiment shows that compared to SigSeT_1, PRoN and PageRank on RTL selected more functionally relevant signals from the USB design.

For the OpenSPARC T2 design modules, the overall behavioral coverage of signals selected by PRoN is *up to 30.12%* (with an *average of 5.64%*) greater than that of the signals selected by SigSeT_1 and *up to 13.12%* (with an *average of 5.83%*) greater than that of the signals selected by HybrSel. For OpenSPARC T2 *we do not report FSM coverage for several design modules, since those modules do not contain any explicit state machines.*

Line coverage: Line coverage (Section 3.2.5) of signals selected by PRoN is *up to 61.25%* (with an *average of 10.34%*) greater than that of the signals selected by SigSeT_1 and *up to 20.31%* (with an *average of 5.52%*) greater than that of the signals selected by HybrSel.

Branch coverage: Branch coverage (Section 3.2.5) of signals selected by PRoN is *up to 17.46%* (with an *average of 8.38%*) greater than that of the

[2]We enabled path coverage in VCS by using the -lca option.

signals selected by SigSeT_1 and *up to 42.72%* (with an *average of 7.4%*) greater than that of the signals selected by HybrSel.

Condition coverage: Condition coverage (Section 3.2.5) of signals selected by PRoN is *up to 24.62%* (with an *average of 14.25%*) greater than that of the signals selected by SigSeT_1 and *up to 50.94%* (with an *average of 11.62%*) greater than that of the signals selected by HybrSel.

Path coverage: For SigSeT_1, the path coverage (Section 3.2.5) was *up to 25.38%* (with an *average of 22.79%*); for HybrSel the path coverage was *up to 23.85%* (with an *average of 20.32%*); and for PRoN, the path coverage was *up to 28.44%* (with an *average of 25.10%*). For large designs, like OpenSPARC T2 design modules, even a small increment in path coverage manifests in the execution of a large number of additional design paths. In our analysis, signals selected by PRoN achieved *up to 4.59%* (with an *average of 3.33%*) more path coverage than SigSeT_1 and HybrSel, implying that signals selected by PRoN executed a larger number of additional design paths compared to the signals selected by SigSeT_1 and HybrSel. This experiment shows that compared to SigSeT_1 and HybrSel, PRoN selects functionally superior signals for tracing from OpenSPARC T2 design modules. This result supports our modification to the PageRank metric to select important internal signals as demonstrated in Section 3.4.3.

Toggle coverage: Toggle coverage (Section 3.2.5) of signals selected by SigSeT_1 is *up to 3.35%* greater than that of PRoN but on average, toggle coverage of the signals selected by PRoN is *0.37% greater* than the signals selected by SigSeT_1. Toggle coverage of signals selected by HybrSel is *up to 4.82%* (with an *average of 2.26%*) greater than that of the signals selected by PRoN.

This experiment shows that signals selected by PRoN achieve higher behavioral coverage on industry standard large-scale designs outperforming the signals selected by the state-of-the-art SRR based techniques.

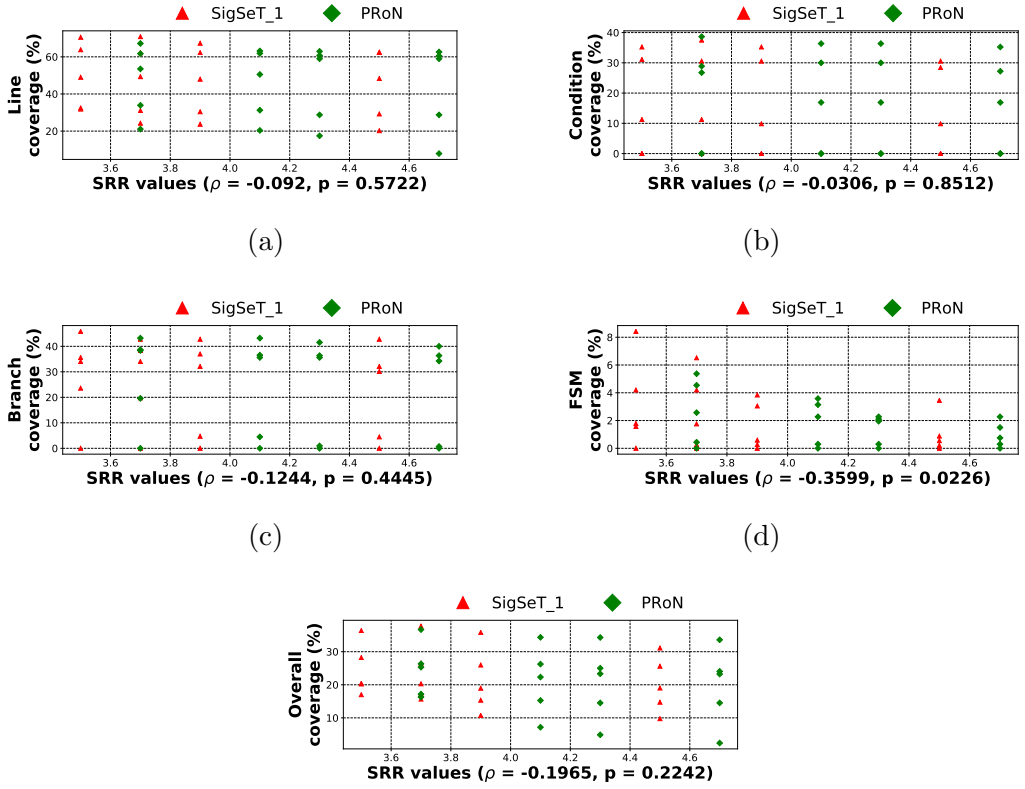


Figure 3.9: Graphs showing lack of correlation between SRR and for (a) Line coverage, (b) Condition coverage, (c) Branch coverage, (d) FSM coverage, and (e) Overall coverage on different USB modules u_0, ... u_5 for the signals selected by SigSeT_1 [55], and PRoN. ρ: Correlation co-efficient between SRR and the coverage metric. **p**: p-value indicating rejection probability for the null hypothesis of an uncorrelated system producing datasets that have ρ as extreme as the one computed from observed datasets.

3.6.4 Correlation analysis between SRR and high-level behavioral coverage metrics

This experiment finds if there is a correlation between high SRR values and the behavioral coverage metrics from pre-silicon, in order to determine the extent of high-level functional coverage of SRR. For each of the USB design modules, we traced top 5%, 10%, 15%, and 20% flip-flops per tool and for each of the OpenSPARC T2 design modules, we traced top 32, 64, 128, and 256 flip-flops per tool. We used the trace signal values and the design netlist to calculate the SRR value via backward justification and forward

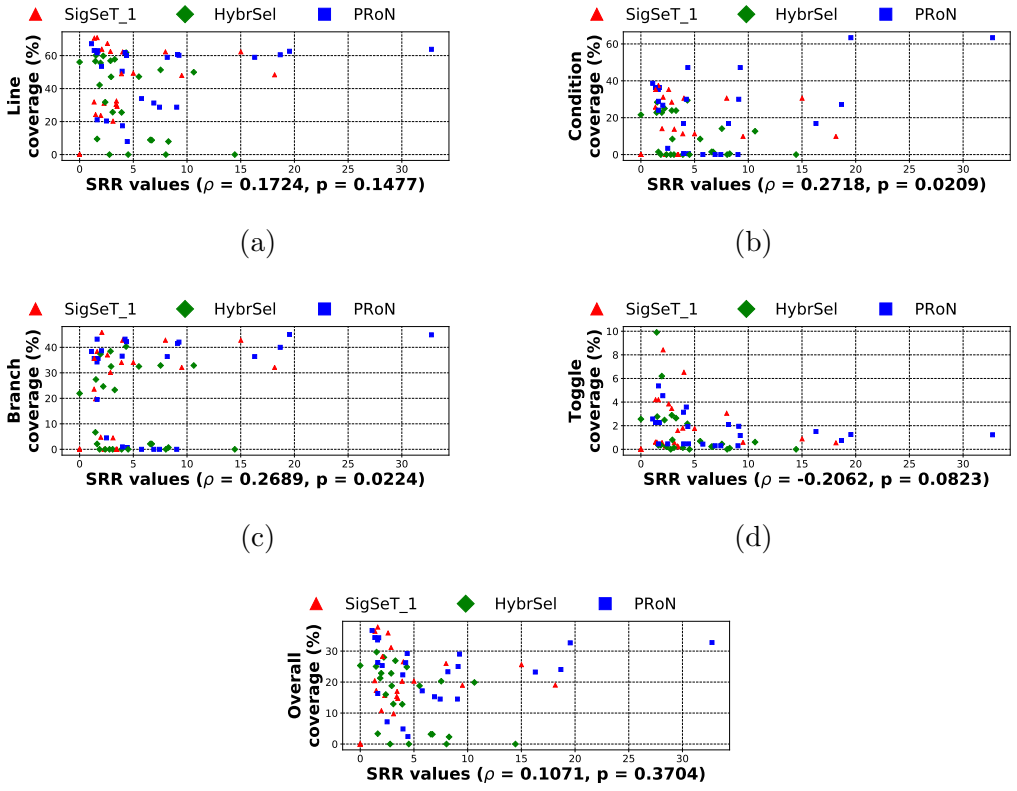


Figure 3.10: Graphs showing lack of correlation between SRR and for (a) Line coverage, (b) Condition coverage, (c) Branch coverage, (d) Toggle coverage, and (e) Overall coverage on different OpenSPARC T2 modules M_1, ... M_6 for the signals selected by SigSeT_1 [55], HybrSel [60], and PRoN. **M1**: pmu. **M2**: mcu_rdpctl_ctl. **M3**: dmu_dsn. **M4**: dmu_ilu. **M5**: ncu_fcd_ctl. **M6**: dmu_clu. ρ: Correlation co-efficient between SRR and the coverage metric. **p**: p-value indicating rejection probability for the null hypothesis of an uncorrelated system producing datasets that have ρ as extreme as the one computed from observed datasets

propagation [88]. Also, we used the traced signal values and the instrumented Verilog code of each of the design module and calculated different coverage metrics using Synopsys VCS. We use scatter plots to analyze correlation between SRR and the coverage metric values for each group of traced flip-flops for each of the design modules.

In Figure 3.9a, Figure 3.9b, Figure 3.9c, Figure 3.9d, and Figure 3.9e we analyze the correlation between SRR and the different components of behavioral coverage for USB design modules. In Figure 3.10a, Figure 3.10b, Figure 3.10c, Figure 3.10d, and Figure 3.10e, we analyze the correlation between SRR and the different components of behavioral coverage for OpenSPARC T2 design modules. For each such scatter plot, we have calculated the Pearson rank correlation coefficient ρ and have shown it below the scatter plot.

This experiment shows that there is no correlation between the SRR value and behavioral coverage. This underscores the point that a high SRR has low to no correlation with functional behavior.

3.6.5 Sensitivity analysis between behavioral coverage and trace buffer width and depth

This experiment finds the sensitivity of the behavioral coverage metrics from pre-silicon with the different configurations of the trace buffer width with a fixed trace buffer depth. For each of the OpenSPARC T2 design modules, we traced top 32, 64, 128, and 256 flip-flops per tool for 512 cycles. We use line plots to analyze the sensitivity between each of the coverage metrics and the different trace buffer width.

In Figure 3.11a, Figure 3.11b, Figure 3.11c, Figure 3.11d, and Figure 3.11e we analyze the sensitivity between different components of the behavioral coverage and different trace buffer width for OpenSPARC T2 design modules.

This experiment shows that the behavioral coverage increases with the increasing width of the trace buffer. This underscores the point that post-silicon observability is positively sensitive to the trace buffer width.

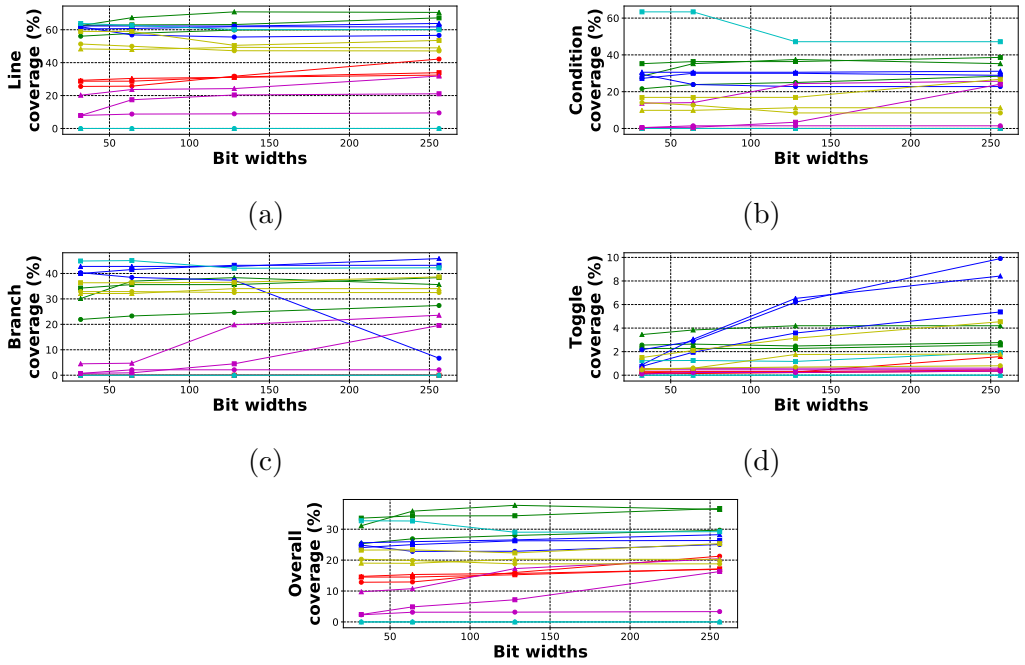


Figure 3.11: Graphs showing change in (a) Line coverage, (b) Condition coverage, (c) Branch coverage, (d) Toggle coverage, (e) Overall coverage and with different configuration of trace buffer width on different OpenSPARC T2 modules M_1, ... M_6 for the signals selected by SigSeT_1 [55], HybrSel [60], and PRoN. **M1**: pmu (■). **M2**: mcu_rdpctl_ctl (■). **M3**: dmu_dsn (■). **M4**: dmu_ilu (■). **M5**: ncu_fcd_ctl (■). **M6**: dmu_clu (■). **SigSeT_1**: ▲, **HybrSel**: ●, **PRoN**: ■.

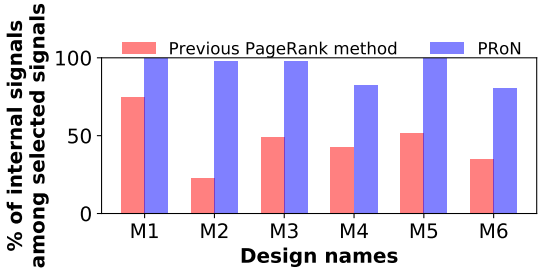


Figure 3.12: Comparison between signals selected by the traditional [63] PageRank algorithm and the modified PageRank algorithm of current work for various OpenSPARC T2 design modules. **M1**: pmu. **M2**: mcu_rdpctl_ctl. **M3**: dmu_dsn. **M4**: dmu_ilu. **M5**: ncu_fcd_ctl. **M6**: dmu_clu.

Table 3.10: Comparative analysis of trace signals from traditional PageRank algorithm [63], and PRoN with respect to simulation-based coverage metrics for different OpenSPARC T2 design modules. **M1**: pmu. **M2**: mcu_rdpctl_ctl. **M3**: dmu_dsn. **M4**: dmu_ilu. **M5**: ncu_fcd_ctl. **M6**: dmu_clu. **L**: Line coverage. **C**: Condition coverage. **F**: FSM coverage. **B**: Branch coverage. **T**: Toggle coverage. **P**: path coverage. **O**: Overall coverage. ⊘: VCS does not report the coverage value. ¶: Highlighting path coverage.

Module Name	**Traditional PageRank algorithm**							**PRoN**						
	O	L	C	F	B	T	P	O	L	C	F	B	T	P
M1	12.59	24.87	⊘	⊘	⊘	0.30	⊘	18.93	33.45	⊘	⊘	⊘	1.42	⊘
M2	28.73	59.34	25.00	⊘	28.77	1.81	⊘	37.69	67.21	38.64	⊘	39.36	2.57	⊘
M3	23.34	60.31	25.00	0.0	38.77	0.47	15.49¶	28.73	60.95	32.89	0.0	42.49	5.07	23.00¶
M4	25.94	55.47	44.46	0.0	32.11	1.17	18.45¶	30.12	61.24	50.94	0.0	42.72	1.98	23.86¶
M5	2.94	9.42	0.81	⊘	0.94	0.61	⊘	16.25	28.91	26.07	⊘	19.58	1.46	⊘
M6	21.66	59.27	11.27	0.0	34.42	0.84	24.16¶	25.35	53.86	26.76	0.0	38.65	4.40	28.44¶

3.6.6 Comparison of the signals selected by traditional PageRank algorithm and PRoN (modified PageRank algorithm)

This experiment demonstrates the improvement in signal selection of PRoN, the modified version of the PageRank algorithm over the traditional [63] PageRank algorithm. We compare the improvement in terms of i) the percentage of the selected internal design signals and ii) the improvement in the behavioral coverage metrics of the selected signals. For this experiment, we choose a trace buffer width of 256 bits for each of the OpenSPARC T2 design modules per method.

Comparison in terms of selected internal design signals: In Figure 3.12 we analyze the percentage of internal design signals among the signals that are selected for tracing by two methods. In traditional PageRank algorithm, *up to 77.34%* of selected signals are design output signals (with an *average of 54.10%*) whereas for the PRoN *no more than 19.14%* of selected signals are design output signals (with an *average of 6.77%*). While for traditional PageRank algorithm *up to 74.60%* of selected signals are internal design signals (with an *average of 45.89%*), for PRoN *up to 100%* of selected signals are internal design signals (with an *average of 93.23%*). Further analysis shows that the output signals that are selected by PRoN are connected to highly important internal design signals in a feedback loop making those signals relevant for tracing. PRoN selects *up to 77.35%* more internal design signals (with an *average of 41.80%*) for tracing compared to the traditional PageRank algorithm of [63].

This experiment shows that for tracing, the modified PageRank algorithm selects significantly more internal design signals compared to the traditional [63] PageRank algorithm method, thereby increasing post-silicon observability.

Comparison in terms of behavioral coverage: In this experiment, we study the behavioral coverage achieved by the selected signals using traditional PageRank algorithm [63] and PRoN. Our experimental setup and the behavioral coverage metrics that are used for this comparison, are similar to that of Section 3.6.3.

In Table 3.10 we compare the behavioral coverage of the signals that are selected by the two methods. Our analysis shows that the signals selected

Table 3.11: High-level functionality covered by PRoN and SigSet_1 selected signals on USB Netlist. **P:** Partial bit selected.

Signal Name	Module Name	Signal Functionality	Sig SeT_1	PRoN
no_bufs0	usbf_pe	**A.** Indicates available buffer size is less than payload size to switch to other buffer, **B.** BUF0 is full in DMA mode (Only BUF0 is used in DMA mode), **C.** Indicates if the BUF1 needs to be selected for next operation by the functional controller	✗	✓
token_pid _sel	usbf_pe	Handshaking signals indicating the packet accepting capacity of the buffer	✗	✓
dma_out _buf_avail	usbf_ep_rf	Indicates that there is a space for at least one MAX_PL_SZ packet in the buffer	✗	✓
inta, intb	usbf_rf	A fully programmable interrupt to provide full flexibility to software, the interrupts may be endpoint dependent or independent, indicating an error condition or overall events that have global meaning	✗	✓
state	usbf_pe	Indicates the states of operation of the USB protocol engine	✓	✓
state	usbf_utmi _ls	Indicates the states of operation of the USB protocol engine	P	✓
abort	usbf_pe	Indicates to abort an ongoing data transfer if the following conditions happen **A.** Buffer overflows (Received data packet size is too big and Rx_Data_Valid is asserted), **B.** Register end points matched and protocol engine is not in `IDLE` mode, **C.** Received packet size is more than MAX_PL_SZ	✗	✓
chirp _count	usbf_utmi _ls	A counter to initiate USB high speed mode	✗	✓
pid_seq _err	usbf_pe	An interrupt notifying USB function controller a loss of sync due to bad packets resulting in CRCs	✗	✓

by PRoN achieves *up to 13.31%* (*average 6.97%*) more overall behavioral coverage compared to the signals selected by the traditional PageRank algorithm. For large designs like OpenSPARC T2 design modules, even a small increment in path coverage manifests in the execution of a large number of additional design paths. In our experiment, signals selected by the PRoN achieves *up to 7.51%* (*average 5.73%*) path coverage compared to the traditional PageRank algorithm implying that signals selected by the PRoN executed a large number of additional design paths compared to the signals selected by the traditional PageRank algorithm.

This experiment shows that signals selected by PRoN achieve superior behavioral coverage on industry standard large-scale designs outperforming the signals selected by the traditional PageRank algorithm.

3.6.7 High-level functionality selected by PRoN on USB netlist

To give a flavor of the type of high-level functionality captured by the signals selected, we provide a qualitative analysis of two algorithms, PRoN and SigSet_1. In Table 3.11, for each signal, we list the corresponding RTL module and its high-level functionality. PRoN selects all the FSM state registers of the USB protocol engine (`usbf_pe`) and the USB line state module (`usbf_utmi_ls`) and other important signals. On the other hand, SigSet_1 selects only one signal completely and the other partially.

3.7 Conclusion

In light of our experimental findings, the use of SRR as a signal selection metric is not advisable. Instead, an alternate metric needs to be proposed for hardware signal tracing, such as assertion coverage [61, 62, 63]. This comprises the number of assertions that can be evaluated using the traced signal values. This metric certainly captures high-level behavioral intent since it uses assertions. It may be noted that it depends heavily on the quality of assertions, increasing the subjectivity of the approach. There is a need to define and characterize a metric for signal selection that reflects high level functionality better than SRR.

In conclusion, we have shown that the state restoration ratio as a metric does not reflect the behavioral coverage of the design relevant to practical post-silicon debugging. Unsurprisingly, we found no study reporting on the usage of SRR based methods on industry-scale design; all reported applications have been on small benchmarks (*e.g.*, ISCAS89) that are not representative of the complexities of an industrial integrated circuit (IC). The current and future needs of industry are better served if more representative metrics are used for signal selection. We present a signal selection method based on analyzing structural connectivity of the circuit netlist and RTL which in turn selects functionally relevant signals by computing variable importance. We demonstrate experiments at a scale and complexity that has hitherto never been used in hardware signal tracing literature. Our algorithm can scale to very large designs with moderate usage of computing resources, and selects high-quality signals that closely reflect high-level behavioral functionality.

CHAPTER 4

APPLICATION LEVEL HARDWARE TRACING FOR SCALING POST-SILICON DEBUGGING

4.1 Introduction

An expensive component of post-silicon SoC validation is application-level use-case validation. Use-case validation forms a key part of compatibility validation (c.f., Section 2.1.1) that requires considerable amount of manual effort and often takes weeks to months of validation time. Consequently, it is critical to determine techniques to streamline and automate this activity.

In this chapter, we develop a method for hardware tracing that specifically targets post-silicon use-case validation (c.f., Problem PR2 of Figure 1.10). The key idea is to raise the design abstraction level at which we apply hardware tracing. We apply hardware tracing at the *application-level* instead of applying at the netlist-level and behavioral-level of Chapter 3 (c.f., Figure 1.8). Given a collection of use-case scenarios and the system-level protocols that they activate (and the constituent messages), our algorithm computes the messages that are most valuable for debugging and error localization. We also develop heuristics for maximizing trace buffer utilization in the context of message selection.

Although state-of-the-art methods [55, 56, 57, 63, 95] optimize SRR to quantify signal restorability of the selected signals, a high restorability (SRR) of gate level signals may not correspond to crucial message buffers for the application use-cases. In our experiments on a USB controller design, we found that existing signal selection techniques could reconstruct *no more than 26%* of required interface messages across various design blocks. Analyzing at the application level provides our method the context to select *100%* of the messages required for debugging.[1] **This underlines the need for a**

[1]SRR based algorithms typically select flip-flops internal to the design for tracing whereas our method selects interface registers (either incoming or outgoing) for the relevant IPs for tracing.

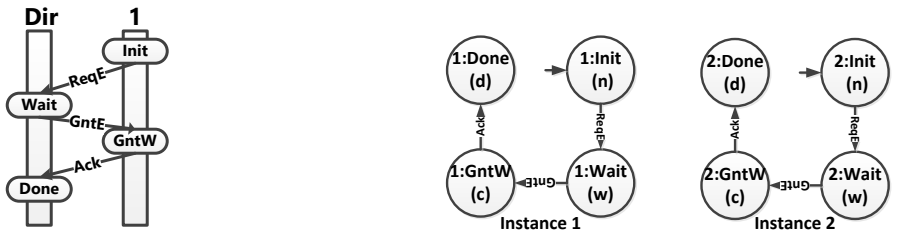
focused approach for message selection that accounts for protocols induced during use-case validation.

To show scalability and viability of our approach, we perform our experiments on a publicly available multicore SoC design OpenSPARC T2 SoC [156, 157] (c.f., Section 3.5). The scale and complexity is orders of magnitude more than traditional ISCAS89 benchmarks used to demonstrate signal selection techniques. We inject complex and subtle bugs, with each bug symptom taking several hundred observed messages (*up to 457* messages) and several hundred thousands of clock cycles (*up to 21,290,999* clock cycles) to manifest. Our analysis shows that we can achieve *up to 100%* trace buffer utilization (*average 98.96%*) and *up to 99.86%* flow specification coverage (*average 94.3%*). Our messages are able to localize each bug to *no more than 6.11%* of the total paths that could be explored. Our selected messages helped to eliminate *up to 88.89%* of potential root causes (*average 78.89%*) and localize to a small set of root causes.

Our method needs a priori definition of system-level protocols at transaction level. Our framework uses protocol formalizations as sequences of transactions or *flows*. There is an increasing trend to generate transaction-level models specifically with formalizations like flows, to enable early validation, prototyping, and software development activities [14, 15, 16, 162]. Our work shows how to leverage this collateral for post-silicon trace selection.

We make following important contributions.

- We propose the first solution to scale hardware tracing to industrial-scale realistic SoCs.
- We develop a targeted message selection for hardware tracing targeted toward post-silicon use-case (application level) validation by leveraging available architectural collaterals (*e.g.*, messages, transaction flows).
- We propose a technique based on mutual information gain to select trace messages at the application level. The selected messages are of high quality and effective for post-silicon use-case failure debugging.



(a) (b)

Figure 4.1: (a) shows a *flow* for an exclusive line access request for a cache coherence flow [16] along with participating IPs. (b) shows two legally indexed instances of cache coherence flow.

4.2 Preliminaries

Conventions: In SoC designs, a message can be viewed as an assignment of Boolean values to the interface signals of a hardware IP. In our formalization below, we leave the definition of message implicit, but we will treat it as a pair $\langle \mathcal{C}, w \rangle$ where $w \in \mathbb{Z}^+$. Informally, $\mathcal{C}$ represents the content of the message and w represents the number of bits required to represent $\mathcal{C}$. Given a message $m = \langle \mathcal{C}, w \rangle$, we will refer to w as *bit-width of* m, denoted by $width(m)$ or $|m|$.

Definition 7 *A* **flow** *is a directed acyclic graph (DAG) defined as a tuple,* $\mathcal{F} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{S}_p, \mathcal{E}, \delta_{\mathcal{F}}, Atom \rangle$ *where* $\mathcal{S}$ *is the set of flow states,* $\mathcal{S}_0 \subseteq \mathcal{S}$ *is the set of initial states,* $\mathcal{S}_p \subseteq \mathcal{S}$ *and* $\mathcal{S}_p \cap Atom = \emptyset$ *is called the set of stop states,* $\mathcal{E}$ *is a set of messages,* $\delta_{\mathcal{F}} \subseteq \mathcal{S} \times \mathcal{E} \times \mathcal{S}$ *is the transition relation and* $Atom \subset \mathcal{S}$ *is the set of atomic states of the flow.*

We use $\mathcal{F}.\mathcal{S}, \mathcal{F}.\mathcal{E}$ etc. to denote the individual components of a flow $\mathcal{F}$. A *stop* state of a flow is its final state after its successful completion. *Atom* is a *mutex* set of flow states *i.e.*, any two flow states in *Atom* cannot happen together. Other components of $\mathcal{F}$ are self-explanatory. In Figure 4.1a, we have shown a cache coherence flow along with the participating IPs and the messages. In Figure 4.1a, $\mathcal{S}$ = {Init, Wait, GntW, Done}, $\mathcal{S}_0$ = {Init}, $\mathcal{S}_p$ = {Done}, $Atom$ = {GntW}. Each of the messages in the cache coherence flow is 1 bit wide, hence $\mathcal{E} = \{\langle ReqE, 1 \rangle, \langle GntE, 1 \rangle, \langle Ack, 1 \rangle\}$.

Definition 8 *Given a flow* $\mathcal{F}$*, an* **execution** ρ *is an alternating sequence of flow states and messages ending with a stop state. For flow* $\mathcal{F}$*,* ρ =

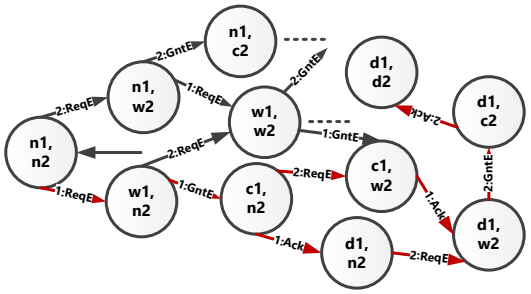


Figure 4.2: Two instances of cache coherence flow of Figure 4.1a interleaved.

$s_0\ \alpha_1\ s_1\ \alpha_2\ s_2\ \alpha_3\ \ldots\ \alpha_n\ s_n$ *such that* $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}, \forall 0 \leq i < n,\ s_i \in \mathcal{F}.S,\ \alpha_{i+1} \in \mathcal{F}.\mathcal{E},\ s_n \in \mathcal{F}.\mathcal{S}_p$. **Trace** *of an execution* ρ *is defined as* $trace(\rho) = \alpha_1\ \alpha_2\ \alpha_3\ \ldots\ \alpha_n$.

An example of an execution of the cache coherence flow of Figure 4.1a is $\rho = \{n, ReqE, w, GntE, c, Ack, d\}$ and $trace(\rho) = \{ReqE, GntE, Ack\}$.

Intuitively, a flow provides a pattern of system execution. A flow can be invoked several times, even concurrently, during a single run of the system. To make precise the relation between an execution of the system with participating flows, we need to distinguish between these instances of the same flow. The notion of *indexing* accomplishes that by augmenting a flow with an "index".

Definition 9 *An* **indexed message** *is a pair* $\alpha = \langle m, i \rangle$ *where* m *is the message and* $i \in \mathbb{N}$*, referred to as the* **index** *of* α*. An* **indexed state** *is a pair* $\hat{s} = \langle s, j \rangle$ *where* s *is a flow state and* $j \in \mathbb{N}$*, referred as the index of* $\hat{s}$*. An* **indexed flow** $\langle \mathcal{F}, k \rangle$ *is a flow consisting of indexed message* m *and indexed state* $\hat{s}$ *indexed by* $k \in \mathbb{N}$.

Figure 4.1b shows two instances of the cache coherence flow of Figure 4.1a indexed with their respective instance number. In our modeling, we ensure by construction that two different instances of the same flow do not have same indices. Note that in practice, most SoC designs include architectural support to enable *tagging*, *i.e.*, uniquely identifying different concurrently executing instances of the same flow. Our formalization simply makes the notion of tagging explicit.

Definition 10 *Any two indexed flows* $\langle \mathcal{F}, i \rangle$, $\langle \mathcal{G}, j \rangle$ *are said to be* **legally indexed** *either if* $\mathcal{F} \neq \mathcal{G}$ *or if* $\mathcal{F} = \mathcal{G}$ *then* $i \neq j$.

Figure 4.1b shows two legally indexed instances of the cache coherence flow of Figure 4.1a. Indices uniquely identify each instance of the cache coherence flow.

A **usage scenario** is a pattern of frequently used applications. Each such pattern comprises multiple interleaved flows corresponding to communicating hardware IPs.

Definition 11 *Let* $\mathcal{F}, \mathcal{G}$ *be two legally indexed flows. The interleaving* $\mathcal{F} ||| \mathcal{G}$ *is a flow called* **interleaved flow** *defined as* $\mathcal{U} = \mathcal{F} ||| \mathcal{G} = \langle \mathcal{F}.\mathcal{S} \times \mathcal{G}.\mathcal{S}, \mathcal{F}.\mathcal{S}_0 \times \mathcal{G}.\mathcal{S}_0, \mathcal{F}.\mathcal{S}_p \times \mathcal{G}.\mathcal{S}_p, \mathcal{F}.\mathcal{E} \cup \mathcal{G}.\mathcal{E}, \delta_{\mathcal{U}}, \mathcal{F}.Atom \cup \mathcal{G}.Atom \rangle$ *where* $\delta_{\mathcal{U}}$ *is defined as:*

$$i)\ \frac{s_1 \xrightarrow{\alpha} s_1' \ \wedge\ s_2 \notin \mathcal{G}.Atom}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1', s_2 \rangle} \quad \text{and} \quad ii)\ \frac{s_2 \xrightarrow{\beta} s_2' \ \wedge\ s_1 \notin \mathcal{F}.Atom}{\langle s_1, s_2 \rangle \xrightarrow{\beta} \langle s_1, s_2' \rangle}$$

where $s_1, s_1' \in \mathcal{F}.\mathcal{S}$, $s_2, s_2' \in \mathcal{G}.\mathcal{S}$, $\alpha \in \mathcal{F}.\mathcal{E}$, $\beta \in \mathcal{G}.\mathcal{E}$. *Every path in the interleaved flow is an execution of* $\mathcal{U}$ *and represents an interleaving of the messages of the participating flows.*

Rule i of $\delta_{\mathcal{U}}$ says that if s_1 evolves to the state s_1' when message α is performed and if g has a state s_2 which is not atomic/indivisible, then in the interleaved flow, if we have a state (s_1, s_2), it evolves to state (s_1', s_2) when message α is performed. A similar explanation holds good for Rule ii of $\delta_{\mathcal{U}}$. For any two concurrently executing legally indexed flow $\mathcal{F}$ and $\mathcal{G}$, $J = \mathcal{F} ||| \mathcal{G}$, for any $s \in \mathcal{F}.Atom$ and for any $s' \in \mathcal{G}.Atom$, $(s, s') \notin J.\mathcal{S}$. If one flow is in one of its atomic/indivisible state, then no other concurrently executing flow can be in its atomic/indivisible state.

Figure 4.2 shows partial interleaving $\mathcal{U}$ of two legally indexed flow instances of Figure 4.1b. Since c_1 and c_2 both are atomic state, state (c_1, c_2) is an illegal state in the interleaved flow. $\delta_{\mathcal{U}}$ and the *Atom* set make sure that such illegal states do not appear in the interleaved flows.

Trace buffer availability is measured in terms of bits thus rendering bit width of a message important. In Definition 12, we define a message combination. Different instances of the same message i.e. indexed messages are not required while computing the bit width of the message combination.

Definition 12 *A* **message combination** $\mathcal{M}$ *is an unordered set of messages. The* **total bit width** W *of a message combination* $\mathcal{M}$ *is the sum total of the bit width of the individual messages contained in* $\mathcal{M}$ *i.e.* $W(\mathcal{M}) = \sum_{i=1}^{k} width(m_i) = \sum_{i=1}^{k} |m_i|, m_i \in \mathcal{M}, k = |\mathcal{M}|$.

We introduce a metric called **flow specification coverage** to evaluate the quality of a *message combination*.

Definition 13 *Let* $\mathcal{F}$ *be a flow. The* **visible flow states** $visible(\alpha)$ *of a message* $\alpha \in \mathcal{F}.\mathcal{E}$ *is defined as the set of flow states reached on the occurrence of message* α i.e., $visible(\alpha) = \{s' | s \xrightarrow{\alpha} s', s, s' \in \mathcal{F}.\mathcal{S}\}$. *The* **flow specification coverage** $FCov(\mathcal{M})$ *of a message combination* $\mathcal{M}$ *is defined as the set union of the visible flow states of all the messages in the message combination, expressed as a fraction of the total number of flow states* i.e., $FCov(\mathcal{M}) = \frac{\cup_{i=1}^{k} visible(\alpha_i)}{|\mathcal{F}.\mathcal{S}|}, k = W(\mathcal{M})$.

4.3 Entropy and mutual information gain

4.3.1 Entropy

The *entropy* measures the uncertainty in a random variable. It was first proposed by Shannon [163]. Entropy originated in the information theory, but we repurpose it to use as a key component in our post-silicon validation solution. Let X be a discrete random variable with possible values $X_{val} = \{x_1, x_2, \ldots, x_n\}$. Let $p(x)$ be the associated probability mass function of X. The entropy of the random variable X is defined as follows.

$$\mathcal{H}(X) = -\sum_{x_i \in X_{val}} p(x_i) log_2 p(x_i) \tag{4.1}$$

where $p(x_i) = \frac{|X = x_i|}{|X_{val}|}$ denotes the fraction of X in which $X = x_i$.

4.3.2 Mutual information gain

In information theory, the *mutual information gain* measures the amount of information that can be obtained about one random variable X by observing

another random variable Y. The concept of mutual information gain is heavily dependent on *entropy* (c.f., Section 4.3.1). More precisely, the conditional entropy of a random variable X with respect to another random variable Y is the reduction in uncertainty in the realization of X when the outcome of Y is known. Mathematically, the mutual information gain $I(X;Y)$ can be defined in terms of conditional entropy as follows.

$$\begin{aligned} I(X;Y)) &= \mathcal{H}(X) - \mathcal{H}(X|Y) \\ &= \mathcal{H}(Y) - \mathcal{H}(Y|X) \\ &= \mathcal{H}(X) + \mathcal{H}(Y) - \mathcal{H}(X,Y) \\ &= \mathcal{H}(X,Y) - \mathcal{H}(X|Y) - \mathcal{H}(Y|X) \end{aligned} \tag{4.2}$$

where $\mathcal{H}(X)$ and $\mathcal{H}(Y)$ are the marginal entropies, $\mathcal{H}(X|Y)$ and $\mathcal{H}(Y|X)$ are the conditional entropies, and $\mathcal{H}(X,Y)$ is the joint entropy of X and Y. The mutual information gain $I(X;Y)$ is a non-negative quantity. Consequently, $\mathcal{H}(X) \geq \mathcal{H}(X|Y)$. For the case of jointly distributed discrete random variables X and Y, the conditional entropy can be defined as follows.

$$\begin{aligned} \mathcal{H}(X|Y) &= -\sum_{x,y} p(x,y) log_2 \left(\frac{p(x,y)}{p(y)} \right) \\ &= \sum_{y} p(y) \mathcal{H}(X|Y=y) \end{aligned} \tag{4.3}$$

For jointly distributed discrete random variables X and Y, the mutual information gain can be defined as follows.

$$I(X;Y) = \sum_{x,y} p(x,y) log_2 \left(\frac{p(x,y)}{p(x)p(y)} \right) \tag{4.4}$$

Maximizing information gain is done in order to increase flow specification coverage during post-silicon debugging of usage scenarios. The message selection procedure considers the *message combination* $\mathcal{M}$ for tracing, whereas to calculate information gain over $\mathcal{U}$, it uses *indexed messages*.

Given a set of legally indexed participating flows of a usage scenario $\mathcal{U}$, bit widths of associated messages, and a trace buffer width constraint, **our method selects a message combination $\mathcal{M}$ such that information gain is maximized over the interleaved flow $\mathcal{U}$** and the **trace buffer**

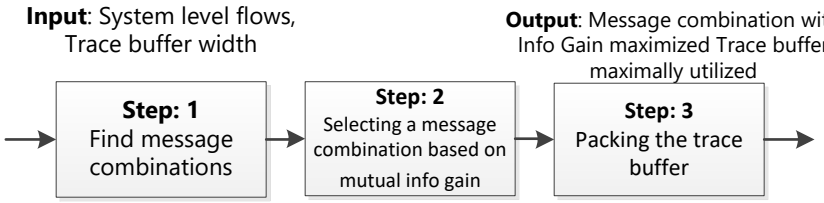


Figure 4.3: Our message selection methodology.

is maximally utilized.

4.4 Our message selection methodology

For the cache coherence flow example of Figure 4.1a, we assume a trace buffer width of 2 bits and concurrent execution of two instances of the flow. *ReqE, GntE*, and *Ack* messages happen between *1-Dir, Dir-1*, and *1-Dir* IP pairs respectively. *ReqE, GntE*, and *Ack* consist of `req, gnt`. and `ack` IP signal and each of the messages is 1-bit wide. Let $\mathbb{B} = \{0, 1\}$ be a binary set. Following the conventions of a message in Section 4.2, $\mathcal{C}(ReqE) = \mathbb{B}^{|req|}$, $\mathcal{C}(GntE) = \mathbb{B}^{|gnt|}$, and $\mathcal{C}(Ack) = \mathbb{B}^{|ack|}$ denote the respective message contents. Figure 4.3 shows our message selection methodology.

4.4.1 Step 1: Finding message combinations

In Step 1, we identify all possible message combinations from the set of all messages of the participating flows in a usage scenario.

While we find different message combinations, we also calculate the total bit width of each such message combinations. Any message combination that has a total bit width less than or equal to the available trace buffer width is kept for further analysis in Step 2.[2] Each such message combination is a potential candidate for tracing.

In the example of Figure 4.1a, there are three messages and $\sum_{k=1}^{3} \binom{3}{k} = 7$ different message combinations. Of these, only one (*ReqE, GntE, Ack*) has

[2]For multi-cycle messages, the number of bits that can be traced in a single cycle is considered to be the message bit width.

a bit width more than the trace buffer width of two bits. We retain the remaining six message combinations for further analysis in Step 2.

4.4.2 Step 2: Selecting a message combination based on mutual information gain

In this step, we compute the mutual information gain of the message combinations computed in Step 1 over the interleaved flow. We then select the message combination that has the **highest mutual information gain** for tracing.

We use *mutual information gain* as a metric to evaluate the quality of the selected set of messages with respect to the interleaving of a set of flows. We associate two random variables with the interleaved flow namely X and Y_i. X represents the different states in the interleaved flow i.e. it can take any value in the set $\mathcal{S}$ of the different states of the interleaved flow. Let $\mathcal{M} = \bigcup_i \mathcal{E}_i$ be the set of all possible indexed messages in the interleaved flow. Let Y_i' be a candidate message combination and Y_i be a random variable representing all indexed messages corresponding to Y_i'. All values of X are equally probable since the interleaved flow can be in any state and hence $p_X(x) = \frac{1}{|\mathcal{S}|}$. To find the marginal distribution of Y_i, we count the number of occurrences of each indexed message in the set $\mathcal{M}'$ over the entire interleaved flow. We define $p_{Y_i}(y) = \frac{\text{\# of occurrences of } y \text{ in flow}}{\text{\# of occurrences of all indexed messages in flow}}$. To find the joint probability, we use the conditional probability and the marginal distribution i.e. $p(x, y) = p(x|y)p(y) = p(y|x)p(x)$. $P(x|y)$ can be calculated as the fraction of the interleaved flow states x is reached after the message $Y_i = y$ has been observed. In other words, $p(x|y)$ is the fraction of times x that are reached, from the total number of occurrences of the indexed message y in the interleaved flow i.e. $p_{X|Y_i}(x|y) = \frac{\text{\# occurrence of } y \text{ in flow leading to x}}{\text{total \# occurrences of } y \text{ in flow}}$. Now we substitute these values in $I(X;Y)$ to calculate the mutual information gain of the state set X w.r.t. Y_i.

In Figure 4.2, $p_X(x) = \frac{1}{15} \forall x \in \mathcal{S}$. Let $Y_1' = \{GntE, ReqE\}$ be a candidate message combination and $Y_1 = \{$*1:GntE, 2:GntE, 1:ReqE, 2:ReqE*$\}$. For $I(X;Y_1)$, we have $p(y = y_i) = \frac{3}{18}, \forall y_i \in Y_1$. Therefore, $p_{X|Y_1}(x|1 : GntE) = \{1/3 \text{ if } x = (c1, n2), 1/3 \text{ if } x = (c1, w2), 1/3 \text{ if } x = (c1, d2)\}$ and $p_{X,Y_1}(x, 1 : GntE) = \{1/18 \text{ if } x = (c1, n2), 1/18 \text{ if } x = (c1, w2), 1/18 \text{ if } x = (c1, d2)\}$.

Similarly, we calculate $p_{X,Y_1}(x, 2 : GntE)$, $p_{X,Y_1}(x, 1 : ReqE)$ and $p_{X,Y_1}(x, 2 : ReqE)$. The mutual information gain is given by $I(X, Y_1) = 1.073$ where $I(X, Y_1) = \sum_{x,y} p(x, y) log \frac{p(x,y)}{p(x)p(y)}$.

Similarly, we calculate the mutual information gain for the remaining five message combinations. We then select the message combination that has the highest mutual information gain, which is $I(X, Y_1) = 1.073$, thereby selecting the message combination $Y_1' = \{\text{ReqE, GntE}\}$ for tracing. Intuitively, in an execution of $\mathcal{U}$ as shown in Figure 4.2, if the observed trace is {*1:ReqE, 1:GntE, 2:ReqE*}, immediately we can intuitively localize the execution to two paths shown in red in Figure 4.2 among the many possible paths of $\mathcal{U}$.

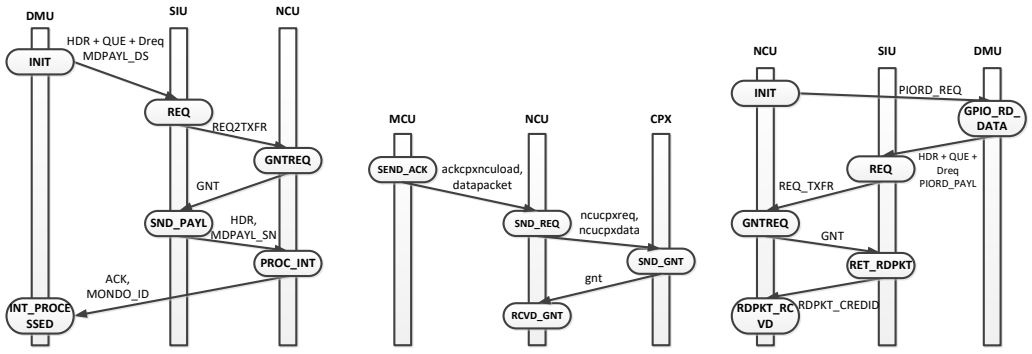
4.4.3 Step 3: Packing the trace buffer

Message combinations with the highest mutual information gain selected in Step 2 may not completely fill the trace buffer. To maximize trace buffer utilization, in this step we *pack* smaller message groups that are small enough to fit in the leftover trace buffer width. Usually, these smaller message groups are part of a larger message that cannot be fit into the trace buffer, *e.g.* in OpenSPARC T2, `dmusiidata` is a 20 bit-wide message whereas `cputhreadid` a subgroup of `dmusiidata` is 6 bits wide. We select a message group that can fit into the leftover trace buffer width, such that the information gain of the selected message combination in union with this smaller message group is maximal. We repeat this step until no more smaller message groups can be added in the leftover trace buffer. The benefits of packing are shown empirically in Section 4.6.1.

In our example, the trace buffer is filled up by the set of selected message combination. The flow specification coverage achieved with Y_1' is 0.7333.

4.5 Experimental setup

Design testbed: We primarily use the publicly available OpenSPARC T2 SoC [156, 157] to demonstrate our result. Figure 3.6 shows an IP level block diagram of OpenSPARC T2. Table 4.1 shows three different usage scenarios considered in our debugging case studies along with participating flows (column 2-6) and participating IPs (column 7). Figure 4.4 demonstrates



(a) (b) (c)

Figure 4.4: Different flows from OpenSPARC T2 SoC. (a) Mondo interrupt flow. (b) NCU upstream flow. (c) PIO read flow.

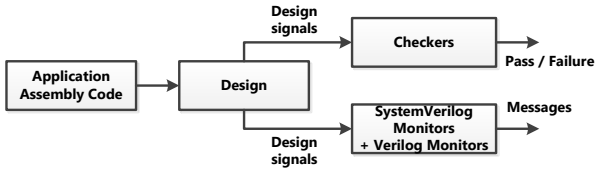


Figure 4.5: Experimental setup to convert design signals to flow messages.

a few of the OpenSPARC T2 flows. We also use the USB design [154] to compare with other methods that cannot scale to the T2.

Testbenches: We used five different tests from `fc1_all_T2` regression environment (c.f., Table 4.2). Each test exercises two or more IPs and associated flows. We monitored message communication across participating IPs during simulation and recorded the messages into an output trace file. We use System-Verilog monitors shown in Figure 4.5 to convert the RTL signals of OpenSPARC T2 into flow messages during execution for our large-scale debugging effort.

Bug injection: We created five different buggy versions of T2, that we analyze as five different case studies. Each case study comprises five different IPs. We injected a total of 14 different bugs across (c.f., Table 4.3) the five IPs in each case. The injected bugs follow two sources, i) sanitized examples of communication bugs received from our industrial partners, ii) "bug model" developed at Stanford University in the QED [112, 113, 114, 115] project capturing commonly occurring bugs in an SoC design. Table 4.3 shows that

Table 4.1: Usage scenarios and participating flows in T2. **PIOR:** PIO read flow. **PIOW:** PIO write flow. **NCUU:** NCU upstream flow. **NCUD:** NCU downstream flow. **Mon:** Mondo interrupt flow. ✓indicates Scenario i executes a flow j and ✗ indicates Scenario i does not execute a flow j. Flows are annotated with (No of flow states, No of messages).

Usage Scenario	**Participating Flows**					**Patici-pating IPs**	**Potential Root Causes**
	PIOR (6, 5)	PIOW (3, 2)	NCUU (4, 3)	NCUD (3, 2)	Mon (6, 5)		
Scenario 1	✓	✓	✗	✗	✓	NCU, DMU, SIU	9
Scenario 2	✗	✗	✓	✓	✓	NCU, MCU, CCX	8
Scenario 3	✓	✓	✓	✓	✗	NCU, MCU, DMU, SIU	9

the set of injected bugs are realistic. Table 4.4 shows tracing statistics of the usage-scenario executions. The tracing statistics implies that the injected bugs are complex and subtle. It took *up to 457* observed messages and *up to 21,290,999* clock cycles for each bug symptom to manifest. These demonstrate complexity and subtlety of the injected bugs. Following [156, 157] and Table 4.3, we have identified several potential architectural causes that can cause an execution of a usage scenario to fail. Column 8 of Table 4.1 shows number of potential root causes per usage scenario.

4.6 Experimental results

In this section, we provide details of our large-scale effort to debug five different (buggy) case studies across three usage scenarios of the OpenSPARC T2 SoC.

4.6.1 Flow specification coverage and trace buffer utilization

Table 4.5 demonstrates the value of the traced messages with respect to flow specification coverage (Definition 13) and trace buffer utilization. These are the two objectives for which our message selection is optimized. Messages selected **without packing** achieve *up to 93.75%* of trace buffer utilization

Table 4.2: Simulation testbench details.

Test Bench	Primary Objective of Testbench
tb1	Generate on-chip Mondo interrupt from PCI Express by injecting an error in memory management unit (MMU) of DMU, send PIO read and write request to IO
tb2	Generate on-chip Mondo interrupt using message signal interrupt (MSI), upstream and downstream memory request and NCU ASI register access
tb3	Upstream and downstream memory requests
tb4	Upstream and downstream memory requests, PIO read and write request to IO
tb5	Mondo interrupt generation in PCI express unit, PIO read and write request to IO
tb6	Mondo interrupt generation by sending a malformed MSI to the IMU of NCU, PIO read and write request to IO

with *up to 97.22%* flow specification coverage. **With packing**, message selection achieves *up to 100%* of trace buffer utilization and *up to 99.86%* flow specification coverage. This shows that we can cover most of the desired functionality while utilizing the trace buffer maximally.

4.6.2 Path localization during debug of traced messages

In this experiment, we use buggy executions and traced messages to show the extent of path localization per bug. Localization is calculated as the fraction of total paths of the interleaved flow. In Table 4.5, columns 7 and 8 show the extent of path localization. **Without packing**, we needed to explore *no more than 6.11%* of interleaved flow paths using our selected messages. **With packing**, we needed to explore *no more than 0.31%* of the total interleaved flow paths during debugging. Even with packing, subtle bugs like NCU bug of buggy design 2 and buggy design 3 needed more paths to explore.

4.6.3 Validity of information gain as message selection metric

We select messages per usage scenario. In Figure 4.6 we analyze the correlation between flow specification coverage and the mutual information gain

Table 4.3: Representative bugs injected in IP blocks of OpenSPARC T2. *Bug depth* indicates the hierarchical depth of an IP block from the top. *Bug type* is the functional implication of a bug.

Bug ID	**Bug Depth**	**Bug Category**	**Bug Type**	**Buggy IP**
1	4	Control	Wrong command generation by data misinterpretation	DMU
2	4	Data	Data corruption by wrong address generation	DMU
3	3	Control	Wrong construction of Unit Control Block resulting in malformed request	DMU
4	4	Control	Generating wrong request due to incorrect decoding of request packet from CPU buffer	NCU
5	2	Control	Wrong request ID construction from memory controller to L2 cache	MCU
6	3	Control	Malformed vector for memory controller availability	NCU
7	3	Control	Misclassifying interrupt thereby generating wrong interrupt acknowledgement	NCU
8	3	Control	Selecting wrong FIFO to service interrupt request	CCX
9	4	Control	Wrong construction of clock domain crossing interrupt request packet	NCU
10	4	Data	Wrong qualification of uncorrectable error in peripheral data	NCU
11	3	Data	Generation of wrong address for read/write data	MCU
12	4	Control	Wrong encapsulation of function to tag and track packet transaction in internal data pipeline by wrong decoding of transaction type	DMU
13	5	Control	Wrong interrupt signal generation for PCI interrupts	DMU
14	3	Control	Wrong address generation to read data and also affecting read request validation	MCU

Table 4.4: Tracing statistics. **NoM**: Number of observed messages between sensitized bug location and observed symptom. **NoC**: Number of cycles between sensitized bug location and observed symptom.

Case study	**Usage Scenario**	**Symptom**	**NoM**	**NoC**	**Diagn-sed buggy IP**	**Actual buggy IPs**
1	Scenario 1	*FAIL: Bad Trap*	60	13647749	DMU	DMU, NCU
2			176	329250	NCU	NCU, CCX
3	Scenario 2	*FAIL: All Threads No Activity*	164	19701000	NCU	NCU, MCU
4			457	21290999	NCU	DMU, NCU
5	Scenario 3	*GLOBAL TimeOut*	65	18624749	MCU	MCU

of the selected messages. Flow specification coverage (c.f., Definition 13) *increases monotonically with the mutual information gain* over the interleaved flow of the corresponding usage scenario. This establishes that **increase in mutual information gain corresponds to higher coverage of flow specification**, indicating that mutual information gain is a good metric for message selection.

Table 4.5: Trace buffer utilization flow specification coverage and path localization of traced messages for 3 different usage scenarios. **FSP Cov**: Flow specification coverage (Definition 13). WP: With packing. WoP: Without packing. 32 bits wide trace buffer assumed.

Case study	**Usage Scenario**	**Trace Buffer Utilization**		**FSP Cov**		**Path Localization**	
		WP	WoP	WP	WoP	WP	WoP
1	Scenario 1	96.88%	84.37%	99.86%	97.22%	0.13%	3.23%
2						0.31%	6.11%
3	Scenario 2	100%	71.87%	99.69%	93.75%	0.26%	5.13%
4						0.10%	2.47%
5	Scenario 3	100%	93.75%	83.33%	77.78%	0.11%	2.65%

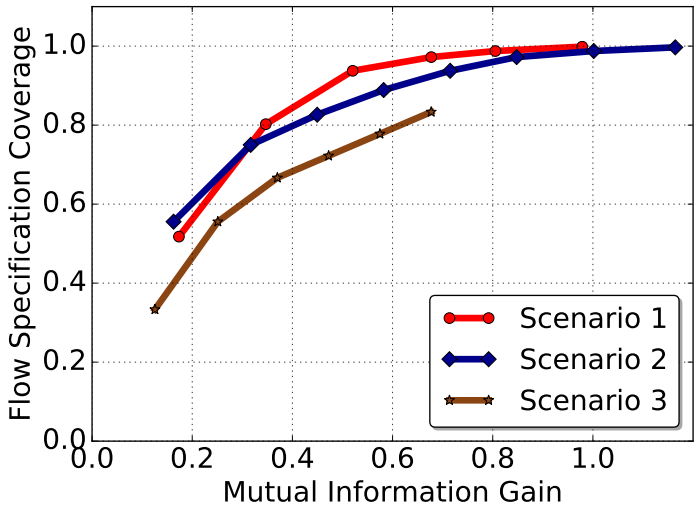


Figure 4.6: Correlation analysis between *mutual information gain* and *flow specification coverage* for different message combinations for three different usage scenarios.

Table 4.6: Comparison of signals selected by our method with those selected by SigSeT_1 [55] and PRoN [63] for the USB design. **P**: Partial bit.

Signal name	**USB module**	**SigSeT**	**PRoN**	**Info gain**
rx_data	UTMI line speed	✗	✓	✓
rx_valid		✗	✓	✓
rx_active		✗	✓	✓
rx_err		✗	✓	✓
rx_data _valid	Packet decoder	✗	✗	✓
token _valid		✗	✗	✓
rx_data _done		✗	✗	✓
idma _done	Internal DMA	✓	✗	✓
tx_data	Packet assembler	✗	✗	✓
tx_valid		✗	✓	✓
tx_valid _last		✗	✗	✓
tx_first		✗	✗	✓
send_token	Protocol engine	✗	✗	✓
token_pid _sel		**P**	**P**	✓
data_pid _sel		**P**	✗	✓

Table 4.7: Selection of important messages by our method.

Message	Affecting Bug IDs	Bug coverage	Message importance	Selected	
				Y / N	Usage scenario
m1	8, 33, 36	0.21	4.76	Y	1, 2
m2	8, 33, 34, 36	0.28	3.57	Y	1, 2
m3	33, 36	0.14	7.14	Y	1, 2
m4	8, 29, 33	0.21	4.76	Y	1, 3
m5	18, 33	0.14	7.14	Y	1, 2
m6	-	-		N	-
m7	-	-		Y	1, 3
m8	33	0.07	14.28	Y	2
m9	1, 33	0.14	7.14	N	-
m10	24	0.07	14.28	Y	2
m11	1, 24	0.14	7.14	Y	2
m12	24	0.07	14.28	Y	2
m13	8	0.07	14.28	Y	2
m14	1, 17, 33	0.21	4.76	Y	2
m15	1, 17, 18, 33	0.28	3.57	N	-
m16	1, 17, 18, 33	0.28	3.57	Y	2, 3

4.6.4 Comparison of our method to existing signal selection methods

To demonstrate that existing Register Transfer Level signal selection methods cannot select messages in system level flows, we compare our approach with an SRR-based method [55] and a PageRank based method [63]. **We could not apply existing SRR based methods on the OpenSPARC T2, since these methods are unable to scale. We use a smaller USB design for comparison with our method.** In the USB [154] design we consider a usage scenario consisting of two flows. Table 4.6 shows that our (mutual information gain based) method selects all of `token_pid_sel`, `data_pid_sel` and other important interface signals for system level debugging. SigSeT, on the other hand selects signals which are not useful for system level debugging. Our messages are composed of interface signals, and achieve a flow specification coverage of *93.65%*, whereas messages composed of interface signals selected by SigSeT and PageRank-based method have a low flow specification coverage of *9%* and *23.80%* respectively.

Table 4.8: Diagnosed root causes and debugging statistics for our case studies on OpenSPARC T2. **Time**: Time needed to manually debug a case study using traced messages.

Case Study ID	**Flows**	**Legal IP Pairs**	**Legal IP pairs investi-gated**	**Messages investi-gated**	**Time (in hours)**	**Root caused architecture level function**
1	3	12	5	25	8	An interrupt was never generated by DMU because of wrong interrupt generation logic
2	3		6	67	3	Wrong interrupt decoding logic in NCU / corrupted interrupt handling table in NCU
3	3	10	8	142	14	Malformed CPU request from cache crossbar to NCU / erroneous CPU request decoding logic of NCU
4	3		6	199	6	Erroneous interrupt dequeue logic after interrupt was serviced
5	4	12	5	65	6	Erroneous decoding logic of CPU requests in memory controller

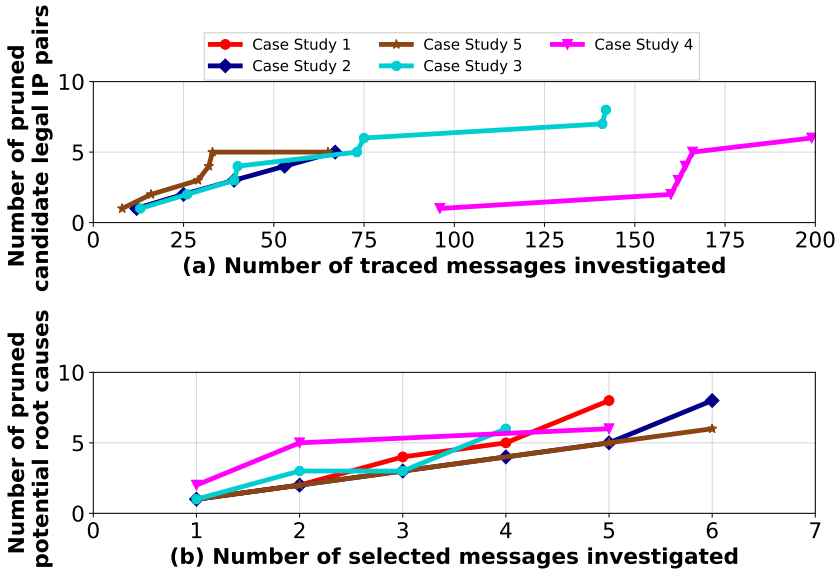
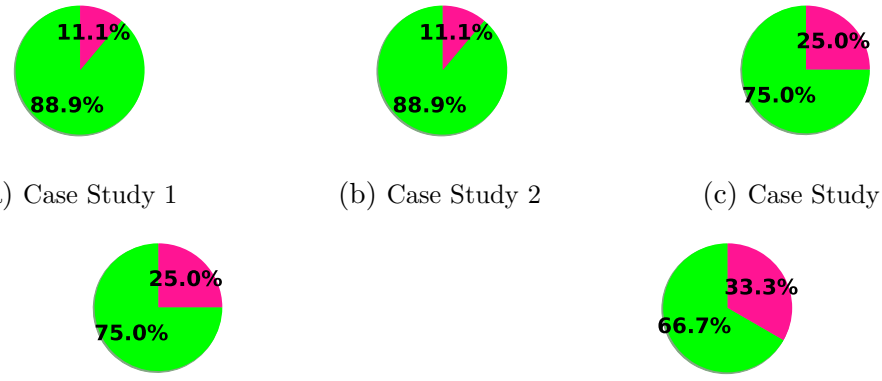


Figure 4.7: Root causing buggy IP.

4.6.5 Selection of important messages by our method

For evaluation purposes, we use *bug coverage* as a metric, to determine which messages are important. A message is said to be *affected* by a bug if its value in an execution of the buggy design differs from its value in an execution of the bug free design. Intuitively, if multiple bugs are affecting a message, it is highly likely that message is a part of multiple design paths. The *bug coverage* of a message is defined as the total number of bugs that affects a message, expressed as a fraction of the total number of injected bugs. From debugging perspective, a message is *important* if it is affected by very few bugs implying that the message symptomizes subtle bugs. Table 4.7 confirms that post-silicon bugs are subtle and tend to affect no more than four messages each. Column 4, 5 and 6 of Table 4.7 show that our method was able to select important messages from the interleaved flow to debug subtle bugs.

Table 4.7 shows that message *m15* is affected by four bugs and message *m9* is affected by two bugs, but due to their size being wider than 32 bits trace buffer, our method does not select them.



(a) Case Study 1 (b) Case Study 2 (c) Case Study 3

(d) Case Study 4 (e) Case Study 5

Figure 4.8: Selected messages-cause pruning distribution for diagnosis. ■ Plausible Cause, ■ Pruned Cause.

4.6.6 Effectiveness of selected messages in debugging usage scenarios

Every message is sourced by an IP and reaches a destination IP. Bugs are injected into specific IPs (Table 4.3). During debugging, sequences of IPs are explored from the point a bug symptom is observed, to find the buggy IP. An IP pair (⟨source IP, destination IP⟩) is *legal* if a message is passed between them. We use the number of legal IP pairs investigated during debugging as a metric for selected messages. Table 4.8 shows that we investigated *an average of 54.67%* of the total legal IP pairs, implying that our selected messages help us focus on a fraction of the legal IP pairs.

To debug a buggy execution, we start with the traced message in which a bug symptom is observed and backtrack to other traced messages. The choice of which traced message to investigate is pseudo-random and guided by the participating flows.

Figure 4.7(a) plots the number of such investigated traced messages and the corresponding candidate legal IP pairs that are eliminated with each traced message. Figure 4.7(b) shows a similar relationship between the traced messages and the candidate root causes, *i.e.*, the architecture level functions that might have caused the bug to manifest in the traced messages. Both graphs show that with more traced messages, more candidate legal IP pairs as well as candidate root causes are progressively eliminated. This implies that every one of our traced messages contributes to the debugging process.

Table 4.9: Representative potential root causes for one case study. Remaining case studies are available in [164].

Selected Messages	Potential Causes	Potential Implication
`reqtot,grant, mondoacknack, siincu, piowcrd dmusiidata. cputhreadid`	**1.** Mondo request forwarded from DMU to SIU's bypass queue instead of ordered queue	**1.** Mondo interrupt not serviced
	2. Invalid Mondo payload forwarded to NCU from DMU via SIU	**2.** Interrupt assigned to wrong CPU ID and Thread ID
`siincu,`	**3.** Non-generation of Mondo interrupt by DMU	**3.** Computing thread fetches operand from wrong memory location

Figure 4.8 shows that traced messages were able to prune out a large number of potential root causes in all five case studies. Our traced messages pruned out *up to 88.89%* (*on an average of 78.89%*) of candidate root causes.

4.7 Qualitative debugging case study on effectiveness of our message selection methodology

It is illuminating to understand the debugging process for one case study to appreciate the role of the selected messages.

Symptom: In this experiment we used traced messages from Table 4.9. The simulation failed with an error message *FAIL: Bad Trap*.

Debugging with selected messages: We consider bug symptom causes of Table 4.9 to debug this case. From the observed trace messages, `siincu` and `piowcrd`, we identify NCU got back correct credit ID at the end of the PIO read and PIO write operation respectively. This rules out two causes out of nine. However, we cannot rule out causes related to PIO payload since a wrong payload may cause computing thread to catch *BAD Trap* by requesting operand from wrong memory location. Absence of trace messages `mondoacknack` and `reqtot` implies that NCU did not service any Mondo interrupt request and SIU did not request a Mondo payload transfer to NCU respectively. Further, there is no message corresponding to `dmusiidata.cputhreadid` in the trace file, implying that DMU was never able to generate a Mondo interrupt request for NCU to process. This rules out all causes except cause **3** (**1 cause out of 9, pruning of 88.89% of possible causes**) to explore further to find the root cause.

Root Cause: From [156, 157], we note that an interrupt is generated only when DMU has credit and all previous DMA reads are done. We found no prior DMA read messages and DMU had all its credit available. Absence of `dmusiidata` message correct CPUID and ThreadID implies that DMU never generated a Mondo interrupt request. This makes DMU a plausible location of the root cause of the bug. It took *approximately eight hours* to manually diagnose this post-silicon failure using traced messages. The diagnosis time includes understanding the flow specifications from OpenSPARC T2 manual, identifying different message interleaving, and identifying message interleaving that is infrequent and deviant from other message interleavings as a diagnosed symptom of the failure.

4.8 Conclusion

We have developed a system-level message selection methodology for SoC post-silicon use-case debugging. Our approach exploits various architectural collateral (*e.g.*, messages, transaction flows) and targets typical usage scenarios exercised during post-silicon debugging. We demonstrate the scalability of our method on the OpenSPARC T2 SoC, and show through quantitative metrics and qualitative analyses, the value of the selected messages in real-world root cause analysis. Furthermore, we showed that existing signal selection methods are not suitable for trace message selection at the system level. To the best of our knowledge, this is the most large-scale application of a hardware tracing approach in published literature, arguing for the practical viability and value to the complex post-silicon debugging process.

CHAPTER 5

FEATURE ENGINEERING FOR SCALABLE APPLICATION LEVEL POST-SILICON DEBUGGING

5.1 Introduction

The post-silicon debug and diagnosis problem is convoluted by the heterogeneous IPs and vertically integrated SoC components. Concurrent execution of multiple flows in different use-cases, extremely long execution traces (potentially spanning over millions of clock cycles), lack of bug reproducibility, and lack of error sequentiality lead to a mostly manual, ad hoc, unsystematic, and extremely time-consuming post-silicon debugging effort in the industry. In our debugging case studies of Section 4.7, it took us *up to 14 hours* (*average 7.4 hours*) (c.f., Table 4.8) to debug each of the case studies.

In this chapter, we endeavor to automate post-silicon use-case debugging by analyzing the intrinsic characteristics of the input trace data without a priori design knowledge (c.f., Problem PR3 of Figure 1.10) such that the diagnosis time is shortened. We consider the traced messages (c.f., Chapter 4) as the input data to the diagnosis problem.

The primary objective of the manual post-silicon debug and diagnosis (c.f., Section 4.7) is to understand the desired behavior from the specification, to identify the correct message interleaving as per the specification, and to identify one or more message interleaving that are deviant from the specifications. Machine learning [165, 166] is a systematic study of algorithms that automatically learn/build a mathematical/statistical model from a large amount of sample data, commonly known as training data examples. Intuitively, machine learning algorithms can substitute human and can learn a model of the *correct* and *buggy* execution using a large amount of post-silicon trace data as training data examples that is generated during use-case validation. The primary challenge of applying machine learning is to construct a representation of input trace data such that the statistical model demarcates

the correct and buggy behavior clearly based on the data features without requiring a priori knowledge of the design.

A buggy design behavior can be considered as a *corner-case* design behavior. A corner-case behavior is *infrequent* and *deviant* from normal design behaviors. In machine learning parlance, *outlier detection* [167, 168] is the technique to identify *infrequent* and *deviant* data points and such infrequent and deviant data points are called *outliers* whereas normal data points are called *inliers*. Hence, if we can map normal design behavior as inliers and buggy design behavior as outliers in the machine learning data space, then we can use outlier detection techniques to automatically diagnose post-silicon failures. Consequently, the task of learning a buggy design behavior transforms into a task of modeling the buggy design behavior as an outlier.

In post-silicon execution, a failure happens due to the occurrence of one or more patterns of consecutive messages that are symptomatic of one or more design bugs. We call such a message pattern as an *anomalous message sequence*. The trace message data has several features *e.g.*, the cycle of occurrence of a message, the IP interface at which message has happened, and the message itself. We call these features as raw features of trace data. Since a buggy design behavior is a corner-case design behavior, it can be considered as an outlier in the post-silicon data space. We endeavored to characterize trace data for anomalousness using raw features. Our investigation found that raw features are insufficient to characterize anomalousness of trace messages for outlier detection.

Hence we engineer domain specific features that are highly relevant to the diagnosis task to control the normal and buggy behavior model as seen by the outlier detection algorithms. The engineered features are generic, *i.e.*, they are transformations that can be applied to any hardware designs. We use those engineered features to map buggy behavior in the raw feature space as outliers in the engineered feature space. Our engineered features capture both infrequency and deviancy of a buggy design behavior with respect to the normal design behaviors. Since anomalous message sequences represent a deviant design behavior, we use our engineered features to map such anomalous message sequences as outlier data points in engineered feature space. To make computation *tractable*, instead of analyzing each of the individual message sequences, we pre-process trace messages to create message aggregates of message sequences and characterize each such aggregates for anomaly.

A message aggregate with infrequent message sequences contains more information than [163, 169, 170] a message aggregate with frequent message sequences. We use *entropy* to quantify the information content of a message aggregate. As the number of infrequent messages sequences in a message aggregate increases, the entropy of the message aggregate increases monotonically. In order to quantify deviancy of a message sequence with respect to other message sequences in the aggregate, we use a *string similarity metric*,[1] in particular *Levenshtein distance* [171]. As an aggregate contains more and more deviant message sequences, the average pairwise Levenshtein distance of the aggregate increases monotonically. We identify message aggregates with both high entropy and high Levenshtein distance as outliers and report them as candidate root causes.

We apply off-the-shelf outlier detection algorithms to the engineered feature space spanning over entropy and Levenshtein distance. In the engineered feature space, message aggregates that represent normal behavior will be very close to each other and densely distributed whereas message aggregates that represent anomalous behavior will be sparsely distributed and distant from normal message aggregates.

The primary benefits of this diagnosis solution are – i) the proposed method automatically learns the normal and anomalous design behaviors from trace message data without training. Consequently, it helps to identify candidate anomalous message sequences without an in-depth understanding of the design, ii) the engineered features are generic and are independent of any particular design and/or application, and iii) the proposed method can shift through a large amount of trace data, thereby improving detection of candidate anomalous message sequences that are symptomatic of design bugs.

To show scalability and effectiveness of our automated diagnosis approach, we perform our experiments on OpenSPARC T2 SoC [156, 157]. We reuse the five different buggy versions of OpenSPARC T2 design that we created in Chapter 4 (c.f., Section 4.5). Our analysis shows that the proposed diagnosis method is computationally efficient. It incurred runtime of *up to 44.3 seconds* and peak memory usage of *up to 508.7 MB* to pre-process trace messages to create aggregates. To detect outlier message aggregates, it incurred runtime of *up to 18.91 seconds* and peak memory usage of *up to 508.2 MB*.

[1] A string similarity metric measures pairwise similarity of two strings.

We also evaluated effectiveness of our engineered features for outlier detection. We found that each of the candidate anomalous message aggregates has entropy of *up to 4.3482* and Levenshtein distance of *up to 3.0*. This shows that our engineered features are highly effective in demarcating anomalous message aggregates from normal aggregates.

We analyzed improvement in bug diagnosis while using automated diagnosis method as compared to manual debugging. We found that the proposed diagnosis method was able to root-cause *up to 66.7%* more injected bugs with *up to 847*× less diagnosis time. Further, the diagnosis method achieved a high precision of *up to 0.769*. This shows that our proposed diagnosis method is effective and can expedite post-silicon debugging.

Our contributions are as follows.

- First, we pose the post-silicon bug diagnosis problem as an outlier detection problem. We propose a machine learning-based scalable and efficient technique to automatically diagnose post-silicon use-case failures. Our bug diagnosis technique learns the buggy design behavior and normal design behavior automatically from the intrinsic characteristics of the input trace data without any prior knowledge of the design.

- Second, we systematically model buggy behavior as an outlier and normal behavior as an inlier in the machine learning data space. To do so, we engineered two features that are highly relevant to the diagnosis task. The features are generic *i.e.*, they are design independent and can be applied to any hardware design. The engineered features characterize the anomalousness of a buggy behavior to tune the model of buggy behavior and normal behavior as seen by the outlier algorithms.

- We establish with empirical evidence that our bug diagnosis technique is highly effective and can diagnose many more bugs at a fraction of time with high precision as compared to manual debugging.

5.2 Preliminaries

We extend the definition of a *trace*(ρ) of an *execution* ρ (c.f., Definition 8) to define *message sequence* and *message aggregate* for diagnosis.

Definition 14 *A* **message sequence** $m(\rho)$ *of a trace(ρ) is defined as a subsequence of the trace of the execution. The* **length** k *of a message sequence* $m(\rho)$ *is defined as the number of messages contained in* $m(\rho)$*. For example, for* $trace(\rho) = \alpha_1\ \alpha_2\ \alpha_3\ \ldots\ \alpha_n$, $m(\rho) = \langle \alpha_1\ \alpha_2\ \alpha_3 \rangle$ *is a message sequence of trace(ρ) of length* $k = 3$*. Any two message sequences* $m_i(\rho)$ *and* $m_j(\rho)$ *of length* k *are* **distinct** *if* $\exists l \in [1, k], \alpha_{i,l} \neq \alpha_{j,l}$ *where* $\alpha_{i,l} \in m_i(\rho)$, $\alpha_{j,l} \in m_j(\rho)$.

Definition 15 *A* **message aggregate** $maggr(\rho)$ *of a trace(ρ) is defined as an unordered set of message sequences of length* k*. Each distinct message sequence in a message aggregate is called an* **unique message sequence** *of that message aggregate. For example,* $maggr(\rho) = \{\langle \alpha_1\ \alpha_2\ \alpha_3 \rangle, \langle \alpha_2\ \alpha_3\ \alpha_4 \rangle\}$ *is a message aggregate of length 3 message sequences of trace(ρ). Each of the* $\langle \alpha_1\ \alpha_2\ \alpha_3 \rangle$ *and* $\langle \alpha_2\ \alpha_3\ \alpha_4 \rangle$ *is an unique message sequence of* $maggr(\rho)$.

5.2.1 Levenshtein distance

The Levenshtein distance is a *string similarity metric* for measuring the *dissimilarity* between two strings. Intuitively, the Levenshtein distance between two strings is the minimum number of single-character edits *e.g.*, insertions, deletions, or substitutions, required to change one string into the other. Mathematically, the Levenshtein distance between two strings a, b (of length $|a|$ and $|b|$) $\mathcal{L}_{a,b}(|a|, |b|)$ is defined as:

$$\mathcal{L}_{a,b}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0 \\ min \begin{cases} \mathcal{L}_{a,b}(i-1,j) \\ \mathcal{L}_{a,b}(i,j-1) \\ \mathcal{L}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise} \end{cases}$$

Here $1_{(a_i \neq b_j)}$ is the *indicator function* equal to 0 when $a_i = b_j$ and equal to 1 otherwise. The $\mathcal{L}_{a,b}(i,j)$ is the distance between the first i characters of a and the first j characters of b. Please note, the first element in the minimum refers to deletion, the seconds element refers to insertion, and the third element refers to match or mismatch, depending on whether the respective symbols are the same. For brevity, we will denote $\mathcal{L}_{a,b}(|a|, |b|)$ as $\mathcal{L}(a, b)$.

Let us consider two string $A =$ '$flee$' and $B =$ '$leer$'. The $\mathcal{L}(A, B)$ is 2, since the following two edits can change A to B, and there is no way to do it with fewer than two edits.

1. $\boldsymbol{f}lee \rightarrow lee$ (deletion of $\boldsymbol{f}$ from the beginning of A).

2. $lee \rightarrow lee\boldsymbol{r}$ (insertion of $\boldsymbol{r}$ at the end of A).

The salient features of Levenshtein distance are – i) it is at least difference of the sizes of the two strings, ii) it is at most the length of the longer string, iii) it is zero *iff* the strings are equal, and iv) if the strings are of the same size, the Hamming distance [172] is an upper bound on the Levenshtein distance. In the example, $\mathcal{L}(A, B)$ is 2 but the Hamming distance is 4.

5.3 Outlier detection for post-silicon debugging

5.3.1 Outliers in machine learning

In machine learning, outliers, also known as anomalies, are defined as data samples that have characteristics or behaviors which notably deviate from our expectation [167]. There are two basic characteristics of outliers [167, 168] – i) outliers are different from the norm and the differences can be captured by the features and ii) outliers are rare comparing to normal data samples. Initially, people identify outliers to remove them as part of a data processing procedure to free machine learning algorithms from the negative influences of outliers. Presently, people are interested in detecting and analyzing outliers because outliers are commonly associated with interesting or suspicious events [167].

Despite the straightforward definition of outliers, detecting outliers is challenging. First, the expected characteristics of the normal samples or the normal regions in the data space are not easy to define. Hence, the boundary between outliers and normal samples are often not precise. Moreover, some outliers only manifest their outlierness in a new feature space that is engineered from the original feature, and the transformation from the original feature space to the appropriate new feature space can be difficult. Second, the groundtruth of the outliers is often not available and the cost of obtaining the groundtruth could be prohibitively expensive. Therefore, in many cases, outliers have to be determined in absence of the guidance of groundtruth.

Unsupervised outlier detection (UOD) algorithms are developed to identify outliers through only the patterns and intrinsic properties of the feature

space, and hence do not require any groundtruth labels. Thus, UOD algorithms offer a high degree of flexibility and find wide applicability.

5.3.2 Different notions of outliers

In the field of outlier detection, the general principle of identifying outliers is to profile the normal samples in a dataset and then determine the boundary between the normal samples and outliers. The way that the normal samples are profiled varies depending on the various notions of outliers.

Classification-based notion: Outliers can be defined by a classifier that profiles the normal samples. The underlying assumption is that a classifier can be learned in the feature space to distinguish between the normal and anomalous class [168]. The classifier would generally learn a representation of the normal samples or a boundary around the normal samples. Hence, any sample that does not fit the representation of the normal samples or stays out of the boundary of the normal samples would be considered as outliers. When the groundtruth is unavailable, the classifier can be learned in an unsupervised manner. *One-class Support Vector Machine* (OCSVM) [173, 174] is an unsupervised outlier detection method that adopts this notion of outliers. OCSVM uses SVM with a Gaussian kernel and soft margin to explicitly learn a decision boundary (hyperplane) that encompasses the majority of the data samples and allows only a small fraction of data samples to lie outside of the decision boundary; the samples that lie outside the boundary are considered as outliers.

Density-based notion: Density-based outliers are based on the assumption that the normal data samples reside in neighborhoods of high density; however, outliers reside in low-density regions [168]. There are generally two ways to define the outlierness of a data sample under the density-based notion of outliers: First, the local density of a data sample can be estimated by the distance of a data sample to its k-nearest neighbors, with larger distances indicating a lower local densities and hence larger degrees of outlierness; the distance can be the distance to the k^{th} distant neighbor or the average of distances of all k neighbors. The *k-Nearest Neighbors* (kNN) [175, 176] is an unsupervised outlier detection technique that adopts this notion of outliers directly and uses the aforementioned distance as the outlier score. Second,

the relative density of each data sample to the density of their neighbors can be estimated and used as an indication of outlierness; a normal sample has a local density that is similar to its neighbors, but an outlier's local density is lower than that of its neighbors. *Local Outlier Factor* (LOF) [177] is an unsupervised outlier detection method that identifies outliers based on the relative density of the neighborhoods. LOF first estimates local densities for each sample. Then, LOF computes the ratio of the local density of a sample to the local densities of its k neighbors to determine if the sample is an outlier.

Spectral-based notion: Spectral-based notion of outliers assumes that the difference between normal samples and outliers can be significantly more apparent when the data is embedded into a lower dimensional subspace [168]. Hence, outlier detection methods that adopts the spectral-based notion of outliers would approximate the data space using a combination or transformation of the original features that can enable the outliers to be easily identified, while capturing the variability in the data. *Principal Component Analysis* (PCA) [178] is a method that can project data into a lower dimensional space, while most of the variability of the data is captured and explained by the new dimensions. The new dimensions computed by PCA can be used to define a subspace that capture the normalcy of the data. In other words, the variability that is not captured by the new dimensions is considered as anomalous. Thus, deviations from the normal subspace indicates outlierness; the deviation can be computed by the summing the projected distances of a sample on all new dimensions. *Isolation Forest* (IForest) [179, 180] is another unsupervised outlier detection method that utilizes the spectral-based notion of outliers in the sense that it attempts to identify outliers using only a subset of the features. IForest recursively select feature and split feature values in random until samples are isolated. Since outliers are rare and they lie further away from the normal samples in the feature space, the number of splittings required to isolate an outlier is less than that of the normal samples; thus, the number of splittings to isolate a sample can be served as the outlier score for that sample.

5.3.3 Metrics of an outlier detection algorithm

Definition 16 *The precision of an outlier detection algorithm is defined as the number of true positives expressed as a fraction of total number of samples labeled as belonging to the outlier class* i.e., $Precision = \frac{t_p}{t_p+f_p}$ *where* t_p = *number of true positives,* f_p = *number of false positives.*

Definition 17 *The recall of an outlier detection algorithm is defined as the number of true positives expressed as a fraction of total number of true positives and false negatives* i.e., $Recall = \frac{t_p}{t_p+f_n}$, *where* f_n = *number of false negatives.*

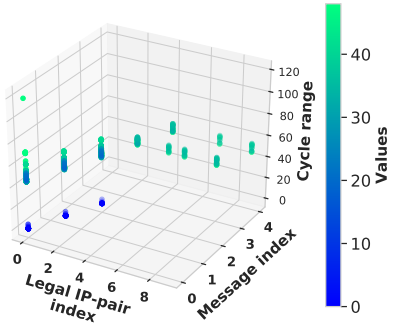
Definition 18 *The accuracy of an outlier detection algorithm is defined as the number of samples that are correctly labeled as belonging to both the outlier class and normal class expressed as a fraction of total number of samples* i.e., $Accuracy = \frac{t_p+t_n}{t_p+t_n+f_p+f_n}$, *where* t_n = *number of true negatives.*

5.4 Bug symptom diagnosis methodology

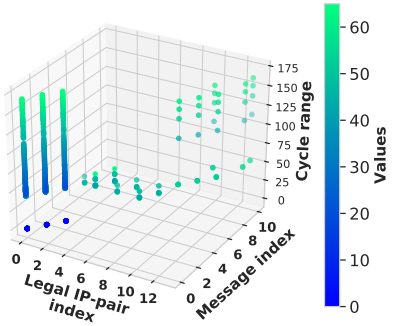
5.4.1 Formulation of post-silicon debugging as an outlier detection problem

A post-silicon execution is normal/non-anomalous if it finishes without any failures *e.g.*, hangs, deadlock, livelock, crash etc., otherwise an execution is erroneous/anomalous. For the diagnosis problem, we consider traced messages during execution as input data. In post-silicon execution, a failure happens due to the occurrence of one or more message sequence(s) that is symptomatic of one or more design bugs. We consider such a message sequence as an *anomalous message sequence*. Since an anomalous message sequence represents a deviant design behavior, we consider such a message sequence as an *outlier* in post-silicon execution data space. Consequently, we formulate post-silicon diagnosis as an outlier detection problem. **Given a set of anomalous post-silicon executions, our diagnosis method identifies one or more candidate anomalous message sequences.**

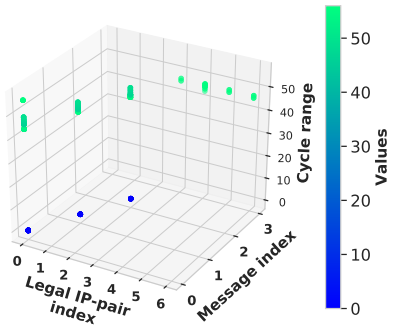
Since post-silicon execution spans millions of clock cycles, hence for tractable computation, we segregate raw trace data in multiple cycle ranges. Further,



(a) Case study 1 ($k = 5$).



(b) Case study 3 ($k = 5$).



(c) Case study 5 ($k = 5$).

Figure 5.1: (a), (b), and (c) show inability of raw feature data to demarcate anomalous message sequences.

we assign an index to every legal IP pair[2] and to every unique message that happens in a post-silicon execution.[3] The segregated trace data has three raw features – i) cycle ranges, ii) the index of a legal IP-pair, and iii) the index of a message that has occurred. In Figure 5.1 we show raw trace data in three-dimensional feature space for several case studies (c.f., Section 5.6) for OpenSPARC T2.

5.4.2 Insufficiency of raw features for diagnosis

An anomalous message sequence has two primary characteristics – i) it is *infrequent* and ii) it is *dissimilar* to other message sequences. An in-depth inspection of Figure 5.1 shows that the trace data in raw feature space has the following deficiencies – i) the raw features provide message-specific infor-

[2]An IP pair is *legal* if a message is passed between them.

[3]This index is an enumeration of traced messages and is different from indexed messages discussed in Definition 9.

mation, ii) in raw feature space outliers are not well demarcated, and iii) the raw features fail to provide context of the failure during diagnosis.

Hence, we pre-process raw trace message data to construct message sequences and characterize each such message sequences for *infrequency* and *dissimilarity* using engineered features (c.f., Section 5.3.1). To make computation *tractable*, instead of analyzing each of the message sequences individually, we analyze message aggregates of message sequences and characterize each such aggregates for the anomaly.

5.4.3 Intuition of engineered features

In order to quantify the characterization of anomalousness, we calculate two engineered feature values of each of the message aggregates – i) entropy (characterizes infrequency) and ii) Levenshtein distance (characterizes dissimilarity).

Entropy as an engineered feature: A message aggregate is characterized as anomalous if it contains one or more infrequent unique message sequences. An aggregate is considered to be more anomalous if it contains many such infrequent unique message sequences. An information theoretic way to quantify the notion of infrequency is to compute the information content of the aggregate. Entropy (c.f., Section 4.3) is one such metric that succinctly quantifies information content. An aggregate with frequent unique message sequences will have less entropy due to less information content. On the other hand, an aggregate with more and more infrequent unique message sequences will have higher entropy due to higher information content. The entropy of a message aggregate is *lower bounded* by 0.0 (when the aggregate contains exactly one unique message sequence) and is *upper bounded* by $log_2(n)$ (when the aggregate contains exactly one of each of the n unique message sequences).

Levenshtein distance as an engineered feature: Entropy fails to characterize the specific relationship that exists between individual unique message sequences of a message aggregate. Consequently, we calculate a *similarity metric*, in particular, Levenshtein distance (c.f., Section 5.2.1) to quantify the dissimilarity of the constituent message sequences in a message aggregate. If a message aggregate contains *similar* unique message sequences, the dissimilarity score will be small whereas if the message aggregate contains

Table 5.1: Definition of anomalies using engineered features entropy and Levenshtein distance. **Ldist**: Levenshtein distance. ✓: Non-anomalous message aggregate. ✗: Anomalous message aggregate.

Ldist \ Entropy	Low	High
Low	✓	✓
High	✓	✗

dissimilar unique message sequences, the dissimilarity score will be large. A message aggregate with higher Levenshtein distance will likely to be more anomalous as compared to another message aggregate with smaller Levenshtein distance. Levenshtein distance of a message aggregate is *lower bounded* by 0.0 (when the aggregate contains exactly one unique message sequence) and is *upper bounded* by the average of Hamming distance [172] of pairwise unique message sequences (when the aggregate contains n different unique message sequences).

Let us consider aggregates **A1**: {'aba', 'bab'} and **A2**: {'aba', 'cdc'} where a, b, c, d are messages. For each of the **A1** and **A2**, the entropy is $log_2(2) = 1$. Although **A2** comprises dissimilar unique message sequences as compared to **A1**, entropy alone fails to capture that dissimilarity. Hence we calculate the Levenshtein distance of each of the aggregates to quantify the dissimilarity of the constituent messages. For **A1**, $\mathcal{L}$('aba', 'bab') = 2 (1 deletion and 1 insertion) and for **A2**, $\mathcal{L}$('aba', 'cdc') = 3 (3 substitutions). Clearly, in spite of having same entropy, Levenshtein distance helped to identify **A2** to be more anomalous than **A1**.

In our diagnosis solution, we define a **message aggregate as anomalous** (*i.e.*, contains anomalous unique message sequences) that has **both high entropy and high Levenshtein distance**. Table 5.1 summarizes our definition of anomalousness of a message aggregate.

Usage of outlier detection algorithms: We apply outlier detection algorithms to the engineered feature data space spanning over entropy and Levenshtein distance. In the engineered feature space, message aggregates that represents normal behavior will be very close to each other and will form a dense cluster. On the other hand, message aggregates that represents anomalous behavior will be sparsely distributed and distant from the normal message aggregates. Outlier algorithms output a ranked list of anomalous

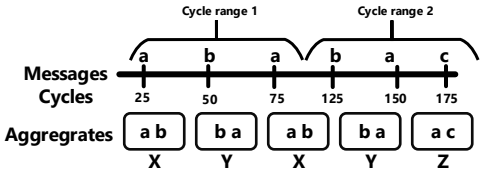


Figure 5.2: Example execution trace and a set of message sequences of length $k = 2$ and granularity $g = 100$ cycles.

message aggregates ranked by outlier scores. We output message sequences contained in top-five anomalous message aggregates as candidate anomalies.

5.4.4 Example for generating engineered feature values from raw feature values

We use an example trace of Figure 5.2 to explain the steps for generating engineered feature values. This methodology is parameterized by i) the length k of the message sequence for which anomaly needs to be detected and ii) the granularity g in number of cycles at which message aggregates need to be created. For this example, we use $k = 2$ and $g = 100$.

Step 1 (Creation of message aggregates): We use a sliding window of length k to create a set of k-length message sequences. The set of message sequences are partitioned into message aggregates based on granularity g. In the example, the set of two-length message sequences is S= $\{\underbrace{ab}_{X}, \underbrace{ba}_{Y}, \underbrace{ab}_{X}, \underbrace{ba}_{Y}, \underbrace{ac}_{Z}\}$. We partition S at a granularity of 100 cycles which creates two message aggregates $s_1 = \{X, Y, X\}$ and $s_2 = \{X, Y, Z\}$ where $X = ab, Y = ba, Z = ac$.

Step 2 (Identifying unique message sequences and their occurrences per message aggregate): We identify unique message sequences per message aggregate and calculate their number of occurrences. In the example, s_1 has two unique message sequences X and Y and s_2 has three unique message sequences X, Y, and Z. In s_1, X happened 2 times and Y happened 1 time. In s_2, each of the X, Y, and Z has happened 1 time.

Step 3 (Calculation of entropy and Levenshtein distance per message aggregate): We calculate entropy and Levenshtein distance for each

of the message aggregates using the information of unique message sequences from Step 2.

In the example, for aggregate s_1, $p(X) = 2/3$ and $p(Y) = 1/3$. Hence $\mathcal{H}(s_1) = -p(X)log_2(X) - p(Y)log_2(Y) = -2/3 * log_2(2/3) - 1/3 * log_2(1/3) = 0.9182$ and $\mathcal{L}(X, Y) = 2$, $\mathcal{L}(X, X) = 0$, and $\mathcal{L}(Y, X) = 2$. The average Levenshtein distance of aggregate s_1 is $(2 + 0 + 2)/3 = 1.33$.

Similarly, for aggregate s_2, $p(X) = 1/3$, $p(Y) = 1/3$, and $p(Z) = 1/3$. Hence $\mathcal{H}(s_2) = -p(X)log_2(X) - p(Y)log_2(Y) - p(Z)log_2(Z) = -1/3 * log_2(1/3) - 1/3 * log_2(1/3) - 1/3 * log_2(1/3) = 1.58$ and $\mathcal{L}(X, Y) = 2$, $\mathcal{L}(X, Z) = 2$, and $\mathcal{L}(Y, Z) = 2$. The average Levenshtein distance of aggregate s_2 is $(2 + 2 + 2)/3 = 2.0$.

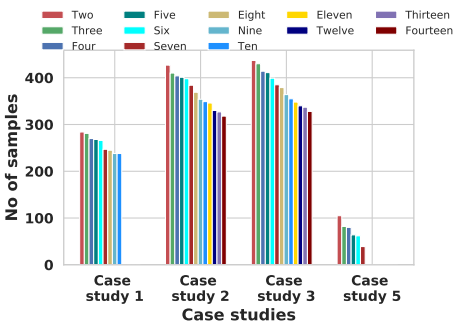
The aggregates s_1 and s_2 are represented by tuples (0.9182, 1.33) and (1.58, 2.0) respectively in engineered feature space. We input these tuples to outlier detection algorithms to detect anomalous message aggregates.

5.5 Experimental setup

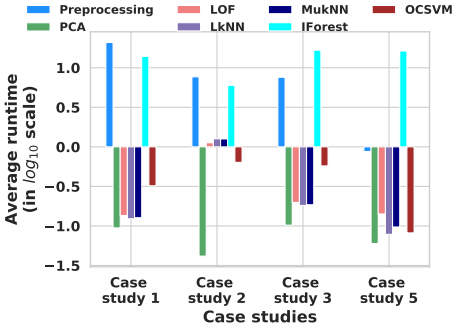
Design testbed: We use the publicly available OpenSPARC T2 SoC [156, 157] to demonstrate our diagnosis results. Figure 3.6 shows an IP level block diagram of T2. For the diagnosis experiments, we use the same set of usage scenarios shown in Table 4.1 and the same five different buggy versions of T2 SoC design that we analyze as five different case studies in Section 4.5.

Testbenches: We used 37 different tests from `fc1_all_T2` regression environment. Each test exercises two or more IPs and associated flows. We monitored message communication across participating IPs and recorded the messages into an output trace file using the System-Verilog monitor of Figure 4.5. We also record the status (passing/failing) of each of the tests.

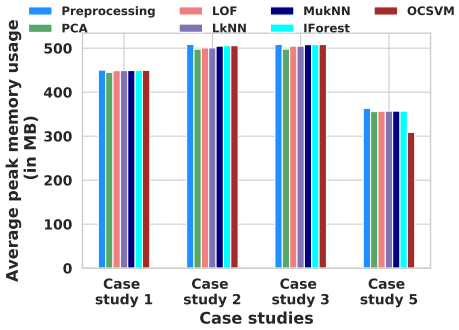
Anomaly detection techniques: We used six different outlier detection algorithms, namely IForest, PCA, LOF, LkNN (kNN with longest distance method), MukNN (kNN with mean distance method), and OCSVM from PyOD [181]. We applied each of the above outlier detection algorithms on the failure trace data generated from each of the five different case studies to diagnose anomalous message sequences that are symptomatic of each of the injected bugs in each of the case studies.



(a) Total number of samples.



(b) Runtime comparison.



(c) Peak memory usage comparison.

Figure 5.3: (a) shows total number of message aggregate samples for different length message sequences for different debugging case studies. (b) and (c) demonstrate that our diagnosis methodology is computationally efficient in terms of runtime and peak memory usage across six different outlier detection algorithms for each of the case studies.

5.6 Experimental results

In this section we provide insights into our bug diagnosis methodology to debug five different (buggy) case studies across three usage scenarios of the OpenSPARC T2 SoC. For these experiments, we have used $g = 100000$ cycles and varied k from two to number of valid IP pairs (c.f., Table 4.8) for each of the case studies. The number of message aggregate samples for different lengths of message sequences for each of the outlier detection algorithm per debugging case study is shown in Figure 5.3a.

5.6.1 Computational efforts for data preprocessing and outlier message sequence diagnosis

In this experiment, we show scalability of the automated diagnosis methodology in terms of runtime and peak memory usage. Figure 5.3b and Figure 5.3c show runtime and peak memory usage for preprocessing and outlier detection algorithms. To calculate the average runtime and average peak memory usage of each of the outlier detection algorithms, we ran each of them *20* times and calculated the average value.

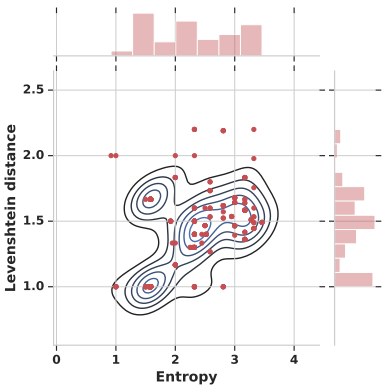
Preprocessing trace message data to create message sequence aggregates incurred a runtime of *up to 44.3 seconds* (*average 10.8 seconds*) and peak memory usage of *up to 508.7 MB* (*average 457.73 MB*). To run each of the outlier detection algorithms on the processed message aggregates incurred only *up to 18.91 seconds* (*average 2.77 seconds*) and peak memory usage of *up to 508.2 MB* (*average 451.27 MB*). Since preprocessing has *up to 443*× (*average 3*×) more runtime than the running each of the outlier detection algorithms, we showed runtime in the log_{10} scale in the Figure 5.3b.

This experiment shows that our preprocessing and diagnosis is computationally efficient.

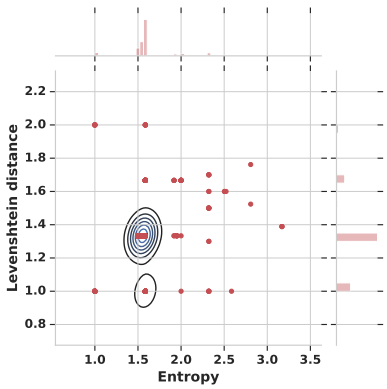
5.6.2 Validity of entropy and Levenshtein distance as engineered feature for outlier message sequence diagnosis

In this experiment, we analyze the effectiveness of *entropy* and *Levenshtein distance* to identify message aggregates that contain anomalous message sequences. In Figure 5.4 we show joint probability distribution of entropy and Levenshtein distance and in Figure 5.5 we show minimum, maximum, and average of entropy and Levenshtein distance of anomalous message aggregates across different length message sequences for three different debugging case studies.

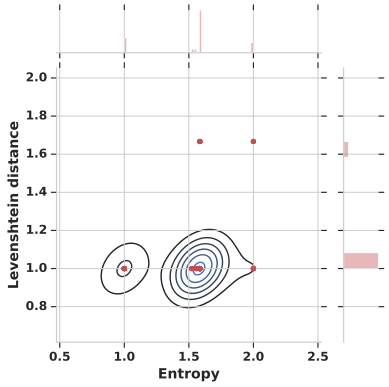
As shown in Figure 5.4, in the engineered feature space, message aggregates for normal behavior form a dense cluster whereas anomalous message sequences are sparsely distributed and are placed at a distance from the normal message aggregates. Further, Figure 5.5 shows that message aggregates that contain anomalous message sequences have entropy of *up to 4.3482* (*av-*



(a) Case study 1 ($k = 5$)

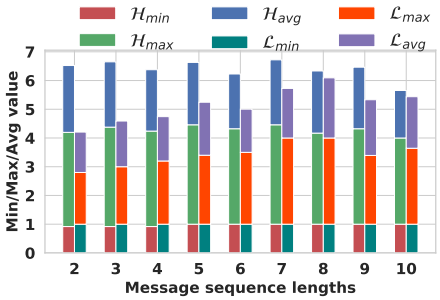


(b) Case study 3 ($k = 5$)

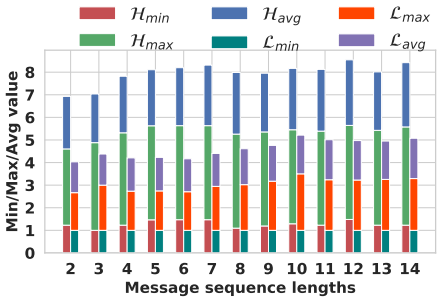


(c) Case study 5 ($k = 5$)

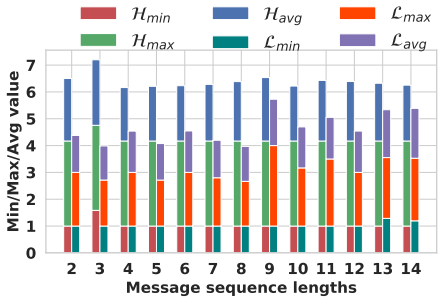
Figure 5.4: (a), (b), and (c) show that the engineered features demarcate normal and anomalous message aggregates.



(a) Case study 1



(b) Case study 2



(c) Case study 3

Figure 5.5: (a), (b), and (c) show that the minimum, maximum, and average value of engineered features are high for anomalous message aggregates irrespective of message sequence lengths. $\langle \mathcal{H}_{min}, \mathcal{H}_{max}, \mathcal{H}_{avg} \rangle$: Minimum, maximum, average entropy. $\langle \mathcal{L}_{min}, \mathcal{L}_{max}, \mathcal{L}_{avg} \rangle$: Minimum, maximum, average Levenshtein distance.

erage 2.08) and Levenshtein distance of *up to 3.0* (*average 1.5734*).

This experiment validates that entropy and Levenshtein distance are valuable and effective engineered features in demarcating the anomalous message aggregates from normal message aggregates.

5.6.3 Agreements among different outlier detection algorithms in detecting outlier message sequences

In this experiment, we assess the extent of agreement between anomalies identified by various outlier algorithms (c.f., Section 5.3). Since this set of algorithms uses different methods for outlier detection, we surmise that the confidence in an anomalous message aggregate is higher, if multiple outlier detection algorithms identify it as such. For this analysis, we consider the

Table 5.2: Diagnosis statistics for different outlier detection algorithms for different case studies using OpenSPARC T2 SoC [156, 157]. **IForest**: Isolation Forest algorithm [179, 180]. **PCA**: Principal Component Analysis [178]. **LOF**: Local Outlier Factor based algorithm [177]. **D**: Fraction of injected bugs diagnosed by an outlier detection algorithm. f_p: Total number of false positive message sequences (no more than 37% anomalous message sequences). **P**: Precision of an outlier detection algorithm. **OS**: Overall diagnosis statistics for each of the outlier detection algorithm per debugging case study.

Case Study	**IForest**				**PCA**				**LOF**			
	D	t_p	f_p	**P**	**D**	t_p	f_p	**P**	**D**	t_p	f_p	**P**
1	0.75	9	4	0.69	0.25	7	3	0.7	0.5	2	2	0.5
2	0.67	17	10	0.63	0.34	24	9	0.73	0.34	12	1	0.92
3	0.34	6	4	0.6	0.34	6	4	0.6	0.34	4	0	1.0
4	1.0	7	3	0.7	0.34	6	4	0.6	0.34	3	2	0.6
5	1.0	8	2	0.8	1.0	8	2	0.8	1.0	8	2	0.8
OS	0.73	9.4	4.6	0.67	0.4	10.2	4.4	0.69	0.47	5.8	1.4	0.81

top 10% of anomalous message aggregates per outlier detection algorithm per case study.

Our analysis showed that six outlier detection algorithms agree for a total of six anomalous message aggregates that diagnose *13.33%* of injected bugs, five outlier detection algorithms agree for a total of 17 anomalous message aggregates that diagnose *53.33%* of injected bugs, three outlier detection algorithms agree for a total of six anomalous message aggregates that diagnose *20%* of injected bugs, two outlier detection algorithms agree for a total of six anomalous message aggregates that diagnose *26.6%* of injected bugs.

This experiment shows that our engineered features are generic to characterize anomalies such that multiple outlier detection algorithms agree on a large number of anomalies that diagnose multiple bugs. This observation motivated us to use a comprehensive anomaly score to rank message aggregates. We explain our comprehensive anomaly score calculation in Section 5.6.6.

Table 5.3: Diagnosis statistics for different outlier detection algorithms for different case studies using OpenSPARC T2 SoC [156, 157]. **LkNN**: k-Nearest Neighbor using largest distance as metric [175, 176]. **MukNN**: k-Nearest Neighbor using mean distance as a metric [175, 176]. **OCSVM**: One-class Support Vector Machine [173]. **D**: Fraction of injected bugs diagnosed by an outlier detection algorithm. f_p: Total number of false positive message sequences (no more than 37% anomalous message sequences). **P**: Precision of an outlier detection algorithm. **OS**: Overall diagnosis statistics for each of the outlier detection algorithm per debugging case study.

Case	**LkNN**				**MukNN**				**OCSVM**			
Study	**D**	t_p	f_p	**P**	**D**	t_p	f_p	**P**	**D**	t_p	f_p	**P**
1	0.25	18	4	0.82	0.75	9	4	0.69	0.75	20	6	0.77
2	0.34	24	9	0.73	0.34	12	8	0.6	0.34	24	9	0.73
3	0.67	10	3	0.77	0.67	10	3	0.77	0.34	6	4	0.6
4	0.34	9	3	0.75	0.67	9	3	0.75	0.67	8	2	0.8
5	1.0	9	3	0.75	1.0	9	3	0.75	1.0	8	2	0.8
OS	0.47	14	4.4	0.76	0.67	9.8	4.2	0.70	0.67	13.2	4.6	0.74

5.6.4 Comparison of precision of different outlier detection algorithms in detecting outlier message sequences

In this experiment, we compare the *precision* (c.f., Definition 16), *recall* (c.f., Definition 17), and *accuracy* (c.f., Definition 18) of each of the outlier detection algorithms in diagnosing anomalous messages sequences per debugging case study. In Table 5.2 and Table 5.3, we show the fraction of injected bugs diagnosed, and the number of true positive and false positive candidate anomalous message sequences identified for each of the outlier detection algorithm per debugging case study. In Table 5.4, we show the fraction of total number of injected bugs diagnosed, total number of true positive, false positive, true negative, and false negative candidate anomalous message sequences identified across all of the outlier detection algorithms per debugging case study. For this analysis, we considered only the top 10% anomalous message aggregates identified by each of the outlier detection algorithm per debugging case study.

Our analysis shows that IForest, MukNN, and OCSVM consistently performed better in anomalous message sequence diagnosis as compared to the other three algorithms PCA, LOF, and LkNN. Each of the outlier detection algorithm diagnosed *up to 100%* of injected bugs. IForest diagnosed on an *average 73%* of injected bugs with a precision of *up to 0.8* (*average 0.69*),

Table 5.4: Overall statistics of automated debugging across all outlier detection algorithms across all case studies. **D**: Fraction of injected bugs detected. **P**: Precision. **R**: Recall. **A**: Accuracy.

Case Study	**D**	**Sequences**				**P**	**R**	**A**
		t_p	t_n	f_p	f_n			
1	0.75	20	2	6	54	0.769	0.27	0.25
2	0.67	29	2	11	24	0.725	0.54	0.45
3	0.67	10	2	3	22	0.769	0.32	0.28
4	1.0	20	0	6	22	0.769	0.48	0.42
5	1.0	9	1	3	4	0.75	0.69	0.56

MukNN diagnosed on an *average 67%* of injected bugs with a precision of *up to 0.77* (*average 0.70*), and OCSVM diagnosed on an *average 67%* of injected bugs with a precision of *up to 0.8* (*average 0.74*) per debugging case study. On the other hand, PCA diagnosed on an average *average 40%* of injected bugs with a precision of *up to 0.8* (*average 0.69*), LOF diagnosed on an *average 47%* of injected bugs with a precision of *up to 1.0* (*average 0.81*), and LkNN diagnosed on an *average 47%* of injected bugs with a precision of *up to 0.82* (*average 0.76*) per debugging case study. Further analysis shows (c.f., Table 5.4) our automated diagnosis technique was able to detect *up to 100%* (*average 81.8%*) of injected bugs with a precision of *up to 0.769* (*average 0.756*) per debugging case study.

In Table 5.4, we also show the recall and the accuracy metric per debugging case study. Our diagnosis methodology achieved *up to 0.69* (*average 0.46*) recall and *up to 0.56* (*average 0.39*) accuracy. We note that in Table 5.4 the value of recall and accuracy are relatively small. This is due to the fact that we are only considering the top 10% anomalous message aggregates for this analysis. Consequently, the t_p in the numerator is calculated from those top 10% anomalous message aggregates whereas f_n and t_n are calculated based on the entire set of message aggregates. Consequently, the numerators are much smaller than the denominators (c.f., Definition 17 and Definition 18) which results in a small value of recall and accuracy.

This experiment shows that our automated diagnosis methodology using engineered features is effective in identifying complex and subtle bugs with high precision.

Table 5.5: Summary of diagnosis improvements achieved using automated diagnosis technique as compared to manual debugging. **N**: Number of candidate anomalous message sequences identified. **T**: Time taken to identify a candidate anomalous message sequence. **D**: Improvement in terms of number of additional bugs diagnosed as a fraction of injected bugs. **t**: Improvement in diagnosis time. ⊘: Not available.

Case Study	Bug ID	Manual		Automated		Improvement	
		N	T (Hrs)	N	T (Secs)	D	t
1	1	1	8	18	61.4	50%	469.1×
	28	⊘	⊘	2			
	29	⊘	⊘				
	36	⊘	⊘	⊘	⊘		
2	17	⊘	⊘	5	58.5	33.3%	184.61×
	18	1	3	24			
	25	⊘	⊘	⊘	⊘		
3	5	⊘	⊘	⊘	⊘	33.3%	847.05×
	8	1	14	6	59.5		
	37	⊘	⊘	4			
4	5	1	6	14	57.5	66.7%	375.65×
	8	⊘	⊘	3			
	37	⊘	⊘	3			
5	24	⊘	⊘	3	48.5	50%	445.36×
	39	1	6	6			

5.6.5 Improvement in diagnosis over manual debugging

In this experiment, we analyze the improvement in diagnosis in terms of number of injected bugs diagnosed and diagnosis time over manual debugging. Table 5.5 (column 7 and column 8) summarizes the diagnosis improvement. We were able to diagnose *up to 66.7% more injected bugs* (*average 46.67%*) with *up to 847*× (*average 464.35*×) less diagnosis time.

This experiment shows that our automated bug diagnosis is effective and expedites debugging.

5.6.6 Comprehensive ranking of outlier message sequences

In Section 5.6.4, our experimental results showed that IForest, OCSVM, and MukNN are the three most effective outlier detection algorithms among six for diagnosing useful anomalous message sequences that can help in debug-

ging. Each of the IForest, OCSVM, and MukNN (c.f., Section 5.3) detect anomalous message aggregates based on a different perspective. IForest selects an anomalous message aggregate based on *shorter path lengths* created by *random selection of a feature* and *recursive partitioning of the feature data*. OCSVM selects an anomalous message aggregate by solving an optimization problem to find a *maximal margin hyperplane* that best separates anomalous message aggregates. MukNN (*i.e.*, k-NN with mean distance as metric) selects an anomalous message aggregate based on a aggregate's local density and the distance to its k^{th} nearest neighbor.

Consequently, to incorporate these different perspectives into our diagnosis methodology, we use a heuristic combination of outlier scores from each of the above three algorithms for each of the message aggregate. We found that a linear combination of outlier scores of a message aggregate is in closer agreement with our empirical findings than relying on outlier score of a message aggregate from each of the individual algorithms. Let x be a message aggregate, $Ano(x)$ be the comprehensive outlier score of x, and $IForest(x)$, $OCSVM(x)$, and $MukNN(x)$ be the outlier score of x using the IForest, OCSVM, and MukNN algorithm respectively. We define $Ano(x)$ as follows.

$$Ano(x) = \frac{IForest(x) + OCSVM(x) + MukNN(x)}{3} \tag{5.1}$$

In our experiments, we rank anomalous message aggregates based on the comprehensive outlier score defined by Equation 5.1.

5.7 Qualitative debugging case study on effectiveness of our diagnosis methodology

It is illuminating to understand a case study to appreciate the effectiveness of our automated bug diagnosis methodology in the debugging process.

Symptom: In this experiment we reused traced messages from Table 4.9. The simulation failed with an error message *FAIL: Bad Trap*.

Issues with manual debugging: The manual debugging using traced messages has several drawbacks. Firstly, it relies on an *in-depth understanding of the flows* and observed messages *e.g.*, `siincu` and `piowcrd` to interpret design functionality. This itself is a manually intensive task. Secondly, it

missed diagnosing multiple bugs. In the manual debug of Section 4.7, *we were only able to detect one among four injected bugs.* This is because a) *manually analyzing a large number of message sequences is tedious and error prone* and b) it is *extremely difficult to identify infrequent and deviant message sequences* that are symptomatic of one or more bugs.

In comparison, a diagnosis technique such as the proposed one (c.f., Section 5.4) can automatically *learn and distinguish* the correct design behavior from buggy design behavior.

Debug with bug diagnosis methodology: We apply our bug diagnosis methodology on the same set of trace messages as before. The methodology identified *five* anomalous message aggregates containing a total of 26 unique message sequences. We found 20 *true positive* anomalous message sequences that are symptomatic of different bugs that we injected in the design. Among these 20 anomalous message sequences, 18 message sequences were symptomatic of the bug that we identified manually. The remaining two message sequences were symptomatic of the other two injected bugs.

Clearly, while debugging manually, we were unable to detect the later two bugs because i) they were more subtle and ii) the symptomatic message sequences were extremely infrequent. Interestingly, *the manual debug took approximately eight hours* to diagnose one symptomatic message sequence. In comparison, the automated bug diagnosis methodology took only *approximately 62 seconds* (an improvement of *469*×) to pre-process the trace messages and to diagnose candidate anomalous message sequences using different outlier detection algorithms. Additionally, the diagnosis method was able to diagnose candidate anomalous message sequences for two more bugs, an improvement of *50%* over manual debugging (c.f., Table 5.5).

This case study shows that our bug diagnosis methodology automates and expedites tedious and error-prone manual debugging process of post-silicon failures.

5.8 Conclusion

We have presented an automated post-silicon bug diagnosis methodology for SoC use-case failures. Our solution uses the power of machine learning and feature engineering to automatically learn the buggy design behavior and

the normal design behavior from the trace data by analyzing intrinsic data feature without requiring a prior knowledge of the design. Our proposed diagnosis solution is highly effective and can diagnose many more bugs at a fraction of time with high precision as compared to manual debugging. We demonstrate the effectiveness of out proposed diagnosis solution using real-world debugging case studies on the OpenSPARC T2 SoC.

CHAPTER 6

ASSERTION RANKING USING RTL SOURCE CODE ANALYSIS

6.1 Introduction

Assertions are used in a wide spectrum of validation acitivities *e.g.*, formal verification, runtime monitoring, dynamic validation, coverage analysis, to validate hardware designs throughout their life cycle. Assertion-based verification heavily depends on the quality of the assertions used. Writing good quality assertions is a very hard problem [6, 7, 45, 46, 47, 48, 49, 50, 51]. Consequently, in industry, manual inspection is required to identify high-quality assertions for verification.

In this chapter, we present a comprehensive assertion-ranking methodology (c.f., Problem PR4 of Figure 1.10) to quantify the "goodness" of an assertion using RTL source code analysis.

One can see intuitively that any behavior in a design can be considered important, if that behavior affects the visible output of the design. The more subtly it affects the output behavior, the more important it is. One way to quantify and compute this functional notion of importance, in terms of design structure, is to find variables that are highly "connected" to other important variables. Such a recursive definition would allow for iterative computation across the variable dependency graph. An important assertion for a target variable would then be one that comprises many important design variables, and captures the design path(s) that are the most important to that target (output).

However, too many important design variables in an assertion hinder its *understandability* and diminish its *practical usability*. The anticipated use cases (*e.g.*, design understanding, validation, and debugging) of our ranking method tend to have a human in the loop who needs to comprehend the design behavior captured by an assertion. In order to *balance the importance*

with *understandability*, we calculate the complexity of an assertion. The resulting assertion complexity is an attempt to quantify and compute the *human comprehensibility* of an assertion.

Intuitively, we consider the complexity of an assertion to be the number of logic levels traversed from the assignment of the variable in the consequent of an assertion to the reference of the variables in its antecedent. Our reasoning is that understandability decreases with traversal of more logic levels when source code is being inspected.

Our approach ranks important assertions that are easily understandable. We compute ranks for propositional as well as temporal assertions.

Defining a ranking scheme for assertions is an inherently subjective task, because of the versatility and varying uses of assertions. In this work, we seek to identify diverse perspectives of a design, and develop a comprehensive ranking methodology that includes these perspectives. Assertion coverage[1] or the behavior covered by an assertion over the RTL source code provides a different perspective on the importance and complexity of an assertion. Recent work [29] provides a method for computing assertion statement coverage. Although the intention there is limited to finding statement coverage in the scope (between the antecedent and consequent) of an assertion, we repurpose statement coverage as a way to provide a "goodness" metric for assertions. We first use that coverage-based ranking as a baseline against which to compare our importance / complexity-based ranking, and then we incorporate it into a comprehensive ranking for assertions.

Our empirical comparison between importance / complexity rankings and coverage-based rankings yielded the following observations. In terms of computational efficiency, *coverage-based ranking is much less efficient in ranking a set of assertions, needing up to 4366× more runtime than importance / complexity-based ranking.*

We compared *bug detection and localization* of top-ranked assertions from the importance / complexity-based ranking and the coverage-based ranking. Our analysis shows that top-ranked assertions from importance/complexity-based ranking detect *up to 1.5×* more bugs per assertion than the top-ranked

[1]This is distinct from the assertion coverage used in commercial tools as a simulation metric. That metric measures how many assertions are stimulated by a test suite and measures the goodness of the test suite. In this work, we focus on the question of how much behavior is covered by a set of properties.

assertions from the coverage-based ranking.

We also compared overlaps in ranks by using the two kinds of rankings for a set of assertions. We found that when the top 20% and bottom 20% of assertions are combined, *on average, ranks of up to 57.26% assertions agree.* This shows that there is some partial agreement between the rankings even though they capture different perspectives. That discovery motivated us to explore a comprehensive ranking scheme for assertions that incorporates coverage along with importance and complexity.

While rank aggregation [182, 183, 184] was an obvious choice for combining the two rankings, the disparities between the ranks are too wide to permit use of that approach. We present a heuristic combination of importance / complexity-based ranking and coverage-based ranking to generate a comprehensive ranking for a set of assertions. We find that a *parameterized linear combination* of importance / complexity-based ranking and coverage-based ranking is in close agreement with our empirical findings. In our analysis, we chose the correlation coefficient between the importance / complexity-based ranking and the coverage-based ranking as the parameter.

Our contributions are as follows.

- We propose a systematic technique to provide a quantitative estimate of the "goodness" of an assertion and means to compare the quality of a set of assertions. Our technique can quantify "goodness" of both manual and automatic RTL assertions.

- We define *assertion importance* and *assertion complexity* metrics to quantify the "goodness" of an assertion. We also develop an algorithm to compute those metrics.

- We also show that importance / complexity-based ranking is consistent with respect to the design functionality. We establish with empirical evidence that top-ranked assertions from importance / complexity-based ranking are valuable for design understanding, localization, and debugging.

- We demonstrate that the computational efficiency of importance / complexity-based ranking is several orders of magnitude greater than that of the only known coverage-based ranking algorithm.

- Finally, we propose a comprehensive ranking for assertions to combine the diverse perspectives of the importance / complexity-based ranking and the coverage-based ranking.

6.1.1 Use cases of assertion ranking

The use cases for our assertion-ranking approach comprise situations where assertions are used and need to be examined for some purpose by a human in the loop. Automatically generated assertions [39, 41, 42, 65] are often numerous and more assertions may be generated than can practically be examined by a human. Ranking of the most important assertions is essential if this technology is to be practicable. We further elaborate the use cases here.

An interesting use case for assertion ranking is *debugging*. Given that debugging effort is a "pain point," assertion ranking can be used to save and prioritize debugging efforts. Our case studies show that debugging with a top-ranked assertion is straightforward, whereas debugging with low-ranked assertions is convoluted and requires complex reasoning (c.f., Section 6.4). Debugging effort is greater when, from the point when a bug's symptom is observed, multiple levels of logic have to be navigated to find the root cause. Our notion of complexity captures just that. In debugging, it translates into navigating much less logic, while simultaneously covering the most important ground.

In *formal verification*, one of the factors influencing the efficiency of the formal verifier is the number of properties and the size of each property. It is important that the properties being verified be succinct, have high behavioral coverage, and do not have very high temporal depth. These qualities are ensured by the importance, complexity, and coverage metrics that we measure in our comprehensive ranking. Further, if the formal verifier runs into a capacity issue, the ranked assertion list can be used as a guideline for deciding which assertions to prioritize for formal verification.

In *simulation*, a large number of assertions result in slowing of the process. In *emulation*, assertions need to be synthesized along with the design in a small part of the die area. Top-ranked assertions from our ranking can produce high-quality assertions that can benefit simulation as well as emulation

performance.

In general, the proposed ranking technique can inform designers of the *quality of the assertions* that they have written. Our technique can provide *hints* to the designers about the important behavior that they might be missing. Further, designers can also get an understanding of what variables and statements in the design have and have not been covered, thereby providing insight into the behavior that remains to be checked.

6.2 Preliminaries

6.2.1 Assertions

Definition 19 *An **assertion** is a linear temporal logic (LTL) formula of the form* $\mathbf{P} = \mathcal{G}(A \rightarrow C)$*, where* $A = A_0 \wedge \mathcal{X}(A_1) \wedge \mathcal{X}\mathcal{X}(A_2) \wedge \ldots \wedge \mathcal{X}^m(A_m)$ *and* $C = \mathcal{X}^n(C_n)$ *and* $n \geq m$*. Here, each* A_i *is a conjunction [149] of a proposition defined in terms of the input and/or register variables of the Verilog design, and* C *is a proposition defined in terms of a given register and/or output variable. The proposition in* C *is defined as a **target variable**.* $\mathcal{X}^n(n \geq 0)$ *is equal to a delay by* n *cycles. Each proposition in each of the* A_i *or in* C *is a signal-value pair where the value can be either 0 or 1.*

A0: $(req2 == 1 \wedge gnt\_ == 1)\ \mathcal{X}\ (req1 == 1) \rightarrow (gnt1 == 1)$ is an assertion for the two-port arbiter of Figure 3.1. The proposition $gnt1$ in the consequent is the target variable.

Definition 20 *A **temporal variable** is defined as a design variable* v *whose value assignments span across multiple clock cycles with respect to the current clock cycle. Let* v^{-k} *denotes* v *at* k *clock cycles in the past relative to the current clock cycle.*[2] *Our methodology treats each such variable as a unique variable,* e.g., $v^{-1} \neq v$*. If there is a loop in the global dependency graph that involves* v*, then that loop will depend on an infinite number of temporal variables. Consequently, our methodology constructs a* relative variable dependency graph *for* v*, transitively expressing its dependencies within a bounded number of clock cycles.*

[2]We use v^{-k} or $[k]v$ interchangeably to denote v at k clock cycles in the past relative to the current clock cycle.

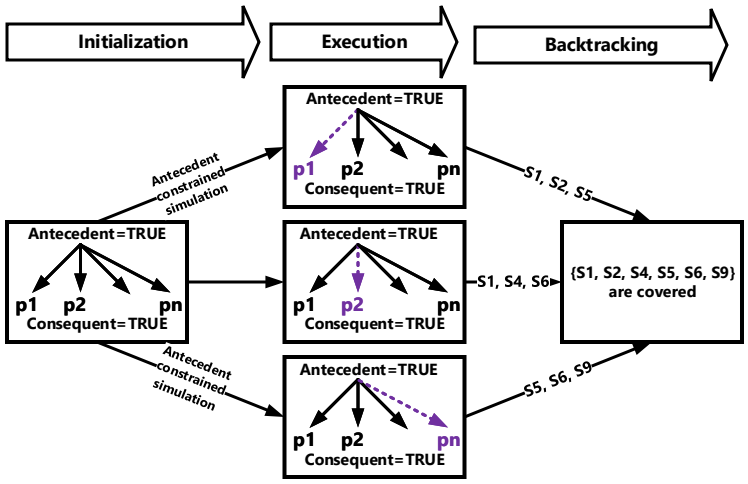


Figure 6.1: Three different phases of correctness-based statement coverage algorithm [29]. p1, p2, ..., pn are different design paths. S1, S2, S4, S5, S6, and S9 are the design statements that are covered by the assertion **P**.

In Figure 3.1, $gnt\_$ is a temporal variable, since its value assignments span multiple cycles at line 6 and line 8.

6.2.2 Statement coverage analysis

In [29] the authors have proposed a *correctness-based statement coverage* algorithm for an assertion **P**: $\mathcal{G}(A \rightarrow C)$. For a given Verilog program $\mathcal{M}$ and a *non-vacuously true* assertion **P** in that Verilog program, a statement S in the Verilog program is said to be *covered* by **P** if the following conditions hold true: i) execution of S depends on some propositional term(s) in the antecedent A, and ii) execution of S or any other statement whose execution is dependent on S makes the consequent C evaluate to true.

The approach of [29] contained three steps: *initialization*, *execution*, and *backtracking* (c.f., Figure 6.1). In the initialization phase, design variables are assigned values corresponding to the antecedent being true, and all other design variables are randomized. In the execution phase, the control data flow graph (CDFG) of the design is executed, and the statements that are executed until the consequent is evaluated are recorded. In the backtracking phase, starting from the point where the consequent was evaluated, concrete executions that were recorded before are analyzed. The set of statements

identified as dependent on the antecedent (from the concrete execution) are reported as covered. The three phases are repeated for a user-specified number of times to simulate more design execution paths. The final set of covered statements are the set union of the statements covered in each iteration.

6.2.3 Statement coverage-based assertion ranking (*SRank*)

We repurpose the algorithm of [29] (c.f., Section 6.2.2 and Figure 6.1) to rank a set of assertions for a Verilog program based on the statement coverage of an assertion.

Definition 21 *The* ***statement-coverage-based rank score*** *(**SRank**) of an assertion* $\mathbf{P}$ *is defined as the number of design statements that are covered for the non-vacuous truth of* $\mathbf{P}$ *in a Verilog program expressed as a fraction of the total number of design statements.*

Let N be the total number of statements of a Verilog program, including conditional statements, blocking and non-blocking statements, and assign statements. Let n be the total number of statements covered by an assertion $\mathbf{P}$. We compute the *statement coverage-based rank score* of $\mathbf{P}$ as $SRank(\mathbf{P}) = n/N$. The *SRank* quantifies the *statement coverage* per assertion and induces a *rank ordering* among a set of assertions. We compare the importance/complexity-based assertion ranking to that of *SRank*.

6.3 Assertion ranking methodology (*IRank*)

6.3.1 Intuition of assertion importance and assertion complexity

In this section, we discuss the intuition behind importance / complexity-based ranking to quantify the "goodness" of a set of assertions by using a systematic analysis of the RTL source code. We focus our ranking method on ranking of assertions with a *single target variable*. The ranking method targets *design verification/validation, design understanding,* and *debugging* as the potential use cases of the ranked assertions. Consequently, the ranking method ranks assertions based on their characterization of the design

functionality. To quantify the characterization of design functionality, we calculate i) *assertion importance* and ii) *assertion complexity*. Our ranking method considers both assertion importance and assertion complexity in a balanced way.

Assertion importance: A design behavior is important if it affects the visible outputs of a design. The design behavior is considered to be more important if it affects output behaviors more subtly. One way to quantify and compute the functional notion of importance in terms of design structure is to find variables that are highly "connected" to other important variables. In other words, a design variable is important if it is a part of many design paths that capture important design behaviors. That recursive definition of variable importance allows for an iterative computation across the variable dependency graph (c.f., Section 3.2.3). An *important assertion* for a *target variable* comprises many important design variables with respect to the target, and captures the design path(s) that are most important to the target.

Variable importance: To calculate assertion importance, we calculated a *global importance score* of each of the design variables in an RTL by using Google's PageRank [64, 153] (c.f., Section 3.2.1) algorithm. We applied the PageRank algorithm on an RTL variable dependency graph (c.f., Section 3.2.3). Intuitively, PageRank ranks dependency graph nodes with many incoming edges and many outgoing edges higher than those with fewer such edges.

Although the global importance score provides a global ranking of all the variables in the design, it does not capture the specific relationship that exists between the target variable and the variable(s) in the antecedent of an assertion across different design paths. For example, consider assertions **C1**: $a \rightarrow f$ and **C2**: $b \rightarrow f$. Assume that the spatial distance between the references of a and the assignment of f is three statements, and that between the references of b and the assignment of f is one statement. In other words, to understand the design behavior captured by **C1**, one needs to analyze three design statements, whereas to understand the design behavior captured by **C2**, one needs to analyze one design statement. Intuitively, one can see that a in **C1** should have a higher importance score than b in **C2** with respect to f as a captures more subtle behavior because of its *higher spatial distance* than

b with respect to f. Now consider **C1**: $a \rightarrow f$ and **C3**: $a \rightarrow \mathcal{X}\mathcal{X}f$. While both **C1** and **C3** refer to a in their antecedents, **C3** refers to a two cycles away from f, as **C3** can be rewritten as $a^{-2} \rightarrow f$ (c.f., Section 6.2.1). That implies that to understand the design behavior captured by **C3**, one needs to reason over two clock cycles, whereas to understand the design behavior captured by **C1**, one needs to reason over one clock cycle. Intuitively, we see that a^{-2} in **C3** should have higher importance than a in **C1**, as a^{-2} captures more subtle behavior because of its *higher temporal distance* than a with respect to f. A variable with higher temporal and spatial distance affects the visible output behavior of the target variable convolutedly. Consequently, the captured behavior is subtle and directly invisible from the source code. A higher importance score for a temporally and spatially distant variable with respect to the target variable quantifies such behaviors.

We compute a *relative importance score* $I_r(v_i, v_t)$ that captures how important a variable v_i is with respect to the target variable v_t of an assertion **P**. The relative importance score emphasizes i) the temporal distance and the spatial distance between a variable v_i in the antecedent and the target variable v_t, and ii) the importance of the covered execution paths between the references to the given variable v_i in the antecedent and assignments to the target variable v_t of an assertion. A variable's relative importance score is transitively dependent on the relative importance scores of the variables it assigns. We use relative importance scores of the variables to calculate an assertion's importance score. An assertion with a high importance score has high temporal and spatial distance between the target variable and the variables in the antecedent and covers the design path(s) that are most important to the target.

Definition 22 *The* ***assertion importance*** *of an assertion* **P** *for a target variable* v_t *is defined as the sum of the relative importance scores* I_r *of the variables in the antecedent of* **P** *with respect to the* v_t*. It is calculated as* $I(\mathbf{P}) = \sum_{v_i \in \mathbf{V}_a} I_r(v_i, v_t)$*, where* $\mathbf{V}_a$ *is the set of variables in the antecedent of* **P**.

It is desirable to construct high-importance assertions for design verification/validation and debugging. A high-importance assertion is composed of a large number of important design variables with respect to the target

variable. Assertions that use such variables usually span a large number of design statements and a large number of clock cycles, and typically have many propositions in their antecedent that hinder its *understandability* and diminishes its *practical usability*. Note that the targeted use cases (*e.g.*, design verification/validation, understanding, and debugging) of our ranking method always have a human-in-the-loop who needs to comprehend the design behavior that is captured by an assertion for effective localization and debugging. In order to *balance the importance* and *the understandability* of the design behavior that is captured by an assertion, we calculate assertion complexity.

Assertion complexity: The assertion complexity attempts to quantify and compute the human comprehensibility of an assertion. To quantify the complexity of an assertion, we need to quantify the complexity of each of the variables in the antecedent of an assertion with respect to the target variable. We argue that the understandability decreases as more and more lines of source code need to be investigated to understand how a variable affects the output behavior of the target variable. A variable is *highly complex* if it requires investigation of a large fraction of Verilog source code to understand its effect on the output behavior. An assertion is *complex* if it is composed of many complex design variables with respect to the target variable. The presence of complex design variables in an assertion enables it to capture complex design behaviors.

Variable complexity: To calculate assertion complexity, we calculate the *relative complexity score* $C_r(v_i, v_t)$ of each of the variables v_i in the antecedent of an assertion with respect to the target variable v_t. The relative complexity score captures the understandability of the dependencies between the target variable and the variable(s) in the antecedent. The relative complexity score emphasizes i) the temporal distance and the spatial distance between a variable in the antecedent and the target variable, and ii) the understandability of the execution paths between references to a given variable in the antecedent and the assignments to the target variable in the consequent. Consider **C3**, which can be rewritten as $a \rightarrow \mathcal{X}^2 f$ (c.f., Section 6.2.1). A validator would need to search transitively for all assignments to f for two clock cycles until the validator finds an assignment in which a is referenced. The complexity score computation of a variable is based on the following tasks: i) analysis of

each variable in a large expression, and ii) development of an understanding of the relationship between spatially and temporally separated variables. We use the relative complexity scores of the variables to calculate an assertion's complexity score. An assertion with a high complexity score has high temporal and spatial distance between the target variable and the variables in the antecedent and covers complex design path(s) between the satisfaction of its antecedent and the truth of its consequent.

Definition 23 *The **assertion complexity** of an assertion* $\mathbf{P}$ *for a target variable* v_t *is defined as the sum of the relative complexity scores* C_r *of the variables in the antecedent of* $\mathbf{P}$ *with respect to the* v_t*. It is calculated as* $C(\mathbf{P}) = \sum_{v_i \in \mathbf{V}_a} C_r(v_i, v_t)$*, where* $\mathbf{V}_a$ *is the set of variables in the antecedent of* $\mathbf{P}$*. The* $C(\mathbf{P})$ *considers all variables in the antecedent of an assertion equally irrespective of their relation to one another via operators. Each variable contributes its relative complexity score to an assertion for its every appearance in the antecedent.*

Ranking: An assertion's rank estimates the importance and the understandability of the design behavior that it captures. In order to facilitate design understanding, validation, and debugging, our proposed ranking method ranks an assertion higher that has a balanced composition of important and complex design variables with respect to the target variable. In other words, the ranking method ranks an assertion higher that has high importance and is also easy to comprehend.

6.3.2 Calculation of importance and complexity of the variables in an assertion

Global importance score: We use the method of Section 3.4.2 to compute global importance score of each of the variable in the RTL design. Let $I_g(v)$ denotes the global importance score of v.

Relative importance and complexity score: Algorithm 2 details the relative importance and complexity score calculation for each variable on which v_t depends within a bounded number of temporal frames k_{max}. The algorithm constructs the relative dependency graph $G_r = (V_r, E_r)$. It requires inputs v, k, and v_t where v denotes the current variable in the depth-first construction

of G_r, and k denotes the current temporal index. Further, let k_{max} denote the maximum temporal length of G_r. We quantify the complexity of a design variable as the number of transitive traversals of the Verilog source code that needs to be made from the assignment of the target variable to the reference of the variable under consideration. Let the function $dependencies(v)$ returns the set of variables on which v depends within one temporal frame; let $temporal(v)$ be *true* if assignments to v span multiple temporal frames; let $expressions(v)$ returns the set of expressions that defines v within one temporal frame; let $sensitivities(v)$ returns the set of expressions that reference v; and let $size(X)$ returns the number of variables used in expression X.

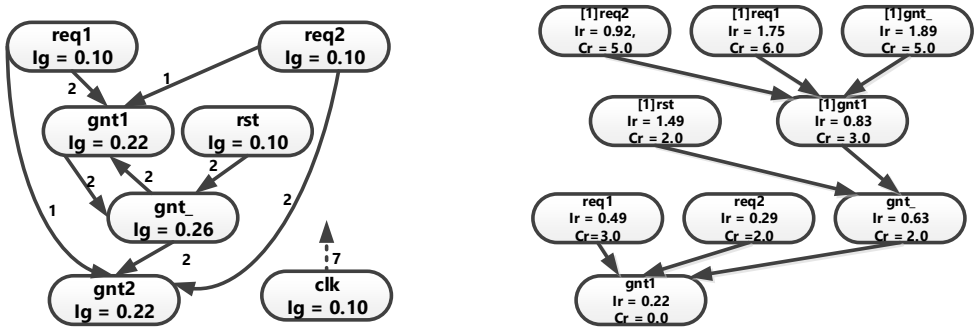
Algorithm 2 Relative variable importance and complexity

```
 1: procedure calc_imp_complx(v, k, v_t)
 2: if k < k_max then
 3:    V ← dependencies(v)
 4:    X ← expressions(v)
 5:    for all v_i ∈ V do
 6:       i_r ← I_g(v_i) + I_r(v, v_t)
 7:       S ← sensitivities(v)
 8:       C_r ← 0
 9:       for all X_i ∈ X ∩ S do
10:          c_r ← c_r + size(X_i)
11:       end for
12:       C_r ← C_r + c_v
13:       V_r ← V_r ∪ (v_i, i_r, C_r)
14:       E_r ← E_r ∪ (v_i, v)
15:    end for
16:    for all v_i ∈ V do
17:       if temporal(v_i) then
18:          calc_imp_complx(v_i, k + 1, v_t)
19:       else
20:          calc_imp_complx(v_i, k, v_t)
21:       end if
22:    end for
23: end if
```

Algorithm 2 checks whether $k < k_{max}$ and terminates if it is not. For each variable on which v depends, the algorithm adds a new node and edge to G_r. The algorithm computes the relative importance score i_r of v_i^{-k} by summing the global importance score of v_i and relative importance score of v with respect to v_t. It also computes the relative complexity score of v_i^{-k} by summing the sizes of the expressions in $X \cap S$ that contains expressions

Table 6.1: Example assertions for the two-port arbiter of Figure 3.1.

ID	Assertions
a0	$(req2 == 1 \wedge gnt\_ == 1)$ ##1 (req1 == 1) $\rightarrow$ $(gnt1 == 1)$
a1	$(req1 == 1 \wedge req2 == 0) \rightarrow (gnt1 == 1)$



(a) (b)

Figure 6.2: Variable dependency graphs (VDG) (same as Figure 3.2) for the two-port arbiter of Figure 3.1. We redraw the VDG for ease of understanding. (a) shows the global VDG where $\mathbf{V} = \{req1, req2, gnt\_, gnt1, gnt2, clk, rst\}$. I_g is the global importance score of a node. (b) shows the relative VDG for `gnt1`. I_r is the relative importance of a node. C_r is the relative complexity of a node.

that both define v and use v_i. Next, for each variable v_i on which v depends, the algorithm increases k if v_i is temporal and recurses.

6.3.3 Assertion ranking based on assertion importance and assertion complexity

We rank a set of assertions by using their *importance* and *complexity* scores as defined in Section 6.3.1 (c.f., Definition 22 and Definition 23). The rank score of an assertion $\mathbf{P}$ is defined as $IRank(\mathbf{P}) = I(\mathbf{P})/C(\mathbf{P})$. Our ranking ensures that an assertion that captures most important design behavior(s) for a target (high assertion importance score) and easy to comprehend (low assertion complexity score), is ranked higher.

6.3.4 Example of *IRank*- and *SRank*-based assertion rankings

We use two assertions shown in Table 6.1 for the two-port arbiter of Figure 3.1. **a0** captures a non-trivial temporal property, whereas **a1** captures a trivial combinational property. Hence, we expect both the importance and complexity scores of **a0** to be higher than those of **a1**.

***IRank* calculation**: Figure 6.2a shows the global VDG of the two-port arbiter of Figure 3.1, labeled with variable names, edge weights, and global importance scores (c.f., Section 3.2.1). Figure 6.2b shows the relative VDG for $gnt1$ as constructed by Algorithm 2. Since **a0** and **a1** are each two cycles long, $k_{max} = 2$. Both algorithms begin with $v = gnt1$, $k = 0$, and $v_t = gnt1$. Since $k < k_{max}$, both algorithms continue.

Line 3 in Algorithm 2 initializes the set V to the variables on which $gnt1$ depends ($V = \{req1, req2, gnt\_\}$), and line 4 initializes the set X to the expressions on which $gnt1$ depends ($X = \{gnt\_, \{req1 \ \&\neg req2\}, req1\}$). Line 7 initializes the set S to the expressions that reference $req1$, ($S = \{req1 \ \& \ \neg req2, req1\}$). Lines 13 –14 add a node for each of the variables in V to the relative dependency graph.

In Algorithm 2, line 6 computes the relative importance score of $req1$ as $I_r(req1, gnt1) = 2 * 0.20 + 0.09 = 0.49$. Since $gnt1$ uses $req1$ in two assignments, $gnt1$ contributes its relative importance score twice to the relative importance score of $req1$. Line 9 computes $X \cap S = S$, and lines 9 –12 compute the relative complexity score of $req1$ as $C_r(req1, gnt1) = 2 + 1 = 3$. Algorithm 2 recurses in lines 16 –21. Since $req1$ and $req2$ are inputs, they do not depend on any variables. Therefore, the algorithm terminates in each of these recursive calls. When Algorithm 2 recurses on $gnt\_$, it increments k, since $gnt\_$ is temporal. Algorithm 2 terminates when $k \geq k_{max}$ or when $V = \emptyset$.

The importance score for **a0** is I(**a0**) = 0.92 + 1.89 + 0.49 = 3.30 and for **a1** is I(**a1**) = 0.72. The complexity score for **a0** is C(**a0**) = 5.00 + 5.00 + 3.00 = 13.00, and for **a1** is C(**a1**) = 5.00. Finally, the rank for **a0** is *IRank*(**a0**) = I(**a0**)/C(**a0**) = 0.254 and for **a1** is *IRank*(**a1**) = I(**a1**)/C(**a1**) = 0.144.

***SRank* calculation**: We calculate *SRank* for **a0** and **a1** according to the modified statement-coverage-based algorithm of Section 6.2.3. Since **a0** is temporal, according to the algorithm in [29], randomization of rst in the

Table 6.2: Manually written assertions for $pci\_master32\_sm$ module for the output $pci\_frame\_en\_out$.

ID	Assertions	*IRank*
b1	$(cur\_state[3] == 1)\#\#1(rdy\_in == 0) \rightarrow (pci\_frame\_en\_out == 0)$	1
b2	$(rdy\_in == 0)\#\#1(pci\_frame\_out\_in == 1 \& pci\_trdy\_in == 0 \& rdy\_in == 0) \rightarrow (pci\_frame\_en\_out == 0)$	2

first cycle followed by a randomization of rst and $gnt\_$ in the second cycle will cover S1 –S9. Since **a1** is combinational, the randomization of $gnt\_$ will cover S5 –S10. Figure 3.1 has nine non-trivial statements; hence *SRank*(**a0**) = 9/9 = 1.0 and *SRank*(**a1**) = 6/9 = 0.66.

6.4 Case study of ranked assertions as an aid in debugging

In this section we show that a ranked list of assertions can aid the debugging process. We wrote two assertions on the Peripheral Component Interconnect (PCI) [185] bridge master state machine for the $pci\_frame\_en\_out$, as shown in Table 6.2. We ranked them using our assertion ranking methodology. The buggy PCI master state machine code is shown in Figure 6.3. We simulated the buggy PCI code along with the assertions **b1** and **b2**, and both of the assertions failed.

6.4.1 Functionality of ***pci_frame_en_out***

pci_frame_en_out is the enable signal for the output ***pci_frame_out***. ***pci_frame_out*** signals that the master state machine is transferring data on the bus. During a transfer, the signal ***pci_frame_en_out*** should remain enabled unless a master abort occurs.

```
1  module pci_master32_sm(input clk_in,
2   reset_in, pci_gnt_in, pci_frame_in,
3   pci_frame_out_in, pci_irdy_in,
4   pci_trdy_in, pci_stop_in, req_in, rdy_in,
5   output pci_frame_out, pci_frame_en_out);
6  reg sm_idle, sm_address, sm_data_phases,
7   sm_turn_arround;
8  reg [3:0] cur_state, next_state;
9
10 wire ch_state_slow = sm_address ||
11  sm_turn_arround || sm_data_phases &&
12  (pci_frame_out_in && mabort1 || mabort2);
13
14 wire ch_state_med = ch_state_slow ||
15  sm_idle && u_have_pci_bus
16  && req_in && rdy_in;
17
18 wire change_state = ch_state_med ||
19  sm_data_phases &&
20  (~(pci_trdy_in && pci_stop_in));
21
22 wire u_dont_have_pci_bus = pci_gnt_in
23  || ~pci_frame_in || ~pci_irdy_in;
```

(a)

```
1  wire u_have_pci_bus = ~pci_gnt_in &&
2   pci_frame_in && pci_irdy_in;
3
4  wire frame_en_slow = (sm_idle &&
5   u_have_pci_bus && req_in || rdy_in)
6   || sm_address || (sm_data_phases
7   && ~pci_frame_out_in);
8
9  wire frame_en_keep = sm_data_phases
10  && pci_frame_out_in && ~mabort1
11  && ~mabort2;
12
13 assign pci_frame_en_out =
14  frame_en_slow || frame_en_keep &&
15  pci_stop_in && pci_trdy_in;
16
17 always @ (posedge reset_in or
18 posedge clk_in)
19   if (reset_in)
20     cur_state <= S_IDLE;
21   else if (change_state)
22     cur_state <= next_state;
```

(b)

```
1  always @ (cur_state or do_write
2  or pci_frame_out_in)
3  begin
4    sm_idle = 1'b0 ;
5    sm_address = 1'b0 ;
6    sm_turn_arround = 1'b0 ;
7    case (cur_state)
8    S_IDLE:
9    begin
10     sm_idle = 1'b1 ;
11     next_state = S_ADDRESS ;
12   end
13   S_ADDRESS:
14   begin
15     sm_address = 1'b1 ;
16     next_state = S_TRANSFER ;
17   end
18   S_TA_END:
19   begin
20     sm_turn_arround = 1'b1 ;
21     next_state = S_IDLE ;
22   end
23   endcase
24 end
25 endmodule
```

(c)

Figure 6.3: A buggy implementation of the PCI bridge master state machine. The state encodings are `S_IDLE = 4'h1`, `S_ADDRESS = 4'h2`, and `S_TA_END = 4'h8`. A bug is injected at line 5 of (b) through mutation of the logical `and` operator (&&) before `rdy_in` to logical `or` operator (||).

6.4.2 Debugging with assertion **b1**

Consider the first cycle of **b1**. The proposition $\boldsymbol{cur\_state[3] == 1}$ indicates that the state machine's current state is **S_TA_END**. Line 21 of Figure 6.3b shows that the state machine will change state if $\boldsymbol{change\_state}$ is asserted. Since the state machine's current state is **S_TA_END**, $\boldsymbol{sm\_turn\_around == 1}$, $\boldsymbol{sm\_idle == 0}$, and $\boldsymbol{sm\_address == 0}$. That implies that $\boldsymbol{ch\_state\_slow == 1}$ (line 10 of Figure 6.3a), $\boldsymbol{ch\_state\_med == 1}$ (line 14 of Figure 6.3a), and, consequently, $\boldsymbol{change\_state == 1}$ (line 18 of Figure 6.3a). In the next cycle, the current state of the state machine is **S_IDLE** which implies that $\boldsymbol{sm\_turn\_around == 0}$, $\boldsymbol{sm\_idle == 1}$, and $\boldsymbol{sm\_address == 0}$. The result is that $\boldsymbol{frame\_en\_keep == 0}$ (line 9 of Figure 6.3b), which makes $\boldsymbol{frame\_en\_keep}$ && $\boldsymbol{\ldots == 0}$ (line 13 of Figure 6.3b). Consequently, **b1** failed because $\boldsymbol{frame\_en\_slow}$ was asserted. Investigation of line 4 of Figure 6.3b shows that $\boldsymbol{frame\_en\_slow}$ could only be asserted if the first OR'ed expression equals 1 (as $\boldsymbol{sm\_address == 0}$ and $\boldsymbol{sm\_data\_phases == 0}$). As $\boldsymbol{rdy\_in == 0}$ is set by the proposition of **b1** in the second cycle ($\boldsymbol{sm\_idle == 1}$, since the current state is **S_IDLE**, and $\boldsymbol{u\_have\_pci\_bus}$ and $\boldsymbol{req\_in}$ are asserted by primary inputs), the signal $\boldsymbol{frame\_en\_slow}$ can be asserted only if $\boldsymbol{rdy\_in}$ is OR'ed instead of AND'ed. The PCI specification says that if a slave device is not ready (as indicated by **rdy_in**), then the master state machine cannot transfer data. Fixing of the logical OR by the logical AND will cause **b1** to pass. In this example, **b1** is consistent with PCI specification and aids debugging by providing valuable hints. Without **b1**'s guidance, a debugging engineer would not be able to identify a starting point for debugging.

6.4.3 Debugging with assertion **b2**

Consider the first cycle of **b2** that has $\boldsymbol{rdy\_in == 0}$. The expressions at line 14 of Figure 6.3a and at line 4 of Figure 6.3b cannot be evaluated definitively via $\boldsymbol{rdy\_in == 0}$ because of the unknown values of other propositions, such as $\boldsymbol{sm\_idle}$, $\boldsymbol{sm\_address}$, and $\boldsymbol{ch\_state\_slow}$. Hence, we start going backwards for the assertion **b2**.

Consider the second cycle of **b2**. The proposition $\boldsymbol{pci\_trdy\_in == 0}$ ensures that the second disjunctive term of the `assign` statement at line

13 of Figure 6.3b is not asserted. The failure of **b2** implies that the signal ***pci_frame_en_out*** == **1**, which in turn implies that ***frame_en_slow*** == **1**. Since the proposition ***pci_frame_out_in*** == **1**, the third disjunctive term of the expression at line 4 of Figure 6.3b is deasserted. Consequently, ***frame_en_slow*** will be asserted if either ***sm_address*** == **1** (*i.e.*, the current state of the state machine in cycle 2 is **S_ADDRESS**) or ***sm_idle*** == **1** (*i.e.*, the current state of the state machine in cycle 2 is **S_IDLE**, as ***rdy_in*** == **0** was set by the proposition in **b2**).

Case I: Assume that the current state of the state machine in cycle 2 is **S_ADDRESS**. Consequently, the state machine's current state in the first cycle of **b2** is **S_IDLE**. In the first cycle of **b2**, ***rdy_in*** == **0** ensures that ***change_state*** == **0** (***sm_address*** == **0**, ***sm_data_phases*** == **0**, and ***sm_turn_around*** == **0**, which assigns ***ch_state_slow*** == **0**, which, in turn, assigns ***ch_state_med*** == **0**). That implies that the state machine's current state in the second cycle cannot be **S_ADDRESS** and therefore ***pci_frame_en_out*** cannot be asserted. That is a contradiction. Hence, our assumption that **S_ADDRESS** is the state machine's current state in cycle 2 is wrong.

Case II: Assume that the current state of the state machine in cycle 2 is **S_IDLE**. Consequently, the state machine's current state in the first cycle of **b2** is **S_TA_END**, implying that ***sm_turn_around*** == **1**. In the first cycle of **b2**, ***sm_turn_around*** == **1** ensures that ***ch_state_slow*** == **1**, which, in turn, makes ***ch_state_med*** == **1** and ***change_state*** == **1**. Therefore, the state transition from **S_TA_END** → **S_IDLE** from cycle 1 to cycle 2 is valid. As ***rdy_in*** == **0** is set by the proposition of **b2** in the second cycle (***sm_idle*** == **1** since the current state is **S_IDLE**, and ***u_have_pci_bus*** and ***req_in*** are asserted by primary inputs), the signal ***frame_en_slow*** can be asserted only if ***rdy_in*** is OR'ed instead of AND'ed. The PCI specification says that if a slave device is not ready (as indicated by **rdy_in**), then the master state machine cannot transfer data. Fixing of the logical OR by the logical AND causes **b1** to pass.

Clearly, both **b1** and **b2** helped to debug the failure. Debugging with **b2** is convoluted and required complex reasoning, whereas debugging with **b1** is simpler and easy to reason about. Hence, one would prefer **b1** to **b2** for debugging.

Table 6.3: Details of different USB and PCI design modules. **LOCs**: lines of executable Verilog code in each design module. **NAA**: number of automatic assertions generated by GoldMine [41] for each design module.

Module name	**LOCs**	**Target Variables**	**NAA**
usbf_idma	361	3	20
usbf_pa	314	4	18
usbf_pd	351	21	758
usbf_pe	651	3	87
usbf_wb	196	4	612
pci_master32_sm	560	9	30

This case study underscores our claim that a ranked list of assertions can improve the debugging process and can help to prioritize the debugging effort.

6.5 Experimental setup

Design testbed: We used publicly available USB [154] and PCI [185] designs from OpenCores to demonstrate our results. Table 6.3 details different USB and PCI design modules. We created five different buggy designs of *usbf_pd* and one buggy design of *usbf_pa*, which we analyzed as six different debugging case studies. The injected bugs are detailed in Table 6.4. We used constrained random testbenches written in SystemVerilog to simulate the buggy designs. We have made the buggy designs and the testbenches available on the web [186].

Assertions used: We used the GoldMine tool [41, 65] to mine assertions for each of the target variables for each of the design modules in Table 6.3. For the sake of completeness, we also used a few manually written assertions for the *usbf_pe* module, shown in Table 6.5.

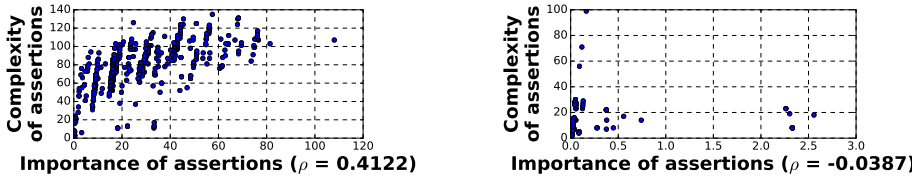
Execution platform: All experiments on the USB and PCI design modules were run on an Intel Xeon CPU E3-1240 8-core processor running at 3.4 GHz with 16 GB RAM.

Table 6.4: Representative bugs injected in different USB design modules. **Bug Category**: Functional implication of the bug.

Bug ID	Module	Injected Bug Detail	Bug Category
1	*usbf_pd*	Wrong state machine transition	Control
2		Wrong condition to send acknowledgement	Control
3		Wrong payload packet ID decoding	Data
4		Wrong latch enable for token storage registers	Control
5		Wrong qualification of receiving token from host	Control
6	*usbf_pa*	Wrong state transition for sending data packets	Data

Table 6.5: Manually written assertions for the *usbf_pe* module for the outputs *send_token* and *rx_dma_en*.

ID	Assertions
m1	$(pid\_SETUP \ \& \ idma\_done \ \& \ \neg abort)\#\#1(state == IDLE) \Rightarrow (send\_token == 1)$
m2	$(csr[27:26] == 2'b01 \ \& \ no\_buf0\_dma)\#\#1(state == IDLE) \Rightarrow (send\_token == 1)$
m3	$(\neg csr[17])\#\#1(to\_large == 1 \ \& \ match == 1)\#\#1(csr[27:26] == 2'b10 \ \& \ state == IDLE) \rightarrow (rx\_dma\_en == 1)$



(a) usbf_wb (b) usbf_pd

Figure 6.4: Graphs (a) and (b) show correlation analysis between assertion importance and assertion complexity.

Table 6.6: Runtime and maximum memory to rank assertions based on importance and complexity (*IRank*) and to rank assertions based on the correctness-based statement coverage of (*SRank*) [29]. To calculate the *SRank* of each assertion, we did 100 iterations to achieve a stable set of covered statements. **T**: runtime in seconds. **Mem**: peak memory usage in MB. **N/A**: measurement not available to report.

Module name	***IRank***		***SRank***	
	T	Mem	T	Mem
usbf_idma	1.2298	832.53	5333.19	444.01
usbf_pa	1.232172	832.28	295.87	227.19
usbf_pe	1.355495	834.52	3307.50	236.63
usbf_wb	1.233261	831.78	4145.22	465.12
usbf_pd	1.247646	832.78	890.63	670.23
pci_master32_sm	1.243567	832.53	3947.99	331.8

6.6 Experimental results

6.6.1 Similarity analysis between assertion importance and assertion complexity

In this experiment, we determined whether assertion importance and assertion complexity are two similar metrics, as implied by the similarity of their calculations as shown in Section 6.3.2. To find any such similarity, we calculated the correlation between assertion importance and assertion complexity.

For this experiment, we used a total of 611 assertions from two different USB design modules (*usbf_wb* and *usbf_pd*). In Figure 6.4a and Figure 6.4b we show the correlations between the *importance* and *complexity* of those 611 assertions. For each such scatter plot, we calculated the Pearson rank correlation coefficient ρ; it is shown in the scatter plot of Figure 6.4a and Figure 6.4b.

The experimental results show that assertion importance and assertion complexity are very weakly correlated. This underscores the point that despite the apparent similarity between assertion importance and assertion complexity calculations, assertion importance and assertion complexity are not dual or similar metrics.

6.6.2 Quantitative comparison between importance/complexity-based ranking and coverage-based ranking methods

In this section, we compare importance/complexity-based ranking (*IRank*) and coverage-based ranking (*SRank*) methods i) to identify the *benefits of using each of the ranking methods* to rank a set of assertions, and ii) *to quantify the overlapping in rankings* for a set of assertions.

We compare *IRank* and *SRank* in terms of i) *computational efficiency* and ii) *effectiveness in debugging* to identify the benefits of each of the ranking methods. To quantify overlapping in rankings, we compare *IRank* and *SRank* in terms of i) *similarity* between assertion importance/coverage-based ranking and assertion complexity/coverage-based rankings and ii) *agreement* in rankings for a set of assertions.

• **Computational efficiency of *IRank* and *SRank* methods**: In this experiment, we compared the computational efficiencies of the *IRank* and *SRank* methods in terms of runtime and peak memory usage when ranking a set of assertions.

For this experiment we used five different USB design modules and one PCI design module. Table 6.6 shows the runtime and peak memory usage found during ranking of assertions using *IRank* and *SRank*, respectively. To calculate the *SRank* of each assertion, 100 iterations were done to stabilize the set of covered statements.

IRank has *up to 3.6*× (*average 2.6*×) more peak memory usage than *SRank*, and *SRank* needed *up to 4366*× (*average 2824.5*×) more runtime for ranking than *IRank*.

IRank incurs high memory usage since it needs to construct the global variable dependency graph and relative variable dependency graph per target variable (c.f., Algorithm 2) at the beginning of the ranking process. That is a memory-intensive operation. On the other hand, *SRank* needs to construct the CDFG of the complete RTL design only once at the beginning of the ranking, and can store it in the main memory of the system. The CDFG can be reused later for any subsequent design execution for ranking. That is a cheaper operation.

Once the importance and complexity scores of all the design variables on which a target variable depends (within a bounded number of temporal

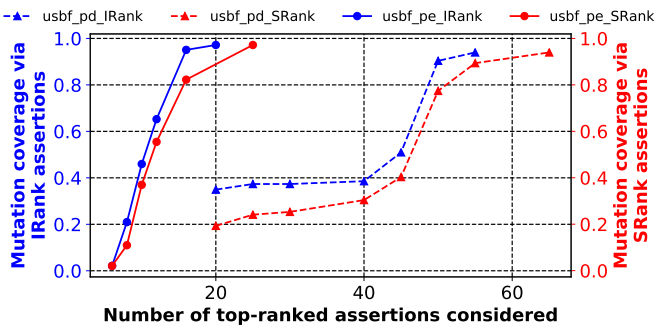


Figure 6.5: Cumulative mutation coverage of top-ranked assertions from *IRank* and *SRank*.

Table 6.7: Bug detection statistics for mutants that were randomly injected using top-ranked assertions from *IRank*. **Mod**: Module under consideration. **TMuts**: Total number of randomly injected mutants. **KMuts**: Number of mutants that caused one or more top-ranked assertion(s) to fail. **PMut**: Percentage of mutants killed. **M/m/A**: Maximum, minimum, and average numbers of assertions failed per injected mutant. **Unq**: Number of unique assertions failed. **M1**: usbf_pe, **M2:** usbf_pd.

Mod	**KMuts/ TMuts**	**PMut**	***IRank***		***SRank***	
			M/m/A	**Unq**	**M/m/A**	**Unq**
M1	138 / 142	97.18%	6 / 4 / 4.97	6	6/3/4.97	7
M2	78 / 83	93.97%	9 / 1 / 2	30	15/2/4.03	32

frames) have been calculated, those scores are reused to calculate importance and complexity for all assertions for a given target variable. Since calculation of assertion importance and complexity consists only of additions (c.f., Definition 22 and Definition 23), *IRank* needs much less time to calculate rank score and to rank a set of assertions. On the other hand, to calculate the statement coverage of each assertion, *SRank* needs to initialize variables in the CDFG based on the propositions in the antecedent of an assertion, randomize the remaining free variable(s), execute the CDFG, and backtrack to identify covered statements (c.f., Section 6.2.2 and Section 6.2.3). Those are time-intensive computations. Hence, *SRank* needs much more time than *IRank* to rank a set of assertions.

This experiment shows that *IRank* is a more computationally efficient ranking method than *SRank*.

• **Debugging effectiveness of top-ranked assertions by *IRank* and *SRank***: In this experiment, we quantitatively compared bug detectability

Table 6.8: Debugging statistics for our case studies that used top-ranked assertions according to *IRank* and *SRank*. **NAF**: Number of assertions failed. **Loc**: Number of design statements investigated. **NA**: Localization not available as no assertions failed.

Bug ID	***IRank* Assertions**		***SRank* Assertions**	
	NAF	**Loc**	**NAF**	**Loc**
1	2	9	2	12
2	3	10	3	12
3	4	7	None	NA
4	1	7	None	NA
5	3	7	3	13
6	1	8	1	10

of the top-ranked assertions found by the *IRank* and *SRank* metrics.

For this experiment, we used two different USB design modules, namely usbf_pe and usbf_pd. For each of the modules, we created two different sets of buggy designs. For the first set of buggy designs, we randomly injected one bug per buggy design by using an in-house Verilog code mutation engine [187]. In the second set of buggy designs, we manually injected one bug (c.f., Table 6.4) at a time per buggy design. The manually injected bugs more closely resembled real-world human errors. We simulated each of the buggy designs by using a constrained random testbench along with top-ranked assertions from the *IRank* and *SRank* methods.

We observe (c.f., Figure 6.5 and Table 6.7) that as more and more top-ranked assertions from *IRank* and *SRank* are included, the *mutation coverage increases monotonically*. We were able to achieve *up to 97.18% (average 95.75%) mutant coverage* by using top-ranked assertions from *IRank* and *SRank*. Further, to achieve similar mutant coverage, we needed *up to 10 (average 8)* more top-ranked assertions from *SRank* than from *IRank*. Please note that our assertion ranking methodology is *orthogonal* to the assertion generation methodology [41, 42]. The ranking methodology identifies a set of assertions based on their "goodness." Consequently, if the original set of assertions fails to capture complete design behavior, the top-ranked assertions will not be able to detect all bugs. For that reason, in our analysis, we were not able to achieve 100% mutant coverage with the top-ranked assertions.

Our analysis (c.f., Table 6.8) of the second set of buggy designs shows that for each of the manually injected bugs, *up to 4* top-ranked assertions from

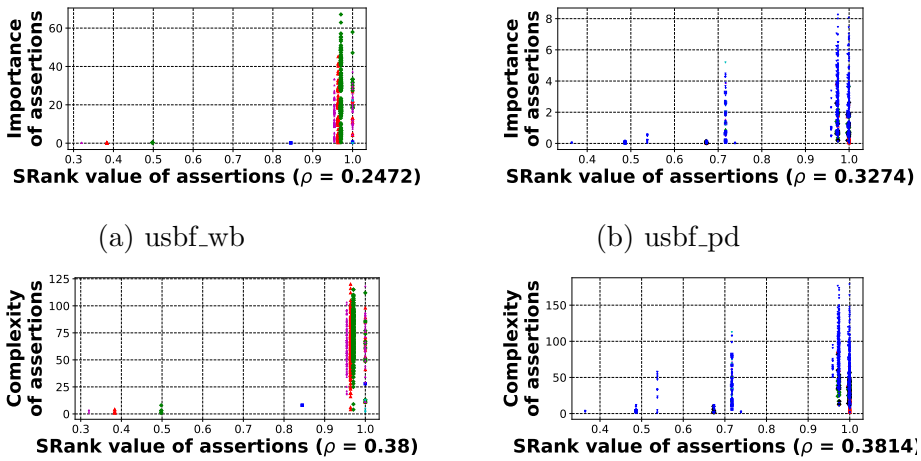
IRank detected the presence of the bug in the design whereas the top-ranked assertions from *SRank* failed to detect the presence of a bug in as many as *two case studies*. Further, the top-ranked assertions from *IRank* localized the bug within *no more than 10 statements* (*average 8 statements*) whereas the top-ranked assertions from *SRank* localized the bug only *within 12 statements* (*average 11.75 statements*). We observe that top-ranked assertions from *IRank* detected *1.4 bugs* per assertion, whereas top-ranked assertions from *SRank* detected only *0.9 bugs* per assertion, implying that **top-ranked assertions from *IRank* detect 1.5× more bugs per assertion than do top-ranked assertions from *SRank***.

IRank ranks an assertion higher that has more important design variables and cover more important design paths. On the other hand, *SRank* ranks an assertion higher if it has a broader scope, *i.e.*, a larger fraction of design statements must be executed in order for that assertion to be non-vacuously true. *SRank* has no systematic way to identify important design variables in an assertion. That different perspective of *SRank* causes it to rank assertions with poor detectability at the top of the ranked list. In Section 6.8.6 and Section 6.8.7, we discuss two debugging case studies to provide more technical insights.

This experiment shows that top-ranked assertions from *IRank* are more effective than top-ranked assertions from *SRank* in detection and localization during debugging.

• **Similarity analysis between assertion importance and coverage-based ranking, and assertion complexity and coverage-based ranking**: In this experiment, our objective was to identify similarities between assertion importance and coverage-based ranking, and assertion complexity and coverage-based ranking. To find similarities, we calculated the correlation between the two components of *IRank*, *i.e.*, assertion importance and assertion complexity with coverage-based ranking.

For this experiment, we used the same set of assertions that we used in Section 6.6.1. Figure 6.6a and Figure 6.6b analyze the correlation between *assertion importance* and *coverage-based ranking*, and Figure 6.6c and Figure 6.6d analyze correlation between *assertion complexity* and *coverage-based ranking*. For each such scatter plot, we have also calculated the Pearson rank correlation coefficient ρ which we have shown in the scatter plots of Fig-



(a) usbf_wb (b) usbf_pd

(c) usbf_wb (d) usbf_pd

Figure 6.6: Graphs (a) and (b) show correlation analysis between assertion importance and statement coverage-based ranking, and graphs (c) and (d) show correlation analysis between assertion complexity and statement coverage-based ranking.

ure 6.6a, Figure 6.6b, Figure 6.6c and Figure 6.6d.

This experiment shows that assertion importance and coverage-based ranking, and assertion complexity and coverage-based ranking, have low to no correlation. This emphasizes that the design aspects captured by the metrics are different.

• **Comparison between rankings by *IRank* and *SRank***: In this experiment, for a set of assertions, we assessed the agreement in rankings between *IRank* and *SRank*. We considered only the top 20% and bottom 20% of

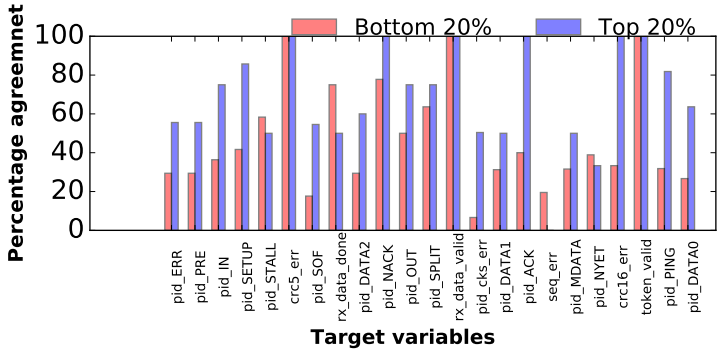


Figure 6.7: Extent of agreement between *IRank*'s and *SRank*'s rankings of the top 20% and bottom 20% assertions on the usbf_pd module.

assertions from the two ranked lists for this analysis.

We considered a total of 1443 assertions for the *usbf_pd* module and ranked them using *IRank* and *SRank*. In Figure 6.7 we show the extent of agreement about the top 20% and bottom 20% of assertions between *IRank* and *SRank*. Our analysis shows that *on average, 68.07%* of the top 20% of assertions from *IRank* and *SRank* agree. On the other hand, *on average, 46.45%* of the bottom 20% of assertions from *IRank* and *SRank* agree. We observed that the ranks of the top 20% and bottom 20% assertions from *IRank* and *SRank* agree for only six target variables and three target variables, respectively. Further, when the top 20% and bottom 20% assertions are combined, *on average, the rank agreement is only 57.26%*.

This experiment shows that in spite of different design perspectives that are captured by *IRank* and *SRank*, there is a partial agreement in their ranking that has paved the way to finding a comprehensive ranking for assertions.

6.6.3 Comprehensive ranking for assertions

We would like to combine the diverse perspectives provided by *IRank* and *SRank* to form a comprehensive ranking scheme.

In order to generate a comprehensive ranking for a set of assertions, we explored *rank aggregation* [182, 183, 184]. Rank aggregation is a normalization technique that combines rankings from an arbitrary number of different ranked lists to generate a comprehensive ranking such that the disparity is minimized. The rank aggregation technique measures the disparity between two arbitrary ranked lists using Kendall-Tau (KT) distance [188].

In our empirical analysis we found that the average KT distance between an *IRank* list and an *SRank* list is very high, on average 38.11 per assertion, indicating that those ranked lists disagree for most assertions. Further, our empirical results in Section 6.6.2 show that i) assertion importance and coverage-based ranking, and assertion complexity and coverage-based ranking have low to no correlation, and ii) assertion ranking via *IRank* and assertion ranking via *SRank* do not agree on average on *up to 50%* of assertions. **Those three empirical results together showed that rank aggregation is infeasible to combine *IRank* and *SRank*.**

Based on our experimental results, we attempted a heuristic combination of *IRank* and *SRank* to generate a comprehensive ranking of assertions. We found that a parameterized linear combination of *IRank* and *SRank* is in closer agreement with our empirical findings than the rank aggregation process is. Let $\mathbf{a}$ be an assertion, $FRank(\mathbf{a})$ be the comprehensive rank of $\mathbf{a}$, and $\mathcal{A}, \mathcal{B}$ be two user-configurable parameters then, $FRank(\mathbf{a}) = \mathcal{A} \times IRank(\mathbf{a}) + \mathcal{B} \times SRank(\mathbf{a})$.

Our case studies in Section 6.8.1 through Section 6.8.5 show that both *IRank* and *SRank* are effective in ranking a set of assertions by capturing different perspectives on a design. Further, our analysis showed that, in certain cases, *SRank* fails to rank assertions that capture important design behaviors at the top of a ranked list. Hence, we combined *IRank* and *SRank* of an assertion $\mathbf{a}$ in the following way,

$$FRank(\mathbf{a}) = \underbrace{(1 - |\rho|)}_{\mathcal{A}} \times IRank(\mathbf{a}) + \underbrace{|\rho|}_{\mathcal{B}} \times SRank(\mathbf{a})$$

Here ρ is the correlation coefficient between *IRank* and *SRank* for a set of assertions to which $\mathbf{a}$ belongs. The intuition behind our favoring of the *IRank* score when *IRank* and *SRank* have low correlation is that the *IRank* score can capture the presence of important design variable(s) in an assertion more accurately than *SRank* can.

6.7 Comparison of data structures used for importance/complexity-based ranking and coverage-based ranking

The importance/complexity-based ranking uses the global variable dependency graph (VDG) of an RTL as the data structure, whereas coverage-based ranking uses the control data flow graph (CDFG) of an RTL as the data structure. In this section, we summarize the key differences between those two data structures.

• **Design information content**: The VDG *captures the control and data dependencies* among different design variables, but *abstracts away all computations* of the design. The CDFG *captures both control and data dependencies*,

and *all computations* involving different design variables. In a CDFG, two variables v_i, v_j can be control and/or data dependent in one or more different design paths. In a VDG, those dependencies are abstracted as an edge weight between the variables v_i, v_j. Hence, a VDG is an *abstracted representation* of a CDFG.

• **Structural difference**: Since CDFG captures dependencies and computation of a design, the number of nodes and the edges are *orders of magnitude higher* than the VDG of the same design. The number of nodes and the edges in a VDG are bounded by the number of the design variables and their pairwise dependencies in the design. Further, a design usually has *multiple* CDFGs per procedural block of the RTL source code whereas a design can have only one VDG.

• **Executability**: A VDG is a *non-executable abstraction* of a design whereas a CDFG of a design is an executable.

• **Design paths**: Since CDFG is an executable, it can *capture different design paths even if one is rarely executed* whereas VDG does not capture any design paths explicitly.

The above mentioned differences in the two data structures have considerable effects on the importance/complexity-based ranking (*IRank*) and coverage-based ranking (*SRank*) in terms of computational efficiency and bug detectability of top-ranked assertions as reported in Section 6.6. In Section 6.8.1 – Section 6.8.7, we analyze several qualitative case studies to provide further technical insights into *IRank* and *SRank*.

6.8 Qualitative case studies on rank comparison

In this section, we demonstrate seven different case studies to explain the different perspectives that the top-ranked assertions from *IRank* and *SRank* capture with respect to a design. For these case studies, we use assertions of Table 6.9 and Table 6.10 for the $usbf\_pd$ module of the USB design. Table 6.11 details global importance scores of the different design variables of $usbf\_pd$ that appears in the assertions of Table 6.9 and Table 6.10.

Table 6.9: Comparison of ranking of a set of 41 assertions for the target variable *seq_err* of usbf_pd module via *IRank* and *SRank*. **Imp**: The importance score of an assertion. **Com**: The complexity score of an assertion. ***IRank***: *IRank* of an assertion. ***SRank***: *SRank* of an assertion.

ID	Assertions	Imp	Com	***IRank***	***SRank***
a25	$(rx\_err == 1)\#\#1(state[0] == 0 \wedge rx\_err == 0 \wedge rx\_valid == 1 \wedge pid[0] == 0 \wedge pid[3] == 1) \rightarrow (seq\_err == 1)$	2.310	74	3	2
a38	$(rx\_err == 0)\#\#1(state[0] == 0 \wedge rx\_err == 0 \wedge rx\_active == 0 \wedge pid[2] == 1 \wedge pid[0] == 0) \rightarrow (seq\_err == 1)$	2.335	76	4	16
a26	$(pid\_MDATA == 1)\#\#2(state[0] == 0 \wedge rx\_err == 0 \wedge rx\_valid == 1 \wedge pid[0] == 0 \wedge pid[3] == 1) \rightarrow (seq\_err == 1)$	1.033	75	21	1
a1	$(state[0] == 1) \rightarrow (seq\_err == 0)$	0.083	1	1	30
a6	$(rx\_active == 1 \wedge pid\_PING == 1) \rightarrow (seq\_err == 0)$	0.231	33	41	35

6.8.1 Case study I

Observation: *IRank* ranks an assertion higher based on an assertion's ability to cover design paths that are critical to design's functionality whereas *SRank* ranks an assertion higher based on an assertion's scope and its ability to cover a large number of design statements.

Example and insight analysis: We show an example where both *IRank* and *SRank* rank an assertion at the top of the ranked list. We consider assertion a25 of Table 6.9 that is ranked high by both *IRank* and *SRank*. The assertion a25 contains two high-importance (c.f. Table 6.11) design variables (***rx_err*** in the first cycle and ***state*** in the second cycle). Recall, in the context of an assertion, high-importance variable implies highly connected design variable (c.f. Section 6.3.2). Inclusion of such important design variables in the assertion causes it to cover design paths that are critical with respect to design's correct functionality. *IRank* was able to identify a25's ability to cover important design paths and hence ranked it higher.

On the other hand, a25 contains a design variable (***rx_err*** in the first cycle) that is one cycle apart from the target variable (***seq_err***). The assertion a25 also contains several design variables (such as ***state*** and ***pid***) which are referenced several statements apart from the assignment of the target variable (***seq_err***). These increase the temporal and spatial scope of a25 respectively, thereby allowing execution of a large fraction of design state-

Table 6.10: Top-ranked assertions from *IRank* and *SRank* that are used in debugging experiments for several target variables of usbf_pd module. **NBD**: Number of bug(s) detected by an assertion.

Target variable	***IRank***			***SRank***		
	ID	**Assertions**	**NBD**	**ID**	**Assertions**	**NBD**
seq_err	a25	$(rx\_err == 1)\#\#1(state[0] == 0 \wedge rx\_err == 0 \wedge rx\_valid == 1 \wedge pid[0] == 0 \wedge pid[3] == 1) \rightarrow (seq\_err == 1)$	3	a26	$(pid\_MDATA == 1)\#\#2(state[0] == 0 \wedge rx\_err == 0 \wedge rx\_valid == 1 \wedge pid[0] == 0 \wedge pid[3] == 1) \rightarrow (seq\_err == 1)$	3
	a38	$(rx\_err == 0)\#\#1(state[0] == 0 \wedge rx\_err == 0 \wedge rx\_active == 0 \wedge pid[2] == 1 \wedge pid[0] == 0) \rightarrow (seq\_err == 1)$	4	a25	$(rx\_err == 1)\#\#1(state[0] == 0 \wedge rx\_err == 0 \wedge rx\_valid == 1 \wedge pid[0] == 0 \wedge pid[3] == 1) \rightarrow (seq\_err == 1)$	3
	a21	$(pid[0] == 0)\#\#1(state[0] == 0 \wedge rx\_err == 0 \wedge rx\_active == 0 \wedge pid[3] == 1) \rightarrow (seq\_err == 1)$	3	a20	$(pid\_IN == 1)\#\#1(state[0] == 0 \wedge rx\_err == 0 \wedge rx\_active == 0 \wedge pid[2] == 1) \rightarrow (seq\_err == 1)$	2
pid_cks_err	a201	$(state[0] == 0 \wedge pid\_DATA1 == 1 \wedge state[2] == 1) \rightarrow \#\#1(pid\_cks\_err == 1)$	1	a83	$(pid\_SETUP == 1)\#\#2(pid[3] == 1 \wedge pid[4] == 1 \wedge pid[0] == 1) \rightarrow (pid\_cks\_err == 1)$	0
pid_ACK	a18	$(pid\_ACK == 1 \wedge state[0] == 0)\#\#1(pid\_le\_sm == 0) \Rightarrow (pid\_ACK == 1)$	1	a11	$(pid[0] == 1)\#\#2(pid[2] == 1) \rightarrow (pid\_ACK == 0)$	0

Table 6.11: Global importance scores of the variables in the antecedent of the assertions of Table 6.9 and Table 6.10. **Imp**: the importance score of a design variable.

Variable Name	**Imp**	**Variable Name**	**Imp**
state	0.03303	*pid_PING*	0.01039
pid	0.01988	*pid_SETUP*	
pid_le_sm	0.01076	*pid_ACK*	
pid_MDATA	0.01039	*rx_err*	0.00984
pid_IN		*rx_valid*	
*pid_DATA*1		*rx_active*	

ments of *usbf_pd* between the satisfaction of a25's antecedent and truth of the consequent of a25. The execution of a large number of design statements implies coverage of significant portion of design functionality. In this case, *SRank* was able to identify the broad scope of a25 resulting in considerable *design functionality coverage* and ranked it higher.

6.8.2 Case study II

Observation: The randomization of the *free variables i.e.*, the design variables which do not have concrete values, can significantly affect coverage-based ranking of an assertion. Randomization is not a part of the *IRank* computation, and as such, does not affect it.

Example and insight analysis: We show an example where *IRank* ranks an assertion at the top of the list whereas *SRank* ranks it lower due to improper randomization of the free variables. We consider assertion a38 of Table 6.9 which is ranked higher by *IRank* but ranked lower by *SRank*. The a38 is similar to the a25 of Table 6.9 that was discussed in Section 6.8.1. Following the same argument of Section 6.8.1, presence of high-importance variables (***rx_err***, ***state***, ***pid***) allows a38 to cover design execution paths that are critical with respect to design's functionality and hence *IRank* ranked it higher.

In comparison, for a38, the *randomization* of the *free variables i.e.*, the design variables without concrete assignments, created variable value combinations which do not satisfy a branch condition or case condition. This caused several design statements to not execute (*e.g.*, statements in the *true* branch or the statements in a case condition) during statement coverage anal-

ysis. This reduced the fraction of design statements that are in the scope of a38. A reduced fraction of design statements in the scope of an assertion implies reduced coverage of design functionality and hence *SRank* ranked it lower.

This is primarily due to that fact that *SRank* relies on *dynamic analysis* of the design CDFG. In contrast, *IRank* statically analyzes the *VDG* of a design. This allows *IRank* to always identify important design variables and rank assertions higher that contain such important design variables.

6.8.3 Case study III

Observation: To rank assertions, *SRank* prioritizes *spatio-temporal relationship* between the target variable and the variable(s) in the antecedent.

Example and insight analysis: We show an example where *SRank* ranked an assertion higher due to the presence of temporally and spatially separated variables in the antecedent with respect to the target variable whereas *IRank* identified the presence of less important design variables in that assertion and ranked it lower. We consider assertion a26 of Table 6.9 which is almost similar to a25 (analyzed in Section 6.8.1) except i) a26 has a different design variable in the first cycle (***pid_MDATA*** instead of ***rx_err***) and ii) a26 is two cycles long. Although, ***pid_MDATA*** has higher global importance score than ***rx_err*** (c.f., Table 6.11), but the relative importance score of ***pid_MDATA*** at two cycles away from the target variable (***seq_err***) is much lower than that of relative importance score of ***rx_err*** at one cycle away from the target variable. Presence of low importance variable (***pid_MDATA***) caused a26 to cover design paths that are not critical to design's functionality. *IRank* was able to distinguish this subtle difference in design functionality coverage of assertion a25 and a26 and ranked a26 lower.

On the other hand, a26 has variable (***pid_MDATA***) that is temporally wide apart from the target variable (***seq_err***) and has variables (***rx_err***, ***state***, and ***pid***) that are referenced several statements apart from the assignment of the target variable (***seq_err***). This causes to widen the temporal and spatial scope of the assertion a26 and to execute a large fraction of design statements while calculating the *SRank* score of a26. Consequently, *SRank* ranked a26 higher in the ranked list. Unlike *IRank*, *SRank* failed to

distinguish the presence of a variable (***pid_MDATA*** in the first cycle) that has a low relative importance score with respect to the target variable and prioritizes the spatio-temporal relationship between the target variable and the variables in the antecedent.

6.8.4 Case study IV

Observation: *IRank* ranks assertion based on its coverage of important design paths irrespective of whether an assertion is combinational or temporal whereas *SRank*'s reliance on statement coverage causes *SRank* to miss an assertion's relevance to the important design behaviors.

Example and insight analysis: We show an example where *IRank* identified presence of an important variable in a combinational assertion and ranks it higher whereas *SRank* ranks it lower due to limited scope of the assertion. We consider assertion a1 of Table 6.9 which is unlike a25, a38, and a26, is a combinational assertion. In-depth inspection of a1 shows that it contains a high-importance (***state***) (c.f., Table 6.11) variable that enables a1 to capture an important design functionality of the state machine of *usbf_pd*. This is interesting since *IRank* was able to identify a1's relevance with respect to design functionality even if a1 is a combinational assertion. Further, complexity of the variable in the antecedent of a1 (***state***) is 1 making a1 to convey *an important design behavior with most comprehensibility*. Consequently, *IRank* ranks it higher.

On the other hand, lack of temporally and spatially separated variable with respect to the target variable (***seq_err***) severely limits the scope of the assertion a1. This causes a tiny fraction of design statements to execute while calculating statement coverage of a1 causing it to cover much less design behavior. Consequently, *SRank* ranks it lower.

6.8.5 Case study V

Observation: Both *IRank* and *SRank* rank assertions low that lack any important design variables and have limited scope.

Example and insight analysis: We consider assertion a6 of Table 6.9. It does not contain *high-importance* variables (c.f., Table 6.11), consequently,

it does not cover important design functionality and hence *IRank* ranks it lower. Also, a6 does not contain variables in its antecedent that are temporally/spatially separated with respect to the target variable (***seq_err***). Hence the scope of the assertion is limited and executed a tiny fraction of design statements covering very less functionality. Consequently, *SRank* ranks it lower too.

In the next two sections, we present two case studies to elaborate our observations that *IRank* and *SRank* have on the detection ability of the top-ranked assertions. We consider the assertions in Table 6.10 for the $usbf\_pd$ module.

6.8.6 Case study VI

Observation: An assertion with a good bug detectability amounts to containing important design variables and a broad scope that makes *IRank* and *SRank* to rank it higher.

Example and insight analysis: We show a case where the assertions with good bug detectability were ranked at the top by both *IRank* and *SRank*. For each of the *IRank* and *SRank*, we consider three top-ranked assertions of Table 6.10 (a25, a38, and a21 for *IRank* and a26, a25, and a20 for *SRank*) for the target variable ***seq_err*** of $usbf\_pd$ module.

Each of the assertions a25, a38, and a21 contain *high-importance* design variables (***state***, ***pid***, ***rx_err***) either in the first cycle or in the second cycle. Following the analysis of Section 6.8.1, presence of high-importance design variables allowed each of the assertions to cover design paths relevant to important design functionality and hence *IRank* ranked them higher. Coverage of such important design paths caused each of the assertions to have good bug detectability. Each one of the injected bugs (bug IDs 1-5 of Table 6.8) were affecting a design path (*e.g.*, state machine state sequencing path affected by bug ID 1) that is relevant to an important functionality. Consequently, each of the top-ranked assertions were able to detect multiple bugs (c.f., column 4 of Table 6.10) **up to 4 bugs (average 3.33 bugs) per assertion**.

Each of the assertions a26, a25, and a20 i) contain variables in the antecedent that are temporally separated (***pid_MDATA***, ***rx_err***, ***pid_IN***) from the target variable (***seq_err***) and ii) contain spatially separated vari-

ables (***state***, ***pid***) that are referenced several statements apart from the assignment to the target variable (***seq_err***). This increases the scope of each of the assertions *i.e.*, it increases the number of design statements that are covered between the satisfaction of the antecedent and the truth of the consequent and hence *SRank* ranked them higher. Following Section 6.8.1, coverage of a large number of design statements implies a possible coverage of important design functionality. Since each of the injected bugs were affecting some important design functionality, the buggy statement was one among the covered statements in each of the cases. Hence, each of the top-ranked assertions were able to detect multiple bugs (c.f., column 7 of Table 6.10) **up to 3 bugs (average 2.67 bugs) per assertion**.

6.8.7 Case study VII

Observation: *IRank*'s prioritization on important design variables causes it to rank assertions with good bug detectability at the top of the list whereas *SRank*'s prioritization on spatio-temporal relationship between the target variable and the variables in the antecedent often causes it to rank assertions with poor bug detectability at the top of the list.

Example and insight analysis: We show a case where *IRank* ranked an assertion higher by identifying the presence of important design variables in the assertion whereas *SRank* ranked an assertion higher with broader scope and poor bug detection ability. For each of the *IRank* and *SRank*, we consider two top-ranked assertions of Table 6.10 (a201 and a18 for *IRank* and a83 and a11 for *SRank*) for the target variables ***pid_cks_err*** and ***pid_ACK*** for the *usbf_pd* module.

Each of the assertions a201 and a18 span across only one cycle compared to a83 and a11 that span across two cycles. This means that the temporal distance between the target variable and the variables in the antecedent are more for a83 and a11 compared to a201 and a18. Higher temporal distance between the target variable and the variables in the antecedent broadens the scope of a83 and a11 causing a83 and a11 to cover more design statements compared to a201 and a18. But just covering more statements is *not sufficient* for bug detection. Each of the assertions a201 and a18 contains high-importance design variable (***state***) which ensures that those two as-

sertions cover important design paths critical to design functionality. As a result, both of them were able to detect the injected bugs (specifically bug ID 1 and bug ID 3 of Table 6.8).

On the other hand, a83 and a11 contain less important design variables ($\boldsymbol{pid\_SETUP}, \boldsymbol{pid}$) thereby a83 and a11 covers design paths that are not critical to design functionality. In spite of broader scope, a83 and a11 failed to capture important design functionality due to the lack of important design variables. *SRank* lacks systematic identification of important design variable and prioritizes the spatio-temporal relationship between the target variable and the variable(s) in the antecedent. Consequently, *SRank* fails to detect an assertion's poor detectability.

These case studies show that *IRank* is consistent with respect to the design functionality that makes top-ranked assertions from *IRank* more valuable than the top-ranked assertions from *SRank* for design comprehension and verification/validation.

6.9 Conclusion

In conclusion, we have demonstrated an effective and computationally efficient assertion ranking framework to evaluate assertion quality. Given assertion's widespread usage in industry in the hardware design verification/validation cycle, we believe a comprehensive ranking framework such as the proposed one, is the right step in the direction of objectifying the desired qualities of assertions. While this work does not provide guidelines on how to write high functional coverage assertions, it provides a path to develop assertion rankings for different special-purpose requirements.

CHAPTER 7

SYMPTOMATIC BUG LOCALIZATION FOR FUNCTIONAL DEBUG OF HARDWARE DESIGNS

7.1 Introduction

Pre-silicon functional debugging is widely accepted as one of the "pain points" of verification. During massive industrial-scale design simulation, a huge amount of simulation data is generated. Hence localizing the root cause is tantamount to finding a needle in haystack. Automated localization to any extent is valuable and can significantly expedite debugging and diagnosis.

In this chapter, we present a methodology for automatically localizing root causes (c.f., Problem PR5 of Figure 1.10) of design bugs during functional verification. This method is based on statistical analysis of failing simulation traces to identify the most suspicious code zones in the RTL design. Intuitively, if there are sufficient simulations where an output (or target) fails, there might be some common patterns across the failing runs that are symptomatic of the failure. Such *symptoms*, if inferred, would correspond to common paths that were executed across a statistically significant number of failing runs for that output.[1] If these symptoms are mapped back to the execution paths in the source code, they would reveal the most suspicious parts of the design.

We mine symptoms across failing traces of a given target variable in the form of temporal logic assertions. To map these symptoms back to the source code, we use the notion of statement coverage of an assertion (c.f., Section 6.2.2). Just as assertions cover statements in the design, statements covered by the symptom can be viewed as being in the scope of the symptom. The collective set of symptoms for an output will then correspond to code zones that are the most suspicious zones for debugging the failure

[1]This can be any target variable. In this work, we use target variable and output synonymously.

in that output. Our localized source code is executable, providing a much smaller simulation trace for inspection by the debugger.

We optimize the above algorithm to avoid false positives as follows. In order to get sufficient evidence, we mine symptoms for the same output across multiple simulation traces, where each trace has a different random seed in the constrained random test. We then find common symptoms across the symptoms mined over all simulation traces for that output. This ensures that only few, highly suspicious symptoms are then mapped back to the code.

Our methodology relies on statistical methods to capture the symptoms of the buggy output. Each symptom corresponds to an execution path in the source code. In order to obtain the code fragment that the symptom summarizes, we map the symptom to the source code using a statistical approach. This approach involves applying different input stimuli that stimulate execution paths that cause the symptom to be true. Although a static source code analysis would give an exact mapping of the symptom to the zone of statements causing that symptom, this is not scalable to large designs. Hence, we use a statistical approach as in [29] making this mapping approximate. The reliance on dynamic, statistical methods makes our approach scalable. We sacrifice completeness for scalability. This means although the zones we localize to are highly suspicious with a high probability of accounting for the bug, we cannot provide a guarantee that the zones we do not localize to are bug free. Empirically, we find that all the injected bugs were localized by our methodology.

Intuitively, we would like to localize to a zone that accounts for many bugs (sensitive) and does not mispredict a bug (precise). We empirically show the sensitivity and precision of our localizations by evaluating the localized code zones for a variety of bugs injected into the USB 2.0 design [154]. We achieve *up to 5%* localization and an average localization of *up to 15%* in the source code; our method identifies these as the most suspicious code zones. The corresponding executable has a simulation trace size that is *up to 80%* smaller than the original trace. We demonstrate that the localized statements belong to functionally related zones in the code instead of isolated code fragments. This allows for better debugging. We use *Importance*, defined for software bug localization [126] to evaluate our localized zones. Higher importance implies the zone is highly sensitive and precise for bug localization. We show an importance score of *up to 0.857* in our localized code zones.

Our contributions are as follows.

- To the best of our knowledge, this is the first solution to provide automatic assertion based statistical bug localization for pre-silicon debugging. Our method leverages the massive volumes of simulation trace data that is generated in typical verification environments, to mine accurate symptoms of buggy behavior.
- Our localization is in terms of executable RTL source code, focusing the suspicious zones to a small fraction of the original source code and simulation traces.
- We provide functionally coherent code zones that can assist understanding of the debugger. Since we use dynamic, statistical methods for all phases, our approach is scalable to large designs.

7.2 Preliminaries

We consider a Verilog RTL design $\mathcal{M}$. For the purpose of RTL source code analysis, we consider $\mathcal{M}$ as a Verilog program. A Verilog program is a parallel composition of a set of concurrent processes. Let V be the set of all signals in $\mathcal{M}$.

Definition 24 *A **simulation run** with respect to a given set of constraints $\mathcal{C}$ is a time annotated n cycle sequence of the values of variables from input to output. A **simulation trace** with respect to a set of constraints $\mathcal{C}$ is the set of all simulation runs for all inputs going to all outputs.*

Definition 25 *A **failure run** with respect to a target variable $v \in V$ is a simulation run such that v has a wrong value at the cycle in which it is checked. The target variable v is called a* failing target variable. *A **failure trace** with respect to a target variable v is the set of all failing runs for v.*

Definition 26 *A **failure symptom** with respect to a failing target variable v is a propositional or temporal assertion mined from a failure trace of that target variable. We denote a failure symptom of v as S_v. The failure symptom is of the same form of P as defined in Definition 19. The scope of a symptom S_v is equivalent to the coverage of assertion S_v.*

Definition 27 *A **localized code zone** within a Verilog program $\mathcal{M}$ is the set of statements in $\mathcal{M}$ which are in the scope of a failure symptom (S_v) with respect to a failing target variable v.*

7.3 Bug localization methodology

Figure 7.1 shows the flow of methodology. The debugging methodology iterates on per failing output. The proposed algorithm can be applied for any target variable. Between two successive passes of this iterative algorithm, for a given failure output, we localize a set of highly suspicious statements that debugger needs to investigate further to fix the bug. One pass of the this iterative algorithm consists of four phases – i) design simulation, ii) mining symptoms from a single failure trace, iii) identifying common symptom across multiple failure traces, and iv) mapping common symptoms to corresponding code zone(s).

We use the two-port arbiter of Figure 3.1 and the assertion (c.f., Definition 19 in Chapter 6) **A0**: $(req2 == 1 \wedge gnt\_ == 1)\ \mathcal{X}\ (req1 == 1) \rightarrow (gnt1 == 1)$ of Section 6.2.1 as a running example in this chapter.

7.3.1 Phase 1: Design simulation

In Phase 1, we simulate the design multiple times with a constrained random test bench for a fixed large number of cycles, to create multiple failure traces for a given output. The constrained random test (CRT) contains an *integer seed* that initializes the testbench random number generator in different initial states to generate different random input stimuli in different simulations. In this phase, the attempt is to gather as much evidence in the form of simulation data as possible for a buggy output.

We use monitors to check if a desired output is buggy during a simulation run. We isolate all the failing simulation runs for an output into a failure trace for the next phase.

For the two-port arbiter of Figure 3.1, we run two simulation runs with two different integer seed values for 100 cycles. In each of this simulation run, monitor for the output `gnt1` triggers indicating a mismatch of the value

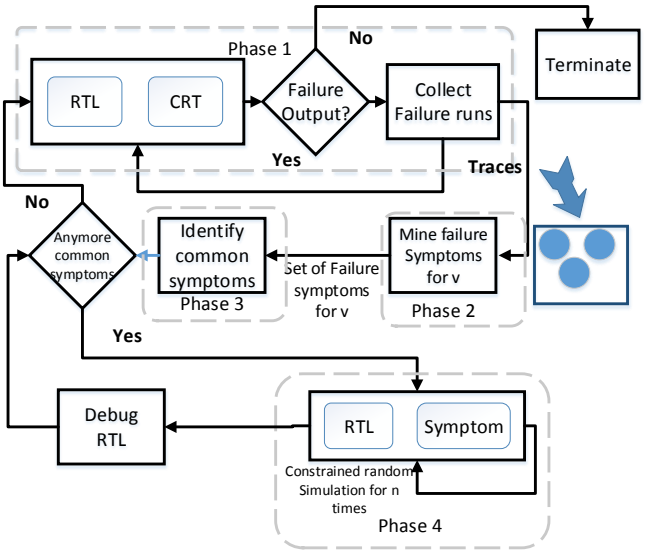


Figure 7.1: Workflow of the proposed debugging approach for a target variable v.

of `gnt1`. We generate a simulation trace from each of these failure runs which forms the set of failure trace of `gnt1` which will be used in Phase 2.

7.3.2 Phase 2: Mining symptoms from a single failure trace

In a sufficiently long simulation trace, a design path is likely to be executed multiple times. Assertion mining engines generate assertions from frequently occurring patterns across multiple design paths in a simulation trace. We repurpose a publicly available assertion mining engine [41] in our context. For every failing output, we provide the assertion miner with a failure trace consisting of all the failing simulation runs for that output. The assertion miner now infers statistically relevant patterns among the frequently executed paths in the failure trace. The resulting assertions are summaries of the frequently occurring behavioral patterns when the output is buggy. These are failure symptoms for that output. Multiple failure symptoms could be generated per output in this phase. In the assertion miner we used, each symptom is internally formally checked, indicating that these are true symptoms. Other assertion mining engines without the formal check can also be used in this phase.

For the two-port arbiter of Figure 3.1, the following symptoms are mined

from failure trace 1 of `gnt1`:

1. $S1.1$: $\mathcal{G}((\neg req2 \wedge gnt\_) \wedge \mathcal{X}(req1) \rightarrow \mathcal{X}(gnt1))$
2. $S1.2$: $\mathcal{G}(\neg req1 \rightarrow \neg gnt1)$
3. $S1.3$: $\mathcal{G}((req1 \wedge req2) \wedge \mathcal{X}(\neg req2) \rightarrow \mathcal{X}(\neg gnt1))$

and from failure trace 2 of `gnt1`

1. $S2.1$: $\mathcal{G}(req1 \wedge req2 \rightarrow gnt1)$
2. $S2.2$: $\mathcal{G}((req1 \wedge req2) \wedge \mathcal{X}(\neg req2) \rightarrow \mathcal{X}(\neg gnt1))$
3. $S2.3$: $\mathcal{G}(\neg req1 \wedge \mathcal{X}(req1) \rightarrow \mathcal{X}(gnt1))$.

From each failure trace of `gnt1`, the assertion mining engine identifies three different suspicious paths occurring frequently in both failure traces.

7.3.3 Phase 3: Identifying common symptom across multiple failure traces

This phase is the optimization step for more sensitive and precise localization. At this point, a set of failure symptoms have been generated for the output for a single failure trace. We repeat this process across multiple failure traces for the same output. We identify *common symptoms* across all sets of failure symptoms generated for the output of interest. Since each symptom summarizes an execution path in the design, multiple symptoms from a single trace might localize to code zones that are not very sensitive or precise. The high number of symptoms per failure trace could also lead to lesser localization, by reporting many code zones as suspicious. We therefore consider only those symptoms that are common across all the failure traces and ranked higher (following the assertion ranking method of Chapter 6) for further analysis. This ensures that the common symptoms we consider are only the most suspicious candidates. This step leverages the already existing large volumes of simulation trace data in industrial settings. Since our method relies on statistical analysis, more data will increase the confidence of our result.

Finding common symptoms amounts to finding common execution paths that could be triggering the bug. The paths would need to be of same length.

Since each symptom is an assertion, we apply the conditions below to find commonality across two assertions. Two assertions S_i and S_j are common if the following occur.

1. The consequent of S_i and S_j have the same signal-value pair for the given output.

2. The temporal delay between successive A_k's (recall antecedent of a symptom is of the form $A = A_0 \wedge \mathcal{X}(A_1) \wedge \mathcal{X}\mathcal{X}(A_2) \wedge \ldots \wedge \mathcal{X}^m(A_m)$ where $\mathcal{X}$ is a delay operator) in the antecedent of S_i and S_j has to be identical. Intuitively, this implies that the two different paths which are identified by that assertions, constitutes a different branch conditions at same delay interval.

3. Each of the A_k's in S_i and S_j should be a conjunction of the same set of propositions (recall a proposition is a signal-value pair where the value can be either "0" or "1"). Intuitively it implies that two paths identified by two assertions essentially constitutes same branch condition in every clock cycle.

For the arbiter of Figure 3.1, two sets of symptoms are mined for output `gnt1` from two different failure traces. Each of the symptoms $S1.1$, $S1.3$, $S2.2$ and $S2.3$ are two cycles long and each of $S1.2$ and $S2.1$ are one cycle long. For $S1.2$ however, the consequent has a signal value pair of $\langle gnt1, 0\rangle$ whereas the consequent of $S2.1$ has a signal value pair of $\langle gnt1, 1\rangle$, violating Condition 1 above. These are not common. Symptoms $S1.1$ and $S2.3$ have a signal value pair $\langle gnt1, 1\rangle$ and symptoms $S1.3$ and $S2.2$ have a signal value pair $\langle gnt1, 0\rangle$. As per Condition 2, each of the symptom pairs has an equal delay severation in between successive A_i's. However, A_0 of $S1.1$ contains the propositions $\{\neg req2, gnt\_\}$ whereas the A_0 of $S2.3$ contains the sole proposition $\{\neg req1\}$ which violates Condition 3. A_0, A_1 of $S1.3$ and $S2.2$ contain exactly the same set of propositions and hence satisfy Condition 3. Since the symptom pair $\langle S1.3, S2.2\rangle$ satisfies all three conditions, this is the only common symptom across the two failure traces.

7.3.4 Phase 4: Mapping common symptoms to functional code zones

A common symptom identified in Phase 3 for a given output implies existence of a highly suspicious path in the design, whose execution causes the given output to fail across multiple traces. In Phase 4, we map back the common symptom for a given output to a particular code zone of the RTL source code. We try to identify all the statements of the RTL source code that are in the scope of the common symptom. Since a symptom is essentially an assertion mined by the assertion mining engine from a failure trace for a given output, we use the definition of the code coverage of assertion as proposed in [29] (c.f., Section 6.2.2 and Figure 6.1). In [29], the above computation is found to be complex if computed statically. Hence, these statements are computed by constrained random simulation of the design under the constraint that antecedent becomes true, until the consequent becomes true. All the statements executed during such a simulation are recorded as covered. This process is repeated multiple times by simulating different paths each time under the same constraints. The simulations are stopped at some pre-decided number of iterations. The constrained random simulation method is scalable but incomplete, since it computes an under-approximation of the set of truly covered statements.

We use a similar method to compute the statements in the scope of a symptom. While we can accurately compute the scope of each symptom that we simulate, we cannot guarantee that all the statements within the scope of the symptom have been simulated. Hence, there is a chance that a bug could lie in a scope that we have not simulated. However, in all our experiments, we have found all the statements in the scope of the symptom. We report these set of localized statements as the most suspicious code zones for further investigation.

In the arbiter, the scope of the symptom $\mathcal{G}((req1 \wedge req2) \wedge \mathcal{X}(\neg req2) \rightarrow \mathcal{X}(\neg gnt1))$ is the suspicious code zone consisting of lines 5, 6, 8, 10, 12, 17 of the Figure 3.1.

Further investigation shows that the bug is in line 14 and that needs to be changed to `gnt1 = req1 & ¬req2`.

For a given output, every common symptom is iteratively mapped to a suspicious code zone. Our experimental analysis shows that a failing output

Table 7.1: List of bugs and bug IDs. DD: Data dependent bugs, CD: Control dependent bugs.

Module Name	No. of Statements	Bugs Injected	Type of Bugs DD / CD
usbf_pa	186	8	4 / 4
usbf_pd	195	4	1 / 3
usbf_idma	234	8	5 / 3
usbf_pe	469	8	1 / 7
usbf_wb	104	4	1 / 1

can be mapped to a single bug in a code zone or multiple bugs in a single code zone, a single bug can affect one or more than one outputs, one or more than one bugs spread across functionally correlated code zones can affect a single output. The algorithm can be invoked after fixing the bug, and will continue until there are no more failing outputs in the simulation phase.

7.4 Experimental setup

Design testbed: We use the publicly available USB 2.0 [154] design to demonstrate our results. In our experiments, each design is simulated 15 times with 15 different integer seed values for 5000 clock cycles. Table 7.1 details the different modules of the USB design and the distribution of the bugs that are injected in different modules. Table 7.2 details several injected bugs, the RTL location of bug injection and the possible symptoms.

Execution platform: All experiments on the USB design modules were run on an Intel Xeon CPU E3-1240 8-core processor running at 3.4 GHz with 16 GB RAM.

7.5 Experimental results

7.5.1 Reduction in failure traces

For debugging, the localized source code alone might not be sufficient, since it does not contain cycle related information. A bug may not manifest in the source code, but might manifest in the sequential behavior. Both source

Table 7.2: Sample bugs injected into different modules of USB design. D: indicates data-dependency bug. C: indicates control dependency bug.

Module Name	**Bug ID**	**Bug Type**	**Bug Detail**
usbf_pa	ID 1	D	Wrong assignment to tx_spec_data causing wrong data to propagate at tx_data
usbf_pd	ID 2	C	Swapped control signals data_done and data_valid_d causing wrong value to rxv1 and propagating wrong rx_data_valid signal
usbf_	ID 7	C	Change of constants in the condition (adr_cb[1:0] == 2'h3) to (adr_cb[1:0] == 2'h0) and (adr_cb[1:0] != 2'h0) to (adr_cb[1:0] != 2'h3) propagating wrong value in wr_last and in word_done forces to store wrong output data even if a complete word is not received. Also sends wrong memory request through mreq request via word_done_r
idma	ID 4	C	Change of logical operator || to && causing wrong assignment in address counter
usbf_pe	ID 2	C	Changed the case condition for PID error re-synchronization which makes data packet ID faulty causing wrong data packet to be sent through idin
usbf_wb	ID 2	D	Changed & to | causing wrong data assignment to wb_req_s1 which in turn causes wrong state transition from state IDLE

Table 7.3: Reduction in simulation length with localized code zone executable as compared to original failure trace length.

Module Name	**Bug ID**	**Target Output**	**Original Simulation** (in cycles)	**Localized Simulation** (in cycles)
usbf_pa	ID 1	tx_data	520	110
usbf_pa	ID 4	tx_data	630	125
usbf_pa	ID 7	tx_valid	510	105
usbf_idma	ID 1	tx_data_st	740	150
usbf_pe	ID 8	idin	755	160

code and simulation are required for effective debugging. Our localized code zone is an executable that can be simulated to generate a much smaller failure trace than the original failure trace. Table 7.3 shows the extent of savings in simulation time by replaying only the localized executable. In each case, we could recreate the failure trace for the given output *within 100 - 160 cycles*, whereas in the original simulation, the first failure of the given output happened well beyond 500 cycles. The smaller failure trace along with the localized statements can assist the debugger to identify sequential bugs.

7.5.2 Functional coherence analysis of localized code zones

We show that our method localizes to functionally coherent code fragments, enhancing the understanding of the human debugger about a failure. We split different modules of USB 2.0 into different functional code fragments as per the specification. In column 3 of Table 7.4, we indicate which of the functional code fragments we injected the bug into. Column 5 shows the failing output. The *Localized Functional Code Fragment* column of Table 7.4 details the functional code fragment that our method localizes to. Our method was able to select *as few as two functional code fragments* as in Bug ID 3 of `usbf_pa`, Bug IDs 4 and 7 of `usbf_idma`. In the case of Bug ID 8 of `usbf_pe`, *as many as six functional code fragments* were selected since this is a subtle bug that is deeply embedded in the design and takes long to propagate to the output `idin`. In each case, the localized code zones are not disconnected fragments, but preserve the functional modularity and integrity of the source code.

7.5.3 Quantitative analysis of localized code zone

In this experiment, we show the extent of localization as the fraction of RTL source code that the human debugger needs to examine as against the entire RTL source code. In the case of Bug ID 5 of `usbf_pa`, the bug is localized to *less than 5%* of the code, but for Bug ID 8 of `usbf_pe`, the bug is localized to *around 29%* of the code. This is due to Bug ID 8 being a subtle sequential bug deeply embedded in the design.

Bug ID 1 of the module `usbf_pa` was injected in the Data Path Mux. Data Path Mux allows data packet ID or 16-bit cyclic redundancy check sum to

Table 7.4: Details of identified bugs and zone mapping.

Module Name	Bug ID	Injected Bug Zone	Localized Functional Code Fragments	Taregt Variable	Code to Investigate
usbf_pa	ID 1	Data Path Muxes	PID Select, Data Path Muxes, CRC Logic, CRC1, CRC2 and WAIT state of state machine	tx_data	25.48%
	ID 3	PID Select	PID Select, Data Path Muxes	tx_data	18.87%
	ID 5	State Machine	Tx Valid Assignment, IDLE state of State Machine	tx_valid_last	4.25%
	ID 4	Data Path Muxes	Data Path Muxes, PID Select, CRC Logic	tx_data	18.75%
	ID 8	CRC Logic	Misc Logic, DATA, WAIT and CRC1 state of the state machine, Data Path Muxes	tx_data	9.75%
	ID 7	CRC1 state of the state machine	CRC Logic, Tx Valid assignment, IDLE, DATA, WAIT and CRC1 state of state machine	tx_valid	16.12%
usbf_pd	ID 2	Data Receiving Logic	Data receiving logic, ACTIVE and DATA state of State Machine Logic	rx_data_valid	18.87%
	ID 4	CRC checking in Data Receiving logic	CRC checking, Active and Data state of State Machine	crc16_err	7.2%
	ID 3	TOKEN state of state machine	Token Decoding logic, CRC logic, TOKEN state of the state machine, Frame number logic	token_valid, crc5_err, frame_no	10.25%
	ID 1	Frame number logic	Token decoding logic, TOKEN state of state machine, Frame number logic	frame_no	7.7%
usbf_idma	ID 7	Rx Logic	Address counter in Misc Logic, Rx Logic	wr_last	10.55%
	ID 4	Address assignment in Misc Logic	Rx Logic, Address Counter in Misc logic	word_done_r	8.18%
	ID 6	Size Counter in Misc Logic	Size counter in Misc Logic, Tx Logic	send_data	6.6%
	ID 1, ID 2, ID 8	Address counter logic	Address Counter logic, Tx Logic, Misc Logic	tx_data_st	18.71%
usbf_pe	ID 2, ID 3	Data PID sequencer	Data PID sequencer, Register file update logic, Buffer decoding allocation check, New buffer address logic, IN and OUT end point logic	idin	27.71%
	ID 4	Current PID decoder	OUT2A, OUT state of state machine, Current PID decoder, Outgoing packet PID assignment, Misc logic	data_pid_sel, token_pid_sel, send_token	11.73%
	ID 5	IN operation logic in Buffer select	IN operation logic, Buffer full / empty logic, New Buffer size logic, Register file update logic	idin	17.71%
	ID 6	Buffer space logic	OUT2A of state machine, Buffer space logic, Endpoint indicator logic, Buffer available check logic	token_pid_sel, send_token	8.47%
	ID 7	Out packet size logic	Out packet size logic, Out end point operation logic, CSR decoding logic, UPDATE2 state of state machine	out_to_small	7.4%
	ID 8	Track logic of control endpoints	Data PID sequencer, Register file update logic, Buffer decoding allocation check, New buffer address logic, IN and OUT end point logic, Track logic of control endpoints	idin	29.86%
usbf_wb	ID 2	Sync WISHBONE request	IDLE, MA_WR and MA_RD states of State Machine, Sync WISHBONE request	ma_req	18.4%
	ID 1	MA_RD state of state machine	State machine logic	ma_req, ma_we	18.35%

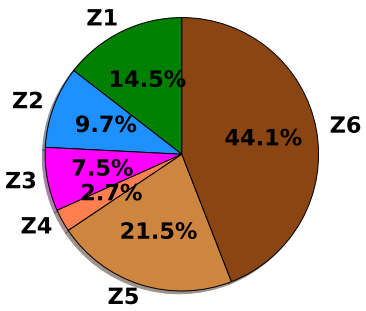
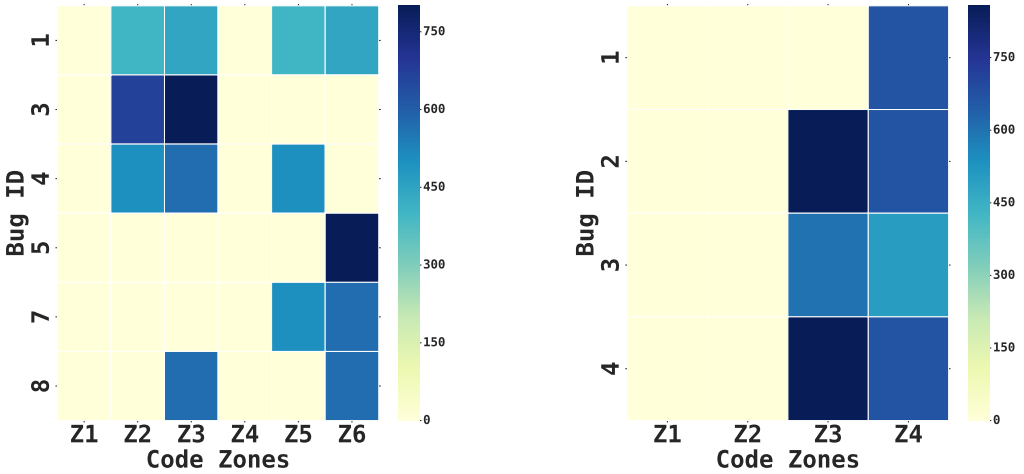


Figure 7.2: Different functional code fragments `usbf_pa` modules. Z1: Misc logic. Z2: PID select. Z3: Data path mux. Z4: Tx valid assignment. Z5: CRC logic. Z6: State machine.

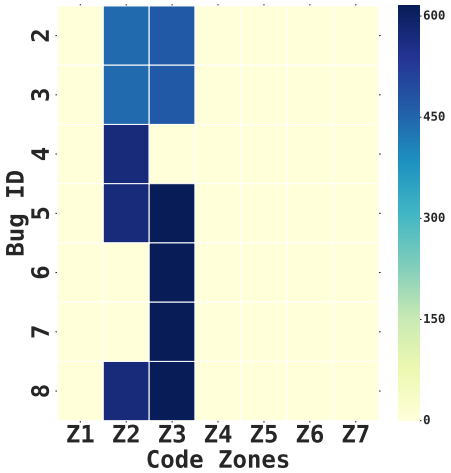
pass depending on whether the USB packet decoding module is in the CRC checking state. The monitor for output `tx_data` failed in Phase 1 of the proposed algorithm. Our algorithm localizes the bug in the functional code fragment of PID select, Data Path Muxes and CRC logic, each of which is a part of the functional code fragment Z1 and a few states of the state machine which is a part of the functional code fragment Z6 of Figure 7.2. Although, `usbf_pa` has six different major functional code fragments, our method successfully discarded two of them and outputs only related four code fragments for further investigation. Further analysis shows that the code contained in Z1 and Z6 account for the 58.6% (c.f., Figure 7.2) of the total RTL code of `usbf_pa` but the last column of Table 7.4 shows we have to only check 25.48% of the RTL code. Our method eliminates 33.12% code from Z1 and Z6 achieving further localization.

We also note two interesting bug scenarios shown in Table 7.4. Bug IDs 1, 2 and 8 of `usbf_idma` cause a single output `tx_data_st` to fail. We identified two different symptoms across multiple failure traces for the output variable `tx_data_st`. One of the symptom localized Bug IDs 1 and 2 and another symptom helped to localize Bug ID 8. Another interesting scenario was Bug ID 4 of `usbf_pe` which simultaneously causes three different outputs namely `data_pid_sel`, `token_pid_sel` and `send_token` to fail. Considering any one failure output for the debugging analysis would do the job. We selected all the three different failed outputs in three independent analyses and were able to locate the bug correctly. The *localized functional code fragment* shown for this case is the union of all the code fragments that were identified by our

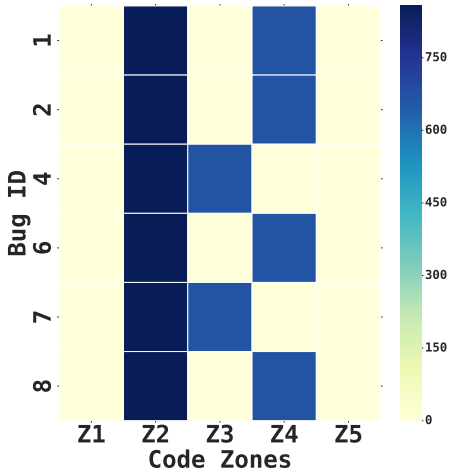


(a) usbf_pa

(b) usbf_pd



(c) usbf_pe



(d) usbf_idma

Figure 7.3: Graphs showing *Importance* of a code zone in identifying a bug on different USB modules. The colorbar on the right side of each graph indicates the numeric *Importance* value of different colors present in different squares. The darker the color the higher the *Importance* of the zone in identifying a particular bug.

algorithm while treating three different outputs independently.

7.5.4 Specificity and sensitivity analysis of localized code zones

We use *Importance* of a code zone [126] as a metric to evaluate our localization. Importance combines sensitivity and precision is by computing their harmonic mean. For example, from Table 7.4, for the module `usbf_pa` we note that Zone 3 i.e. Data Path Mux Logic localizes two different bugs in the code and hence its sensitivity is very high. Further, Data Path Mux logic appears with three other code zones for the Bug ID 1 and hence the precision of Data Path Mux Logic w.r.t Bug ID 1 is $\frac{1}{4}$. Hence, the *Importance* of Data Path Mux Logic w.r.t Bug ID 1 is $\frac{2}{\frac{1}{2}+4} = \frac{4}{9}$. In Figure 7.3, we graphically represent the importance of each of the code zone w.r.t each of the bug we identified. Darker the color of a square, higher is its importance to a particular bug.

7.6 Conclusion

State-of-the-art debugging tools like Synopsys Verdi and Cadence Simvision aid visualization, but do not provide bug localization. To the best of our knowledge, we present the first automated, efficient assertion-based solution to aid RTL debugging through bug localization. We believe that the proposed debugging method will automate and expedite an otherwise tedious and manual RTL functional debugging.

CHAPTER 8

CONCLUSION

In this dissertation, our objective was to provide automation in the unsystematic, ad hoc, and manual SoC validation flow. To achieve this objective, we proposed scalable and vertically integrated solutions for SoC validation.

We have proposed scalable, efficient and effective hardware tracing that can be applied at the different level of design abstraction. We depart from the netlist-level abstraction of prior art and apply our hardware tracing solution at the behavioral level and at the application level of a SoC. Applying hardware tracing at higher design abstraction enabled us to scale hardware tracing to designs containing more than a million flip-flops which is beyond the capacity of the state-of-the-art hardware tracing solutions. We showed that our hardware tracing techniques are computationally efficient and selects high-quality and high-information content signals that are valuable for failure diagnosis.

We have also developed a machine-learning-based post-silicon debug and diagnosis solution. We pose post-silicon debug and diagnosis problem as an outlier detection problem. We engineered two generic features that are highly relevant to the diagnosis task to characterize a post-silicon buggy execution. We used our engineered features to transform raw trace data to the engineered feature space to demarcate normal design behavior from buggy behavior. We have shown that our solution is scalable, effective, and improves debugging by diagnosing many more bugs at a fraction of time as compared to manual debugging.

To improve the quality of assertion-based verification, we have presented an automated assertion ranking technology that analyzes hardware source code and ranks a set of assertions based on their design functionality coverage. We have shown that our ranking methodology is computationally efficient, ranks an assertion higher that has high functional coverage and cover important design paths, and top-ranked assertions have high bug detectability.

Due to a rapid changing design paradigm and design complexity, functional debug of contemporary hardware designs are becoming increasingly difficult. To aid functional debug we have proposed an assertion-based automated bug localization technique for RTL. Our technique is based on identifying statistically relevant common symptoms across failing simulation traces through mining, and mapping these back to the corresponding execution paths in the RTL source code. We showed that our technique can localize to small, focused, functionally coherent code zones that can expedite debugging.

In summary, we have presented a suite of techniques for functional validation of industrial-scale SoCs that are a significant departure from traditional pre-silicon and post-silicon validation. We have overthrown the idea of gate-level analysis for SoC post-silicon validation and instead of going *bottom-up* we have emphasized a *top-down* perspective. We have shown with conclusive empirical evidence that going forward, application-level analysis is the key to scale post-silicon validation to industrial-scale SoCs. The techniques that are proposed in this dissertation are the first step to bridge the widening gap between academic research and the present and future requirements for industrial scale SoC post-silicon validation. Our proposed techniques can bring order in an otherwise chaotic SoC validation paradigm and introduces automation in current unsystematic, ad hoc, and manual settings. Finally, our proposed validation flow can plug into the current industrial validation process but can provide multiple order of magnitudes of benefit.

CHAPTER 9

RESOURCES

In this chapter, we discuss how to obtain and use various tools that are developed as part of this dissertation.

9.1 PRoN: Hardware tracing tool for netlist-level and behavioral-level designs

The PageRank based hardware tracing tool *i.e.*, PRoN can be downloaded from `https://gitlab.engr.illinois.edu/dpal2/tcad_journal_iccad_15_t2_syn/tree/master/ICCAD15_Extension` and [52].

9.1.1 Software requirements

The PRoN tool has been implemented using Python 2.7.x and Perl 5.40.x. One can use the synthesized OpenSPARC T2 netlists to conduct new experiments. Otherwise, one would need Synopsys Design Compiler [189] to synthesize a design to gate-level netlist. For synthesis we use NanGate 45 nm library [160]. Our signal selection tool expects a netlist in ISCAS89 format. We provide additional scripts that convert DC synthesized list to standard ISCAS89 format. These conversion scripts can only work for the NanGate 45 nm library.

9.1.2 Quick start

The PRoN tool is available in the *python_code* directory and can be run using the following command.

python python_code/iscas89_pagerank_ana.py ⟨design_netlist⟩

The tool outputs a list of flip-flop signals sorted according to their enhanced PageRank score. A comprehensive run script to run PRoN and other state-of-the-art tool is available in as *python_code/run_all_netlisy.py*.

9.1.3 Details of directory hierarchy of the repository

- **perl_code**: The directory contains Perl scripts to convert synthesized Design Compiler netlists to ISCAS89 format.
- **python_code**: The directory contains the primary PRoN tool and additional auxiliary Python codes for various purposes.
- **scripts**: The directory contains example Tcl scripts for synthesizing different OpenSPARC T2 modules. These can be reused if a different technoilogy library is used for synthesis.
- **synthesized_netlist**: The directory contains the synthesized netlists and ISCAS89 format netlists for a wide variety of OpenSPARC T2 design modules.
- **testbenches**: The directory contains constrained random testbenches for various OpenSPARC T2 design modules for simulation.

9.2 Application-level hardware tracing tool

The application-level message selection tool has two different repositories.

The first repository contains different buggy OpenSPARC T2 SoC designs that can be downloaded from `https://gitlab.engr.illinois.edu/dpal2/opensparct2` and [53]. The master branch contains the original OpenSPARC T2 code. Each of the buggy version of the OpenSPACR T2 is available in the same repo as a different branch. To reuse any buggy design, an appropriate branch of this repository needs to be cloned.

The second repository contains the application-level message selection framework and the signal-to-message conversion framework. These can be downloaded from `https://gitlab.engr.illinois.edu/sharma53/post_silicon_protocol_lts/tree/merge_branch` and [53].

9.2.1 Details of the directory hierarchy of the repository

- **ascode**: This directory contains application-level message selection tool and the necessary configuration file. Section 9.2.2 details the steps to run the tool.
- **monitors**: This directory contains the Verilog monitors that monitor interface signals of different IPs, convert them into equivalent messages and store the messages in a trace file for analysis. These Verilog monitors need to be simulated with OpenSPARC T2 design modules for monitoring.
    1. *inbound*: This directory contains monitors for observing signals going from the uncore to the core side of the OpenSPARC T2.
    2. *outbound*: This directory contains monitors for observing signals going from the core to the uncore side of the OpenSPARC T2.
    3. *other_monitors*: This directory contains monitors for observing signals between NCU, NIU, and DMU.

9.2.2 Quick start

The repository contains a sample config.cfg file. To construct an interleaved flow for a set of flows, one needs to specify the following parameters per flow in the config file.

1. *noofinstances*: number of concurrent instances of each of the flows.
2. *procolonodes*: state nodes of each of the participating flows.
3. *protocolatom*: state(s) of a flow that needs to be executed atomically.
4. *protocol*: a Python dictionary that contains source state node, the message that needs to occur and the destination state node. Example of a such dictionary is provided in the config.cfg file.

Also, the user needs to mention which flows to be interleaved and to be used for message selection and the trace buffer width at the beginning of the configuration file.

The steps to run the tool are as follows.

1. Run *lts.py* to create the interleaved flow of all the flows. After completion, this script will dump the resulting interleaved flow in a serialized ltsdump file.

2. Run *msg_sel.py* to select messages. This script will parse the ltsdump file and will output the selected messages for tracing.

9.3 Post-silicon debug and diagnosis tool

The post-silicon diagnosis tool can be downloaded from `https://tinyurl.com/yxmztg5v` and [54].

9.3.1 Software requirements

The tool has been implemented using Python 2.7.x and Perl 5.40.x. Apart from standard Python libraries, it also needs PyOD [181] and Scikit-learn [190]. We have tested our implementation with PyOd version 0.6.8 and Scikit-feature version 1.0.0.

9.3.2 Quick start

The tool can be run using the following command.

```
python src/automated_debugging.py -m anomaly -g 100000
```

where "-m" tells the method to be used and "-g" tells the granularity of parsing the message sequence to create message aggregates. The repository contains a detailed README file.

9.4 GoldMine: Assertion ranking tool

The assertion ranking engine has been implemented as a part of GoldMine assertion generation tool. The assertion ranking engine can rank both automatically generated and manually written assertions. To obtain assertion ranking engine along with GoldMine, navigate to `https://bitbucket.org/`

`goldmines_code/goldmine_new/src/v1.0.1/` and clone the git repository.It can also be downloaded as a tar ball from [191].

9.4.1 Software requirements

- **Python backend libraries**: The assertion ranking engine has been implemented in Python 2.7.x. In order to run it properly, it needs additional library support. We have detailed the different Python libraries along with their version against which GoldMine was tested at [192].
- **Verilog simulator**: GoldMine is designed to use both Synopsys [193] VCS and IVerilog [194] Verilog simulator to generate random simulation trace data for the data mining engines. GoldMine can also accept value change dump (VCD) format trace file on the command line if either of the Verilog simulators is not available.
- **Formal verification engine**: GoldMine uses Cadence IFV [152] to formally verify the assertions it generates. GoldMine will label an assertion *unverified* if IFV is not available. The assertion ranking engine will rank all generated assertions regardless of their formal verification status.

9.4.2 Quick start

GoldMine assertion ranking engine can be executed using the following command.

goldmine [options] ⟨input_files⟩

For example, one can use the following command to generate and rank assertions for a two-port arbiter that is available as *install/verilog/Arbiter/arb2.v*.

goldmine verilog/arb2.v

GoldMine will parse the input Verilog design files, simulate a random testebench (if a VCD file is not supplied at the command line) and will generate a set of assertions for the design top module. GoldMine will store all its analysis outputs, different graphs, and the intermittent scripts inside a

goldmine.out/⟨top_module⟩ directory inside the run directory. The meaning of different files and directories that are relevant to assertion ranking are given below.

- ⟨**top_module**⟩**/static**: ⟨*top_module*⟩*.def* contains the variable definition chain details, ⟨*top_module*⟩*.use* contains the variable-use chain details. ⟨*top_module*⟩*.dep* shows the edge weights of the global variable dependency graph, ⟨*top_module*⟩*.rank* shows global importance score of each of the design variables, and ⟨*top_module*⟩*.path* contains the different paths in the design per procedural block.

  The *cdfg* directory contains the control-data flow graph of each of the procedural block that can help in design understanding and in debugging. The *cone* directory contains the relative dependency graph per output for a specified temporal length. This graph is used for relative importance and relative complexity score calculation per output. *var_dep_graph* contains the global variable dependency graph and the graph summary.

- ⟨**top_module**⟩**/verif**: This directory contains a sub-directory for each of the mining engine used. Within each of these sub-directories, there are directories named after the outputs. Each such directory contains two files, ⟨*output_name*⟩*.gold* and ⟨*output_name*⟩*.cone*.

  The ⟨*output_name*⟩*.gold* file contains all the GoldMine generated assertions sorted according to the IRank score along with their importance and complexity score. The ⟨*output_name*⟩*.cone* file contains the relative importance and relative complexity score of each of the variables in the cone-of-influence of an output.

9.4.3 GoldMine command-line options and configuration

GoldMine is highly customizable using a configuration file. Detail of the different parameters in the configuration file and a detailed description of all of the command line options can be found at [195].

REFERENCES

[1] P. Mishra, R. Morad, A. Ziv, and S. Ray, "Post-silicon validation in the SoC era: A tutorial introduction," *IEEE Design & Test*, vol. 34, no. 3, pp. 68–92, 2017. [Online]. Available: https://doi.org/10.1109/MDAT.2017.2691348

[2] W. Chen, S. Ray, J. Bhadra, M. S. Abadir, and L. Wang, "Challenges and trends in modern SoC design verification," *IEEE Design & Test*, vol. 34, no. 5, pp. 7–22, 2017. [Online]. Available: https://doi.org/10.1109/MDAT.2017.2735383

[3] "Qualcomm Snapdragon 855 SoC," https://www.qualcomm.com/products/snapdragon-855-mobile-platform.

[4] "Samsung Exynos 9820 SoC," https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-9-series-9820.

[5] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, 2009, pp. 825–885. [Online]. Available: https://doi.org/10.3233/978-1-58603-929-5-825

[6] H. Foster, D. Lacey, and A. Krolnik, *Assertion-based design*, 2nd ed. Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[7] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*. [Online]. Available: https://doi.org/10.1145/2744769.2744921, pp. 48:1–48:6.

[8] "Verilog hardware description language," http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1620780.

[9] "VHDL hardware description language," https://ieeexplore.ieee.org/document/4772740.

[10] "SystemVerilog LRM," http://standards.ieee.org/getieee/1800/download/1800-2012.pdf.

[11] S. Mitra, S. A. Seshia, and N. Nicolici, "Post-silicon validation opportunities, challenges and recent advances," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837280 pp. 12–17.

[12] P. Patra, "On the cusp of a validation wall," *IEEE Design and. Test of Computers*, vol. 24, no. 2, pp. 193–196, 2007.

[13] S. Yerramilli, "Addressing post-silicon validation challenge: Leverage validation and test synergy," in *Keynote, Intl. Test Conf.*, 2006.

[14] E. Singerman, Y. Abarbanel, and S. Baartmans, "Transaction based pre-to-post silicon validation," in *Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011*. [Online]. Available: http://doi.acm.org/10.1145/2024724.2024855, pp. 564–568.

[15] R. Fraer, D. Keren, Z. Khasidashvili, A. Novakovsky, A. Puder, E. Singerman, E. Talmor, M. Y. Vardi, and J. Yang, "From visual to logical formalisms for SoC validation," in *Twelfth ACM/IEEE MEMOCODE 2014, Lausanne, Switzerland, October 19-21, 2014*. [Online]. Available: https://doi.org/10.1109/MEMCOD.2014.6961855, pp. 165–174.

[16] M. Talupur, S. Ray, and J. Erickson, "Transaction flows and executable models: Formalization and analysis of message passing protocols," in *FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*, pp. 168–175.

[17] J. Keshava, N. Hakim, and C. Prudvi, "Post-silicon validation challenges: How EDA and academia can help," in *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837278, pp. 3–7.

[18] J. W. O'Leary and D. M. Russinoff, "Modeling algorithms in SystemC and ACL2," in *Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and Its Applications, Vienna, Austria, 12-13th July 2014*. [Online]. Available: https://doi.org/10.4204/EPTCS.152.12, pp. 145–162.

[19] D. M. Russinoff, "A mathematical approach to RTL verification," in *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. [Online]. Available: https://doi.org/10.1007/978-3-540-73368-3_2, p. 2.

[20] D. M. Russinoff, "A case study in fomal verification of register-transfer logic with ACL2: The floating point adder of the AMD athlon$^{\text{tm}}$ processor," in *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*. [Online]. Available: https://doi.org/10.1007/3-540-40922-X_3, pp. 3–36.

[21] A. Slobodová, J. Davis, S. Swords, and W. A. H. Jr., "A flexible formal verification framework for industrial scale validation," in *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July, 2011*. [Online]. Available: https://doi.org/10.1109/MEMCOD.2011.5970515, pp. 89–97.

[22] A. Slobodová, "Formal verification methods for industrial hardware design," in *SOFSEM 2001: Theory and Practice of Informatics, 28th Conference on Current Trends in Theory and Practice of Informatics Piestany, Slovak Republic, November 24 - December 1, 2001, Proceedings*. [Online]. Available: https://doi.org/10.1007/3-540-45627-9_10, pp. 116–135.

[23] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of model checking*. Springer, 2018. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8

[24] S. Jha, Y. Lu, M. Minea, and E. M. Clarke, "Equivalence checking using abstract BDDs," in *Proceedings 1997 International Conference on Computer Design: VLSI in Computers & Processors, ICCD '97, Austin, Texas, USA, October 12-15, 1997*, 1997. [Online]. Available: https://doi.org/10.1109/ICCD.1997.628891, pp. 332–337.

[25] "A computational logic for applicative common LISP (ACL2)," http://www.cs.utexas.edu/users/moore/acl2.

[26] "PVS specification and verification system," http://pvs.csl.sri.com.

[27] E. M. Clarke and O. Grumberg, "The model checking problem for concurrent systems with many similar processes," in *Temporal Logic in Specification, Altrincham, UK, April 8-10, 1987, Proceedings*, 1987. [Online]. Available: https://doi.org/10.1007/3-540-51803-7_26, pp. 188–201.

[28] H. Chockler, O. Kupferman, and M. Y. Vardi, "Coverage metrics for temporal logic model checking*," *Formal Methods in System Design*, vol. 28, no. 3, pp. 189–212, 2006. [Online]. Available: https://doi.org/10.1007/s10703-006-0001-6

[29] V. Athavale, S. Ma, S. Hertz, and S. Vasudevan, "Code coverage of assertions using RTL source code analysis," in *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*. [Online]. Available: http://doi.acm.org/10.1145/2593069.2593108, pp. 61:1–61:6.

[30] L. Liu, D. Sheridan, W. Tuohy, and S. Vasudevan, "Towards coverage closure: Using GoldMine assertions for generating design validation stimulus," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, March 2011, pp. 1 –6.

[31] H. Chockler, O. Kupferman, R. P. Kurshan, and M. Y. Vardi, "A practical approach to coverage in model checking," in *CAV*, 2001, pp. 66–78.

[32] "Synopsys Zebu," https://www.synopsys.com/verification/emulation.html.

[33] "Cadence Palladium Z1 enterprise emulation system," https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/acceleration-and-emulation/palladium-z1.html.

[34] "Veloce 2 emulator," https://www.mentor.com/products/fv/emulation-systems.

[35] L. Liu and S. Vasudevan, "Efficient validation input generation in RTL by hybridized source code analysis," in *Design, Automation and Test in Europe, DATE 2011, Grenoble, France, March 14-18, 2011*. [Online]. Available: https://doi.org/10.1109/DATE.2011.5763253, pp. 1596–1601.

[36] L. Liu and S. Vasudevan, "STAR: Generating input vectors for design validation by static analysis of RTL," in *IEEE International High Level Design Validation and Test Workshop, HLDVT 2009, San Francisco, CA, USA, 4-6 November 2009*. [Online]. Available: https://doi.org/10.1109/HLDVT.2009.5340179, pp. 32–37.

[37] A. Ahmed, F. Farahmandi, and P. Mishra, "Directed test generation using concolic testing on RTL models," in *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*. [Online]. Available: https://doi.org/10.23919/DATE.2018.8342260, pp. 1538–1543.

[38] A. Ahmed and P. Mishra, "QUEBS: Qualifying event based search in concolic testing for validation of RTL models," in *2017 IEEE International Conference on Computer Design, ICCD 2017, Boston, MA, USA, November 5-8, 2017*. [Online]. Available: https://doi.org/10.1109/ICCD.2017.36, pp. 185–192.

[39] L. Liu and S. Vasudevan, "Automatic generation of system level assertions from transaction level models," *J. Electronic Testing*, vol. 29, no. 5, pp. 669–684, 2013. [Online]. Available: https://doi.org/10.1007/s10836-013-5403-y.

[40] "SystemC high-level description language," https://www.accellera.org/downloads/standards/systemc.

[41] S. Hertz, D. Sheridan, and S. Vasudevan, "Mining hardware assertions with guidance from static analysis," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 32, no. 6, pp. 952–965, 2013. [Online]. Available: https://doi.org/10.1109/TCAD.2013.2241176

[42] J. Malburg, T. Flenker, and G. Fey, "Property mining using dynamic dependency graphs," in *22nd Asia and South Pacific Design Automation Conference, ASP-DAC 2017, Chiba, Japan, January 16-19, 2017*. [Online]. Available: https://doi.org/10.1109/ASPDAC.2017.7858327, pp. 244–250.

[43] S. Vasudevan, D. Sheridan, S. J. Patel, D. Tcheng, W. Tuohy, and D. R. Johnson, "GoldMine: Automatic assertion generation using data mining and static analysis," in *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010*. [Online]. Available: https://doi.org/10.1109/DATE.2010.5457129, pp. 626–629.

[44] M. Boule, J. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," in *8th International Symposium on Quality of Electronic Design (ISQED 2007), 26-28 March 2007, San Jose, CA, USA*, 2007. [Online]. Available: https://doi.org/10.1109/ISQED.2007.38 pp. 613–620.

[45] A. A. Bayazit and S. Malik, "Complementary use of runtime validation and model checking," in *2005 International Conference on Computer-Aided Design, ICCAD 2005, San Jose, CA, USA, November 6-10, 2005*. [Online]. Available: https://doi.org/10.1109/ICCAD.2005.1560217, pp. 1052–1059.

[46] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99. New York, NY, USA: ACM, 1999. [Online]. Available: http://doi.acm.org/10.1145/302405.302467 pp. 213–224.

[47] C. S. Păsăreanu and W. Visser, "Verification of Java programs using symbolic execution and invariant generation," in *Model Checking Software*, S. Graf and L. Mounier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 164–181.

[48] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar, "A technique for invariant generation," in *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS 2001. London, UK: Springer-Verlag, 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=646485.694471 pp. 113–127.

[49] J. Yang and D. Evans, "Dynamically inferring temporal properties," in *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '04. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/996821.996832 pp. 23–28.

[50] D. Wang and J. Levitt, "Automatic assume guarantee analysis for assertion-based formal verification," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1120725.1120963 pp. 561–566.

[51] H. Foster, "Applied assertion-based verification: An industry perspective," *Foundations and Trends in Electronic Design Automation*, vol. 3, no. 1, pp. 1–95, 2009. [Online]. Available: https://doi.org/10.1561/1000000013

[52] "Observability enhancement framework at different abstraction," https://github.com/paldebjit/netlistbehavselection.

[53] "Application-level observability enhancement and failure diagnosis framework," https://github.com/paldebjit/applevelselection.

[54] "Post-silicon failure diagnosis framework," https://github.com/paldebjit/pdebug.

[55] K. Basu and P. Mishra, "Efficient trace signal selection for post silicon validation and debug," in *VLSI Design (VLSI Design), 2011 24th International Conference on*. IEEE, 2011, pp. 352–357.

[56] D. Chatterjee, C. McCarter, and V. Bertacco, "Simulation-based signal selection for state restoration in silicon debug," in *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*. IEEE, 2011, pp. 595–601.

[57] K. Rahmani, P. Mishra, and S. Ray, "Efficient trace signal selection using augmentation and ILP techniques," in *Fifteenth ISQED 2014, Santa Clara, CA, USA, March 3-5, 2014*. [Online]. Available: http://dx.doi.org/10.1109/ISQED.2014.6783318, pp. 148–155.

[58] K. Rahmani, S. Ray, and P. Mishra, "Post-silicon trace signal selection using machine learning techniques," *IEEE Trans. VLSI Syst.*, vol. 25, no. 2, pp. 570–580, 2017. [Online]. Available: https://doi.org/10.1109/TVLSI.2016.2593902

[59] M. Li and A. Davoodi, "A hybrid approach for fast and accurate trace signal selection for post-silicon debug," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2013, pp. 485–490.

[60] M. Li and A. Davoodi, "A hybrid approach for fast and accurate trace signal selection for post-silicon debug," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 33, no. 7, pp. 1081–1094, 2014. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2014.2307533

[61] F. Farahmandi, R. Morad, A. Ziv, Z. Nevo, and P. Mishra, "Cost-effective analysis of post-silicon functional coverage events," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*. [Online]. Available: https://doi.org/10.23919/DATE.2017.7927022, pp. 392–397.

[62] X. Liu and R. Vemuri, "Assertion coverage aware trace signal selection in post-silicon validation," in *20th International Symposium on Quality Electronic Design, ISQED 2019, Santa Clara, CA, USA, March 6-7, 2019*. [Online]. Available: https://doi.org/10.1109/ISQED.2019.8697793, pp. 271–277.

[63] S. Ma, D. Pal, R. Jiang, S. Ray, and S. Vasudevan, "Can't see the forest for the trees: State restoration's limitations in post-silicon trace signal selection," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015, Austin, TX, USA, November 2-6, 2015*. [Online]. Available: https://doi.org/10.1109/ICCAD.2015.7372542, pp. 1–8.

[64] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proceedings of the Seventh International Conference on World Wide Web 7*, ser. WWW7. Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V., 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=297805.297827 pp. 107–117.

[65] "GoldMine assertion generator," http://goldmine.csl.illinois.edu.

[66] L. Liu, C. Lin, and S. Vasudevan, "Word-level feature discovery to enhance quality of assertion mining," in *2012 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2012, San Jose, CA, USA, November 5-8, 2012*. [Online]. Available: https://doi.org/10.1145/2429384.2429424, pp. 210–217.

[67] L. Liu, D. Sheridan, V. Athavale, and S. Vasudevan, "Automatic generation of assertions from system level design using data mining," in *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July, 2011*. [Online]. Available: https://doi.org/10.1109/MEMCOD.2011.5970526, pp. 191–200.

[68] Z. Poulos, Y. Yang, and A. G. Veneris, "A failure triage engine based on error trace signature extraction," in *2013 IEEE 19th International On-Line Testing Symposium (IOLTS), Chania, Crete, Greece, July 8-10, 2013*. [Online]. Available: https://doi.org/10.1109/IOLTS.2013.6604054, pp. 73–78.

[69] Z. Poulos and A. G. Veneris, "Clustering-based failure triage for RTL regression debugging," in *2014 International Test Conference, ITC 2014, Seattle, WA, USA, October 20-23, 2014*. [Online]. Available: https://doi.org/10.1109/TEST.2014.7035339, pp. 1–10.

[70] Z. Poulos, Y. Yang, A. G. Veneris, and B. Le, "Simulation and satisfiability guided counter-example triage for RTL design debugging," in *Fifteenth International Symposium on Quality Electronic Design, ISQED 2014, Santa Clara, CA, USA, March 3-5, 2014*. [Online]. Available: https://doi.org/10.1109/ISQED.2014.6783384, pp. 618–624.

[71] A. Becker, D. Maksimovic, D. Novo, M. Ewaida, A. G. Veneris, B. Jobstmann, and P. Ienne, "FudgeFactor: Syntax-guided synthesis for accurate RTL error localization and correction," in *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings*. [Online]. Available: https://doi.org/10.1007/978-3-319-26287-1_16, pp. 259–275.

[72] J. Adler, D. Maksimovic, and A. G. Veneris, "Root-cause analysis for memory-locked errors," in *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*. [Online]. Available: http://ieeexplore.ieee.org/document/7459465/, pp. 1054–1059.

[73] N. Veira, Z. Poulos, and A. G. Veneris, "Suspect set prediction in RTL bug hunting," in *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*. [Online]. Available: https://doi.org/10.23919/DATE.2018.8342261, pp. 1544–1549.

[74] Z. Poulos and A. G. Veneris, "Failure triage in RTL regression verification," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 37, no. 9, pp. 1893–1906, 2018. [Online]. Available: https://doi.org/10.1109/TCAD.2017.2783303

[75] N. Veira, Z. Poulos, and A. G. Veneris, "Suspect2vec: A suspect prediction model for directed RTL debugging," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference, ASPDAC 2019, Tokyo, Japan, January 21-24, 2019*. [Online]. Available: https://doi.org/10.1145/3287624.3287661, pp. 681–686.

[76] "Synopsys Verdi: Automated debug system," https://www.synopsys.com/verification/debug/verdi.html.

[77] D. Pal, S. Ma, and S. Vasudevan, "Emphasizing functional relevance over state restoration in post-silicon signal tracing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, p. 1, 2018.

[78] D. Pal, A. Sharma, S. Ray, F. M. de Paula, and S. Vasudevan, "Application level hardware tracing for scaling post-silicon debug," in *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*. [Online]. Available: https://doi.org/10.1145/3195970.3195992, pp. 92:1–92:6.

[79] D. Pal, Z. Zheng, and S. Vasudevan, "Feature engineering for scalable application-level post-silicon debug," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[80] S. Hertz, D. Pal, S. Offenberger, and S. Vasudevan, "A figure of merit for assertions in verification," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference, ASPDAC 2019, Tokyo, Japan, January 21-24, 2019*. [Online]. Available: https://doi.org/10.1145/3287624.3287660, pp. 675–680.

[81] D. Pal, S. Offenberger, and S. Vasudevan, "Assertion ranking using RTL source code analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, p. 1, 2019.

[82] D. Pal and S. Vasudevan, "Symptomatic bug localization for functional debug of hardware designs," in *29th International Conference on VLSI Design and 15th International Conference on Embedded Systems, VLSID 2016, Kolkata, India, January 4-8, 2016*. [Online]. Available: https://doi.org/10.1109/VLSID.2016.14, pp. 517–522.

[83] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoCs," in *Proceedings of the 43rd Annual Design Automation Conference*, ser. DAC '06. New York, NY, USA: ACM. [Online]. Available: http://doi.acm.org/10.1145/1146909.1146916, pp. 7–12.

[84] H. F. Ko and N. Nicolici, "Combining scan and trace buffers for enhancing real-time observability in post-silicon debugging," in *15th European Test Symposium, ETS 2010, Prague, Czech Republic, May 24-28, 2010*. [Online]. Available: https://doi.org/10.1109/ETSYM.2010.5512781, pp. 62–67.

[85] K. Han, J.-S. Yang, and J. A. Abraham, "Enhanced algorithm of combining trace and scan signals in post-silicon validation," in *VLSI Test Symposium (VTS), 2013 IEEE 31st*. IEEE, 2013, pp. 1–6.

[86] K. Rahmani and P. Mishra, "Efficient signal selection using fine-grained combination of scan and trace buffers," in *VLSI Design and 2013 12th International Conference on Embedded Systems (VLSID), 2013 26th International Conference on*. IEEE, 2013, pp. 308–313.

[87] K. Rahmani, S. Proch, and P. Mishra, "Efficient selection of trace and scan signals for post-silicon debug," *IEEE Trans. VLSI Syst.*, vol. 24, no. 1, pp. 313–323, 2016. [Online]. Available: https://doi.org/10.1109/TVLSI.2015.2396083

[88] K. Basu, P. Mishra, and P. Patra, "Efficient combination of trace and scan signals for post silicon validation and debug," in *2011 IEEE International Test Conference, ITC 2011, Anaheim, CA, USA, September 20-22, 2011*. [Online]. Available: https://doi.org/10.1109/TEST.2011.6139157, pp. 1–8.

[89] K. Han, J.-S. Yang, and J. A. Abraham, "Dynamic trace signal selection for post-silicon validation," in *VLSI Design and 2013 12th International Conference on Embedded Systems (VLSID), 2013 26th International Conference on*. IEEE, 2013, pp. 302–307.

[90] K. Basu, P. Mishra, P. Patra, A. Nahir, and A. Adir, "Dynamic selection of trace signals for post-silicon debug," in *Microprocessor Test and Verification (MTV), 2013 14th International Workshop on*, Dec 2013, pp. 62–67.

[91] F. Refan, B. Alizadeh, and Z. Navabi, "Bridging pre-silicon and post-silicon debugging by instruction-based trace signal selection in modern processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 7, pp. 2059–2070, July 2017.

[92] B. Kumar, K. Basu, A. Jindal, M. Fujita, and V. Singh, "Improving post-silicon error detection with topological selection of trace signals," in *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct 2017, pp. 1–6.

[93] B. Kumar, A. Jindal, M. Fujita, and V. Singh, "Post-silicon observability enhancement with topology based trace signal selection," in *2017 18th IEEE Latin American Test Symposium (LATS)*, March 2017, pp. 1–6.

[94] X. Liu and Q. Xu, "On signal selection for visibility enhancement in trace-based post-silicon validation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 31, no. 8, pp. 1263–1274, 2012.

[95] H. F. Ko and N. Nicolici, "Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 28, no. 2, pp. 285–297, 2009. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2008.2009158

[96] H. F. Ko and N. Nicolici, "Automated trace signals identification and state restoration for improving observability in post-silicon validation," in *Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, 2008*. [Online]. Available: https://doi.org/10.1109/DATE.2008.4484858, pp. 1298–1303.

[97] P. Komari and R. Vemuri, "A novel simulation based approach for trace signal selection in silicon debug," in *34th IEEE International Conference on Computer Design, ICCD 2016, Scottsdale, AZ, USA, October 2-5, 2016*. [Online]. Available: https://doi.org/10.1109/ICCD.2016.7753280, pp. 193–200.

[98] K. Rahmani, P. Mishra, and S. Ray, "Scalable trace signal selection using machine learning," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on Computer Design*. IEEE, 2013, pp. 384–389.

[99] H. F. Ko and N. Nicolici, "Automated trace signals selection using the RTL descriptions," in *2011 IEEE International Test Conference, ITC 2010, Austin, TX, USA, November 2-4, 2010*. [Online]. Available: https://doi.org/10.1109/TEST.2010.5699214, pp. 144–153.

[100] B. Kumar, K. Basu, M. Fujita, and V. Singh, "RTL level trace signal selection and coverage estimation during post-silicon validation," in *2017 IEEE International High Level Design Validation and Test Workshop, HLDVT 2017, Santa Cruz, CA, USA, October 5-6, 2017*, 2017. [Online]. Available: https://doi.org/10.1109/HLDVT.2017.8167464 pp. 59–66.

[101] K. Basu and P. Mishra, "RATS: Restoration-aware trace signal selection for post-silicon validation," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 21, no. 4, pp. 605–613, 2013.

[102] S. Park and S. Mitra, "IFRA: Instruction footprint recording and analysis for post-silicon bug localization in processors," in *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8-13, 2008*. [Online]. Available: https://doi.org/10.1145/1391469.1391569, pp. 373–378.

[103] S.-B. Park, T. Hong, and S. Mitra, "Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA)," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 10, pp. 1545–1558, 2009.

[104] S. Park and S. Mitra, "IFRA: Post-silicon bug localization in processors," in *IEEE International High Level Design Validation and Test Workshop, HLDVT 2009, San Francisco, CA, USA, 4-6 November 2009*. [Online]. Available: https://doi.org/10.1109/HLDVT.2009.5340160, pp. 154–159.

[105] S. Park, A. Bracy, H. Wang, and S. Mitra, "BLoG: Post-silicon bug localization in processors using bug localization graphs," in *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837367, pp. 368–373.

[106] F. M. de Paula, M. Gort, A. J. Hu, and S. J. E. Wilton, "BackSpace: Moving towards reality," in *Ninth International Workshop on Microprocessor Test and Verification, MTV 2008, Austin, Texas, USA, 8-10 December 2008*. [Online]. Available: https://doi.org/10.1109/MTV.2008.22 pp. 49–54.

[107] F. M. de Paula, M. Gort, A. J. Hu, S. J. E. Wilton, and J. Yang, "BackSpaceL: Formal analysis for post-silicon debug," in *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*. [Online]. Available: https://doi.org/10.1109/FMCAD.2008.ECP.9, pp. 1–10.

[108] F. M. de Paula, A. Nahir, Z. Nevo, A. Orni, and A. J. Hu, "TAB-BackSpace: Unlimited-length trace buffers with zero additional on-chip overhead," in *Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011*. [Online]. Available: https://doi.org/10.1145/2024724.2024821, pp. 411–416.

[109] F. M. de Paula, A. J. Hu, and A. Nahir, "nuTAB-BackSpace: Rewriting to normalize non-determinism in post-silicon debug traces," in *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. [Online]. Available: https://doi.org/10.1007/978-3-642-31424-7_37, pp. 513–531.

[110] A. DeOrio, J. Li, and V. Bertacco, "Bridging pre- and post-silicon debugging with BiPED," in *2012 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2012, San Jose, CA, USA, November 5-8, 2012*. [Online]. Available: http://doi.acm.org/10.1145/2429384.2429403, pp. 95–100.

[111] A. DeOrio, D. S. Khudia, and V. Bertacco, "Post-silicon bug diagnosis with inconsistent executions," in *2011 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2011, San Jose, California, USA, November 7-10, 2011*. [Online]. Available: https://doi.org/10.1109/ICCAD.2011.6105414, pp. 755–761.

[112] T. Hong, Y. Li, S.-B. Park, D. Mui, D. Lin, Z. A. Kaleq, N. Hakim, H. Naeimi, D. S. Gardner, and S. Mitra, "QED: Quick error detection tests for effective post-silicon validation," in *Test Conference (ITC), 2010 IEEE International*. IEEE, 2010, pp. 1–10.

[113] D. Lin, T. Hong, F. Fallah, N. Hakim, and S. Mitra, "Quick detection of difficult bugs for effective post-silicon validation," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. IEEE, 2012, pp. 561–566.

[114] D. Lin, T. Hong, Y. Li, F. Fallah, D. S. Gardner, N. Hakim, and S. Mitra, "Overcoming post-silicon validation challenges through quick error detection (QED)," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*. IEEE, 2013, pp. 320–325.

[115] D. Lin, T. Hong, Y. Li, S. Kumar, F. Fallah, N. Hakim, D. Gardner, S. Mitra et al., "Effective post-silicon validation of system-on-chips using quick error detection," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 33, no. 10, pp. 1573–1590, 2014.

[116] D. Lin and S. Mitra, "QED post-silicon validation and debug: Frequently asked questions," in *19th Asia and South Pacific Design Automation Conference, ASP-DAC 2014, Singapore, January 20-23, 2014*. [Online]. Available: http://dx.doi.org/10.1109/ASPDAC.2014.6742937, pp. 478–482.

[117] K. A. Campbell, D. Lin, S. Mitra, and D. Chen, "Hybrid quick error detection (H-QED): Accelerator validation and debug using high-level synthesis principles," in *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*. [Online]. Available: https://doi.org/10.1145/2744769.2753768, pp. 53:1–53:6.

[118] E. Singh, C. W. Barrett, and S. Mitra, "E-QED: Electrical bug localization during post-silicon validation enabled by quick error detection and formal methods," in *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. [Online]. Available: https://doi.org/10.1007/978-3-319-63390-9_6, pp. 104–125.

[119] E. Singh, D. Lin, C. W. Barrett, and S. Mitra, "Logic bug detection and localization using symbolic quick error detection," *CoRR*, vol. abs/1711.06541, 2017. [Online]. Available: http://arxiv.org/abs/1711.06541

[120] "CBMC: A tool for checking ANSI-C programs," https://www.cprover.org/cbmc.

[121] A. Horn, M. Tautschnig, C. G. Val, L. Liang, T. Melham, J. Grundy, and D. Kroening, "Formal co-validation of low-level hardware/software interfaces," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. [Online]. Available: http://ieeexplore.ieee.org/document/6679400/, pp. 121–128.

[122] E. W. Dijkstra, "A constructive approach to the problem of program correctness," *BIT Numerical Mathematics*, vol. 8, no. 3, pp. 174–186, Sep 1968. [Online]. Available: https://doi.org/10.1007/BF01933419.

[123] C. Lin, L. Liu, and S. Vasudevan, "Generating concise assertions with complete coverage," in *Great Lakes Symposium on VLSI 2013 (part of ECRC), GLSVLSI'13, Paris, France, May 2-4, 2013*. [Online]. Available: https://doi.org/10.1145/2483028.2483088, pp. 185–190.

[124] D. Sheridan, L. Liu, H. Kim, and S. Vasudevan, "A coverage guided mining approach for automatic generation of succinct assertions," in *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems, Mumbai, India, January 5-9, 2014*. [Online]. Available: https://doi.org/10.1109/VLSID.2014.19, pp. 68–73.

[125] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, Aug 2016.

[126] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065014, pp. 15–26.

[127] T. M. Chilimbi, B. Liblit, K. K. Mehra, A. V. Nori, and K. Vaswani, "HOLMES: Effective statistical debugging via efficient path profiling," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070506 pp. 34–44.

[128] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: Simultaneous identification of multiple bugs," in *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006*. [Online]. Available: https://doi.org/10.1145/1143844.1143983, pp. 1105–1112.

[129] P. A. Nainar, T. Chen, J. Rosin, and B. Liblit, "Statistical debugging using compound boolean predicates," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*. [Online]. Available: https://doi.org/10.1145/1273463.1273467, pp. 5–15.

[130] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu, "Statistical debugging using latent topic models," in *Machine Learning: ECML 2007, 18th European Conference on Machine Learning, Warsaw, Poland, September 17-21, 2007, Proceedings*. [Online]. Available: https://doi.org/10.1007/978-3-540-74958-5_5, pp. 6–17.

[131] D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz, "Scalable temporal order analysis for large scale debugging," in *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*. [Online]. Available: https://doi.org/10.1145/1654059.1654104,

[132] S. Horwitz, B. Liblit, and M. Polishchuk, "Better debugging via output tracing and callstack-sensitive slicing," *IEEE Trans. Software Eng.*, vol. 36, no. 1, pp. 7–19, 2010. [Online]. Available: https://doi.org/10.1109/TSE.2009.66

[133] B. Liblit, "Automated detection and repair of concurrency bugs," in *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*. [Online]. Available: https://doi.org/10.1007/978-3-642-34188-5_3, p. 3.

[134] P. A. Nainar and B. Liblit, "Adaptive bug isolation," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May*. ACM, 2010.

[135] L. Fei, K. Lee, F. Li, and S. P. Midkiff, "Argus: Online statistical bug detection," in *Fundamental Approaches to Software Engineering, 9th International Conference, Held as Part of the Joint European Conferences on Theory and Practice of Software, Vienna, Austria, March 27-28, 2006, Proceedings*, ser. Lecture Notes in Computer Science, L. Baresi and R. Heckel, Eds., vol. 3922. Springer, pp. 308–323.

[136] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101949, pp. 273–282.

[137] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: Statistical model-based bug localization," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. [Online]. Available: https://doi.org/10.1145/1081706.1081753, pp. 286–295.

[138] Z. Poulos and A. G. Veneris, "Exemplar-based failure triage for regression design debugging," *J. Electronic Testing*, vol. 32, no. 2, pp. 125–136, 2016. [Online]. Available: https://doi.org/10.1007/s10836-016-5577-1

[139] V. Boppana and M. Fujita, "Modeling the unknown! towards model-independent fault and error diagnosis," in *Test Conference, 1998. Proceedings, International*, Oct 1998, pp. 1094–1101.

[140] A. Veneris and I. Hajj, "Design error diagnosis and correction via test vector simulation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 18, no. 12, pp. 1803–1816, Dec 1999.

[141] Y. Chen, S. Safarpour, J. Marques-Silva, and A. G. Veneris, "Automated design debugging with maximum satisfiability," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 29, no. 11, pp. 1804–1817, 2010. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2010.2061270

[142] R. Berryhill and A. G. Veneris, "Efficient suspect selection in unreachable state diagnosis," *Ann. Math. Artif. Intell.*, vol. 82, no. 4, pp. 261–277, 2018. [Online]. Available: https://doi.org/10.1007/s10472-018-9578-x

[143] R. Berryhill and A. G. Veneris, "A complete approach to unreachable state diagnosability via property directed reachability," in *21st Asia and South Pacific Design Automation Conference, ASP-DAC 2016, Macao, Macao, January 25-28, 2016*. [Online]. Available: https://doi.org/10.1109/ASPDAC.2016.7428000, pp. 127–132.

[144] R. Berryhill and A. G. Veneris, "Efficient selection of suspect sets in unreachable state diagnosis," in *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2016, Fort Lauderdale, Florida, USA, January 4-6, 2016.* [Online]. Available: http://isaim2016.cs.virginia.edu/papers/ISAIM2016_Boolean_Berryhill_Veneris.pdf

[145] K. Hsieh, W. Chen, L. Wang, and J. Bhadra, "On application of data mining in functional debug," in *The IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2014, San Jose, CA, USA, November 3-6, 2014*. [Online]. Available: http://dx.doi.org/10.1109/ICCAD.2014.7001424, pp. 670–675.

[146] E. Singh, D. Lin, C. Barrett, and S. Mitra, "Symbolic quick error detection for pre-silicon and post-silicon validation: Frequently asked questions," *IEEE Design & Test*, vol. 33, no. 6, pp. 55–62, 2016. [Online]. Available: https://doi.org/10.1109/MDAT.2016.2590987

[147] E. Singh, K. Devarajegowda, S. Simon, R. Schnieder, K. Ganesan, M. R. Fadiheh, D. Stoffel, W. Kunz, C. W. Barrett, W. Ecker, and S. Mitra, "Symbolic QED pre-silicon verification for automotive microcontroller cores: Industrial case study," *CoRR*, vol. abs/1902.01494, 2019. [Online]. Available: http://arxiv.org/abs/1902.01494

[148] E. Singh, K. Devarajegowda, S. Simon, R. Schnieder, K. Ganesan, M. R. Fadiheh, D. Stoffel, W. Kunz, C. W. Barrett, W. Ecker, and S. Mitra, "Symbolic QED pre-silicon verification for automotive microcontroller cores: Industrial case study," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*. [Online]. Available: https://doi.org/10.23919/DATE.2019.8715271, pp. 1000–1005.

[149] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, ser. SFCS '77. Washington, DC, USA: IEEE Computer Society, 1977. [Online]. Available: http://dx.doi.org/10.1109/SFCS.1977.32, pp. 46–57.

[150] L. Liu, D. Sheridan, W. Tuohy, and S. Vasudevan, "A technique for test coverage closure using GoldMine," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 31, no. 5, pp. 790–803, 2012. [Online]. Available: https://doi.org/10.1109/TCAD.2011.2177461

[151] J. Cendrowska, "PRISM: An algorithm for inducing modular rules," *International Journal of Man-Machine Studies*, vol. 27, no. 4, pp. 349 – 370, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020737387800032

[152] "Cadence Incisive Formal Verifier," https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/incisive-formal-verification-platform.html.

[153] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford University, Technical Report, 1998.

[154] "USB 2.0," 2008, http://opencores.org/project,usb.

[155] "Intel core i7 processors," https://www.intel.com/content/www/us/en/processors/core/8th-gen-processor-family-s-platform-datasheet-vol-1.html.

[156] "OpenSPARC T2 micro-architecture specification Vol 1," http://www.oracle.com/technetwork/systems/opensparc/t2-07-opensparct2-socmicroarchvol1-1537750.html.

[157] "OpenSPARC T2 micro-architecture specification Vol 2," http://www.oracle.com/technetwork/systems/opensparc/t2-08-opensparct2-socmicroarchvol2-1537751.html.

[158] "LC3B processor," https://courses.engr.illinois.edu/ece411/mp/LC3b_ISA.pdf.

[159] "PRoN framework," https://sites.google.com/view/dpal2/tools.

[160] "Nangate FreePDK," http://www.nangate.com/?page_id=2325.

[161] "Valgrind Massif tool," http://valgrind.org/docs/manual/ms-manual.html.

[162] Y. Abarbanel, E. Singerman, and M. Y. Vardi, "Validation of SoC firmware-hardware flows: Challenges and solution directions," in *The 51st Annual DAC '14, San Francisco, CA, USA, June 1-5, 2014*. [Online]. Available: http://doi.acm.org/10.1145/2593069.2596692 pp. 2:1–2:4.

[163] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1948.tb01338.x

[164] "Zoom out and see better: Scalable message tracing for post-silicon SoC debug," 2017, http://hdl.handle.net/2142/98857.

[165] K. P. Murphy, *Machine learning - A probabilistic perspective*, ser. Adaptive computation and machine learning series. MIT Press, 2012.

[166] T. M. Mitchell, *Machine learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[167] M. Goldstein and S. Uchida, "A comparative evaluation of unsupervised anomaly detection algorithms for multivariate data," *PloS one*, vol. 11, no. 4, p. e0152173, 2016.

[168] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.

[169] M. Hutter and M. Zaffalon, "Distribution of mutual information from complete and incomplete data," *Computational Statistics & Data Analysis*, vol. 48, no. 3, pp. 633 – 657, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167947304000842

[170] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[171] "Levenshtein distance," https://en.wikipedia.org/wiki/Levenshtein_distance.

[172] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, April 1950.

[173] B. Schölkopf, J. C. Platt, J. C. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution," *Neural Comput.*, vol. 13, no. 7, pp. 1443–1471, July 2001. [Online]. Available: https://doi.org/10.1162/089976601750264965

[174] M. Amer, M. Goldstein, and S. Abdennadher, "Enhancing one-class support vector machines for unsupervised anomaly detection," in *Proceedings of the ACM SIGKDD Workshop on Outlier Detection and Description*. ACM, 2013, pp. 8–15.

[175] F. Angiulli and C. Pizzuti, "Fast outlier detection in high dimensional spaces," in *Proceedings of the 6th European Conference on Principles of Data Mining and Knowledge Discovery*, ser. PKDD '02. London, UK, UK: Springer-Verlag, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=645806.670167 pp. 15–26.

[176] S. Ramaswamy, R. Rastogi, and K. Shim, "Efficient algorithms for mining outliers from large data sets," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '00. New York, NY, USA: ACM, 2000. [Online]. Available: http://doi.acm.org/10.1145/342009.335437 pp. 427–438.

[177] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "LOF: Identifying density-based local outliers," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '00. New York, NY, USA: ACM, 2000. [Online]. Available: http://doi.acm.org/10.1145/342009.335388 pp. 93–104.

[178] M. Shyu, S. Chen, K. Sarinnapakorn, and L. Chang, "A novel anomaly detection scheme based on principal component classifier," in *Proceedings of the IEEE Foundations and New Directions of Data Mining Workshop, in conjunction with the Third IEEE International Conference on Data Mining (ICDM'03)*, 2003, pp. 172–179.

[179] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation-based anomaly detection," *ACM Trans. Knowl. Discov. Data*, vol. 6, no. 1, pp. 3:1–3:39, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2133360.2133363

[180] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, ser. ICDM '08. Washington, DC, USA: IEEE Computer Society, 2008. [Online]. Available: http://dx.doi.org/10.1109/ICDM.2008.17 pp. 413–422.

[181] Y. Zhao, Z. Nasrullah, and Z. Li, "PyOD: A Python toolbox for scalable outlier detection," *arXiv preprint arXiv:1901.01588*, 2019. [Online]. Available: https://arxiv.org/abs/1901.01588

[182] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar, "Rank aggregation methods for the web," in *Proceedings of the 10th International Conference on World Wide Web*, ser. WWW '01. New York, NY, USA: ACM, 2001. [Online]. Available: http://doi.acm.org/10.1145/371920.372165 pp. 613–622.

[183] J. C. de Borda, *Memoire Sur Les Elections Au Scrutin*. Histoire del'Académie Royale des Sciences, 1784. [Online]. Available: https://books.google.com/books?id=L6UWkgAACAAJ

[184] M. Condorcet, *Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix*. Paris: Imprimerie Royale, 1785. [Online]. Available: https://books.google.com/books?id=L6UWkgAACAAJ

[185] "PCI," https://opencores.org/project,pci.

[186] "Buggy design and testbenches," https://drive.google.com/drive/folders/1YOIjGn-PxorCR_xZTVnk3tuZtqy1_q99.

[187] "Random code mutation engine," https://goo.gl/Tyde4S.

[188] "Kendall-Tau distance," http://theory.stanford.edu/~sergei/slides/www10-metrics.pdf.

[189] "Synopsys Design Compiler," https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html.

[190] "Scikit-feature Python library," https://github.com/jundongl/scikit-feature.

[191] "GoldMine v1.0.1 download," https://drive.google.com/open?id=1ZVJFt1ShHUPbsYKh5K3sDwW9xJKcOkfc.

[192] "Python backend library requirement for GoldMine," https://sites.google.com/view/goldmine-illinois/tool?authuser=0#h.p_ThLe70x0p6Iy.

[193] "Synopsys VCS Verilog simulator," https://www.synopsys.com/verification/simulation/vcs.html.

[194] "IVerilog Verilog simulator," http://iverilog.icarus.com/.

[195] "GoldMine configuration parameters," https://sites.google.com/view/goldmine-illinois/tool?authuser=0#h.p_YHVz0IvvqRb7.