PROTOCOL-DIRECTED TRACE SIGNAL SELECTION FOR POST-SILICON VALIDATION

BY

ABHISHEK SHARMA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Associate Professor Shobha Vasudevan

ABSTRACT

Due to the increasing complexity of modern digital designs using NoC (network-on-chip) communication, post-silicon validation has become an arduous task that consumes much of the development time of the product. The process of finding the root cause of bugs during post-silicon validation is very difficult because of the lack of observability of all signals on the chip. To increase observability for post-silicon validation, an effective silicon debug technique is to use an on-chip trace buffer to monitor and capture the circuit response of certain selected signals during its post-silicon operation. However, because of area limitations for debug structures on chip and routing concerns, the signals that are selected to be traced are a very small subset of all available signals. Traditionally, these trace signals were chosen manually by system designers who determined what signals may be needed for debug once the design reaches post-silicon. However, because modern digital designs have become very complex with many concurrent processes, this method is no longer reliable. Recent work has concentrated on automating the selection of low-level signals from a gate-level analysis. But none of them has ever been able to interpret the trace signals as high-level meaningful debugging information.

In this work, we present an automated protocol-directed trace selection where the guiding force is the set of system-level protocols. We use a probabilistic formulation to select messages for tracing and then further analyze these solutions. This method produces traces that allow a debugger to observe when behavior has deviated from the correct path of execution and localize this incorrect behavior for further analysis. Most importantly, unlike the previous gate-level analysis based methods, this method can be applied during the chip design phase when most of the debug features are also designed. In addition, this method drastically reduces the time needed to select signals, as we automate a currently manual process.

To my family and friends, for their love and support.

ACKNOWLEDGMENTS

I take this opportunity to express my gratitude to everyone who has supported me through the course of my master's study. I am thankful for their guidance and encouragement. I am not sure if I would have been able to make it through without their support.

I would like to express my sincere gratitude to my adviser, Professor Shobha Vasudevan, for the continuous support during my graduate study and research. I am truly grateful for all her help and expertise that she provided for this project. Her motivation and dedication to work played an important role.

I thank Sandip Ray for his help in formulating this problem and sharing his expertise in this area. I would like to express my appreciation to Debjit Pal for his help with tools and hardware verification concepts during this work. Without their help, this work would not have been possible. I would also like to express my heartfelt thanks to wonderful colleagues in my research group for sharing their experience and ideas on various topics.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

DfD	Design for Debug
FPGA	Field-Programmable Gate Array
IC	Integrated Circuit
IP	Intellectual Property
LTS	Labelled Transition System
NoC	Network-on-Chip
RTL	Register Transfer Level
SoC	System-on-Chip

CHAPTER 1

INTRODUCTION

1.1 IC Design Process

The design process for integrated circuits (ICs) starts with an initial customer requirements specification. This specification is driven by the current and future market needs and consists of high-level features and behavior the design should contain and ensure. From these high-level specifications, the design is slowly refined into a final product. Figure 1.1 shows this design process.

From the initial back-of-the-envelope sketches/calculations done by system architects, the design moves to estimation models, which likely consist of various high-level and fast performance models to ensure the design requirements are met. It is during this stage that much of the exploration of the design is done, before it moves onto an abstract model that allows for execution. This stage of design would typically be done in a system-level design language such as SystemC [1] or SystemVerilog [2]. These languages create non-cycle accurate models that can be used for further verification and performance checking. After the abstract model phase, the model is implemented in RTL and can be simulated, or run on other platforms. RTL will typically be run on small FPGA boards for smaller portions of the design or large emulation machines that combine many FPGA chips to run larger portions or even whole designs together. Once the design reaches a certain level of health, it will be laid out and fabricated on silicon, leading to a chip that can be used for post-silicon validation.

Validation happens at every step of the design process to help ensure that bugs are caught as early as possible and to further refine the design. Once a bug is found at any stage in the design process, the design is refined (at the necessary level) and then validated once again. As indicated by the

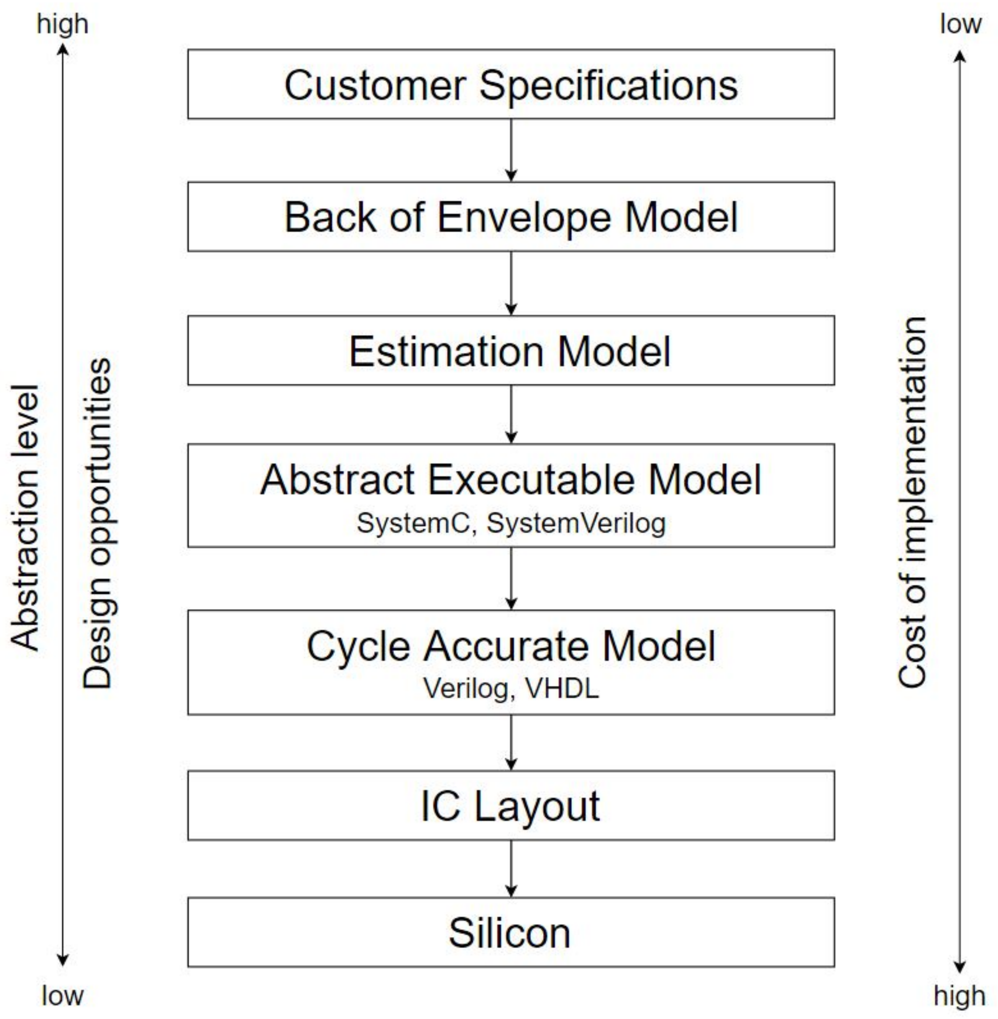

Figure 1.1: Design flow of digital IC

right-hand axis in Figure 1.1, the further down the design flow a design is, the more costly in terms of both time and money it is to make changes. Products have to be delivered in a strict time frame as it affects the market share and potential earnings. The problem is compounded by the left-hand axis, which indicates that the design is easier to change early in the design flow. Both of these factors lead to the need to eliminate bugs as early as possible to keep changes easy to make and, in turn, keep costs down.

1.2 Pre-silicon Verification and Post-silicon Validation

Keeping with Moore's law, integrated circuits are becoming more complex and hence difficult to debug. Recent studies have shown that validation in modern IC designs takes up to 70% of design time in current designs and has become the single most challenging bottleneck of current industry [3]. In order to design and test products in a timely manner, much effort has been put into automated methods to test and check large digital designs. Many of these efforts have been focused in pre-silicon verification, where access to the entire design is available for checking. However, without a substantial increase in the speed at which simulations can be performed or tests can be run on FPGAs or emulation machines, tests for functionality have been pushed to post-silicon where they are faster to run. These tests for functional correctness, in addition to tests for electrical bugs, defined as bugs that appear due to electrical reasons such as crosstalk or vdroop, have made post-silicon validation the bottleneck in many designs [4, 5].

Pre-silicon verification can be defined as the process that is undertaken by verification engineers to ensure the correctness of a design before the design is fabricated into an actual silicon chip. Pre-silicon verification occurs at every phase of design until fabrication, but for the purposes of this work, pre-silicon verification will refer mainly to the verification that is done at the RTL level on simulators, FPGAs, or emulators just before layout and fabrication. Validating for electrical bugs in pre-silicon is limited, and therefore most of pre-silicon verification is concerned with finding and fixing functional bugs, defined as bugs in the specified behavior of the design or its implementation in RTL. At this stage, verification engineers have full observability of the design under verification (DUV), but the number of cycles in which the DUV can be simulated is several orders of magnitude smaller than actual hardware. Therefore, untested cases and corner cases introduce bugs into the post-silicon phase, in which a manufactured chip only has standard inputs and outputs.

Post-silicon validation, as the name suggests, refers to the process undertaken by validation engineers to ensure the correctness of a design after the design is fabricated. It involves operating one or more manufactured chips in actual application environments to validate correct behaviors over specified operating conditions. Post-silicon validation, as the last step in the validation process, must be as exhaustive as possible, testing for both electrical bugs and functional bugs. According to several industry reports, post-silicon validation is becoming significantly difficult and prohibitively expensive because existing techniques cannot cope with the sheer complexity of future systems [6–8]. Both pre-silicon and post-silicon validation have their own challenges and advantages that contribute to the overall effort of validation:

- **Execution Speed**
  Pre-silicon validation tests are run on simulators, emulators, or FPGAs that implement the design's RTL. Simulators are very slow and suitable only for very small portions of the design. FPGAs are able to implement portions of RTL for large designs or even the entire RTL of smaller designs, and are much faster than simulation, up to 3 orders of magnitude faster [9]. Emulators combine many FPGAs to implement the entire (or near entire) RTL for large designs; however, the size of the implementation comes at the cost of speed, as these machines typically run on the order of hundreds of kHz or several MHz compared to hundreds of MHz for a single FPGA [10]. Post-silicon tests are run at full speed on the actual silicon chip, usually on the order of GHz for the whole design. The difference in speed allows more exhaustive tests to be run in post-silicon that touch the entire design. The large speed difference between emulation and post-silicon test means that a test that takes 1 hour in post-silicon would take more than a month to run a on full-chip emulation model. In addition to the execution speed improvement, post-silicon validation usually has the advantage of offering the ability to run full-chip tests on many prototypes at once. Emulators are typically limited to many fewer available prototypes because of the costs associated with each emulator. From all of these factors, we can conclude that post-silicon tests can cover a much larger portion of the design than pre-silicon tests.

- **Observability**
  Pre-silicon validation has the advantage of having a fully observable design, where every signal can be observed during execution. Limited pin-out and other factors in post-silicon limit the observability of internal signals to a small portion of all signals. The limited observability in post-silicon creates a challenge in both detecting bugs and debugging

them. Design-for-debug (DfD) structures on chip attempt to create as much observability as possible, but are still limited by factors including area and routing. Some DfD techniques to improve observability are described later in this chapter.

- **Iterations**
  Once a bug and its root cause are found, it will be fixed and the design will change. In pre-silicon, these changes are relatively low-cost as they only involve changing RTL and the added time to revalidate that portion of the design. However, in post-silicon, the costs of fixing a bug are much higher. The process of fixing the bug can be more involved than in pre-silicon if layout changes need to be made, but the main cost is in the re-spin of the silicon. Once changes are made to the design, it takes weeks or months to receive the next spin of the silicon, which would then need to be validated again. In addition to the increased time cost, the actual monetary cost of creating another spin is not trivial.

Post-silicon validation has significant overlap with pre-silicon design verification and manufacturing (or production) testing. Traditionally, most hardware design bugs are detected during pre-silicon verification, and manufacturing defects are targeted by manufacturing testing. While both manufacturing testing and pre-silicon verification continue to be essential, post-silicon validation is becoming extremely important because of several unique aspects [4,11,12].

1. Pre-silicon verification alone cannot be relied upon to capture all bugs since simulation is several orders of magnitude slower that actual hardware. Popular pre-silicon verification techniques such as constrained random simulation and formal verification all suffer from scalability; therefore, they are infeasible for full-chip verification.

2. As circuits become more and more complex and hard to correctly model, electrical bugs, such as signal integrity (cross-talk, power supply noise) and thermal effects, can only be detected once a chip is manufactured; therefore, it is impossible to test before post-silicon validation.

3. Unlike manufacturing defects, post-silicon bugs may be caused by subtle interactions between a design and physical effects (the so-called

electrical bugs) or by design errors (the so-called logic bugs). It may be very difficult to create accurate and effective fault models for such bugs.

1.3 Validation Process

According to [6], 35% of design time is spent in post-silicon debug. This is a significant portion of time for any design, and in a competitive market the difference in time to market (TTM) can be the difference between a successful and an unsuccessful product. To understand why debug time is currently the bottleneck in the entire TTM, we can refer to the four major steps in the validation process [13].

- Finding a bug: Bugs must be both activated and detected to be found. Test plans for a given design may rely on different coverage statistics to determine when a design has been tested fully.
- Localizing the bug: If a bug is found, the process of localizing it begins. Localizing refers to narrowing the search for the bug to a smaller area of the design. Localizing a bug can involve using debug tools and looking at traces, using a variety of tests and checking the end result, using some flavor of signature checking [14, 15], reducing fault latency in the initial detection [16], and other methods [17, 18].
- Finding the root cause: Finding the root cause of a bug is related to localizing the bug, but in large systems, different techniques may be used to localize and find the root cause.
- Fixing the bug: Once the root cause of a bug is determined, potential fixes can be proposed. In some cases, the fix may be trivial, but in other cases, a fix could necessitate other changes in the design. Once a bug is fixed, the design will need to be tested again to verify that the fix for one bug did not cause more bugs.

Numerous challenges exist in every aspect of post-silicon validation. Instead of a comprehensive survey of all challenges, we highlight a few most important ones as follows:

1. Test pattern generation: In order to check a circuit for functional correctness, it is necessary to verify the internal signal states of the circuit with some golden reference. While tests can be run at-speed on the hardware, test generation and simulation constitute the bottleneck in this process, limiting it to the performance level of pre-silicon simulation. Consequently, design houses are forced to spend enormous computational resources on test generation and simulation servers [18].

2. Reliance on system-level simulation: System-level simulation is several orders of magnitude slower than actual silicon (e.g., 1,000 cycles / second in simulation vs. 1 billion cycles / second for a 1GHz chip) [11]. In order to obtain a golden reference design, system-level simulation is required to achieve correct signal values of every cycle for the entire design. This simulation is very slow; therefore, a functional bug typically takes hours to days to be localized vs. electrical bugs that require days to weeks.

3. Failure reproduction: When a bug is detected, we need to restore the circuit to bug-free state and then re-simulate the circuit with error causing stimuli. Unfortunately, many bugs, such as electrical bugs, and bugs in complex SoCs with multiple clock domains and asynchronous I/Os, are very hard to reproduce [19, 20].

4. Coverage metrics: Code coverage analysis is already a very hard domain of pre-silicon verification. Quantifying coverage of post-silicon validation tests is very challenging due to limited controllability and observability; hence, it is harder to mutate a design or monitor an assertion failure [21, 22].

5. Observability enhancement: A major challenge in post-silicon validation is the limited observability of internal states caused by the limited storage capacity available for post-silicon validation. For one to be able to gather as much data as possible from the DUV in order to understand the nature of the error, design-for-debug (DFD) hardware is commonly inserted into the design. Scan chains and trace buffers are the two most commonly used DFD techniques. However, as the hardware and software interactions between different blocks in an SOC are difficult to verify during pre-silicon verification, it is expected that more DFD

hardware will be deployed in the future. In order to reduce the implementation cycle, while at the same time avoiding an excessive overhead introduced by the DFD hardware, it is becoming a significant challenge to reach the suitable decisions for DFD insertion [18] [23–33, 33–42].

6. Error detection: Long error detection latency, the time elapsed between the occurrence of an error due to a bug and its manifestation as an observable failure, limits the effectiveness of existing bug localization techniques. Simulation is orders of magnitude slower than actual silicon [43]; formal analysis over more than hundreds of cycles can be difficult [44]; and tracing is limited by the availability of on-chip storage [6]. In addition, long error detection latencies may also result in increased error masking, i.e., an error may not propagate to an observable point [16, 45, 46]. Bugs in uncore components of SoCs, such as cache controllers, memory controllers and on-chip networks, can result in very long error detection latencies of several millions to billions of clock cycles unless special attention is paid to shorten these long error detection latencies [47].

7. Bug localization: The process of identifying the location of a detected hardware bug and the cycle(s) during which the bug produces error(s), is a major bottleneck for complex integrated circuits. Among the four major steps we mentioned in this chapter, bug localization dominates the cost [13]. Many post-silicon bug localization techniques can be used, such as IFRA [14, 48, 49] and BLoG [15], simulation-based debug [50, 51], or debug techniques based on formal methods [52–54]. These techniques can directly benefit from the extremely short error detection latencies and improved coverage of QED. With the increasing complexity of uncore components in SoCs, new techniques for localizing bugs inside uncore components are required.

To tackle the aforementioned challenges, many excellent and inspiring works have been proposed. They can be categorized in, but not restricted to, the following fields. In this thesis, we will focus on observability enhancement; specifically, we will propose new methods to select high quality trace buffer signals to help debugging for post-silicon validation.

1. Error detection
    (a) Test suites generation
    (b) Quick error detection
2. Observability enhancement
    (a) Efficient scan/trace signal selection
    (b) Trace signal compression
    (c) Bridging pre-silicon verification and post-silicon validation
3. Bug localization
    (a) Formal method for error localization
    (b) Special on-chip recorder, collecting footprints of execution
    (c) Root cause identification
4. Bug fixing
    (a) Microcode patches or special design techniques
5. Error-resilient system design

1.4 Post-silicon DfD Structures

To combat the problem of low observability in post-silicon debug there are a variety of well-known DfD structures that are inserted into designs that allow validation engineers to observe portions of the design and root cause bugs.

1.4.1 Scan Chains

The primary goal of the scan-based technique is to reuse the internal scan chains, which are placed in the DUV to increase the controllability and observability during manufacturing test [55]. Scan chains are DfD structures created by appending a 2:1 mux to each flip-flop in a portion of the design. The idea behind a scan chain is to halt the design, enable the scan chain,

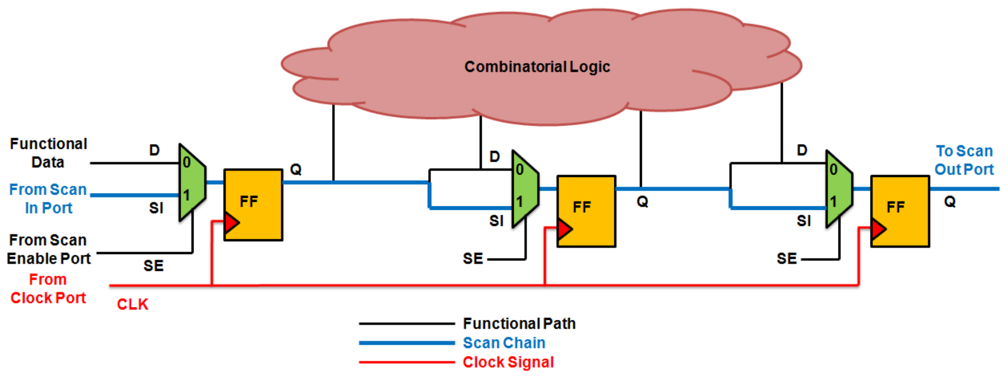

Figure 1.2: Scan chain

then run the clock to output the values of a large set of flip-flops serially on a single output pin. During manufacturing test, the functional pins are used as scan pins for loading multiple scan chains concurrently to reduce test time. However, during debug, these scan chains are concatenated, as shown in Figure 1.2. In this case the scan chain is loaded/unloaded through a serial interface, which is accessible in-system. By capturing data in the internal state elements and offloading them through the scan chains (called scan dump), failure analysis can be performed offline to identify bugs in a design [56].

Scan chains are useful for finding stuck-at bugs, where a given signal is stuck at either a logic 0 or 1, because they allow the capture of a large set of signals. The combinational logic between two signals in a scan chain is known, so they can also be used to detect faulty gates. Scan chains are also used in signature checking schemes and in manufacturing testing. However, before the debug experiment is reproducible, which is not the case for most failures in-field, there is little knowledge about what caused the failure at the observable outputs. Therefore, stopping the debug experiment and re-running it will not guarantee that the failure will occur again. One can let the circuit execution continue by offloading the scan chains through shadow latches, but this may incur a larger area penalty; besides, until a scan dump is completed it is not possible to capture data in consecutive clock cycles. More importantly, the captured data is always done in reaction to an event of interest, and hence it is difficult to record the states that lead to that particular event, which may be crucial during debugging [57, 58]. For this reason, scan changes are not well suited to gain temporal observability in a

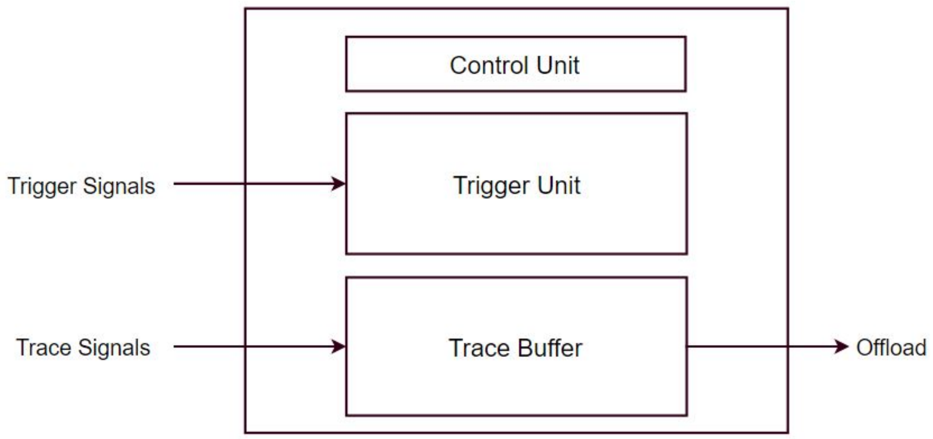

Figure 1.3: Embedded logic analyzer

design. This is usually left to trace buffers

1.4.2 Trace Buffers

An embedded logic analyzer [59], an example of which is shown in Figure 1.3, is divided into four components: control unit, trigger unit, sample unit, and offload unit. The control unit contains one or more finite state machines (FSMs) with programmable registers. The programmable registers can be configured using a serial interface like JTAG for receiving control instructions. This allows the FSMs to control the other units in the ELA to gather different sets of data in multiple experiments. Embedded logic analysis has enabled storing some of the signal states onto an on-chip trace buffer, which can later be used to reconstruct the unknown signals. A trace buffer can be thought of as a storage structure that can hold N bits. A trace buffer is said to have both a *width* and *length*. We will call the number of bits we are allowed to write simultaneously the *width* of the trace buffer. The number of entries in the trace buffer would be the *length*, such that $N = width.length$. Typically, a trace buffer will write a new entry every clock cycle.

The size of the trace buffer, N, must be limited so that the area overhead associated with adding a trace buffer does not become too large compared to the size of the design. In addition, the width of the trace buffer is also limited due to routing concerns and write speed considerations. Therefore, we will always be limited in the width. This limited capacity constrains the number and cycles of signals to be stored; hence, selecting a powerful subset of internal signals becomes one of the most important topics in post-silicon validation.

As mentioned before, scan chain is more like a snapshot of a system moment while trace buffer records more temporal information of the system execution [60]. Scan chain can provide the value of many signals, but only in a short time span, say, one cycle; in contrast, trace buffer can only store a small amount of signals, but each of them can have thousands of cycles. Since this trace buffer is very small, the amount of data that can be collected during a single post-silicon validation run is ultimately limited by the capacity of on-chip trace buffers [61]. Trace buffers are used extensively in industry and usually combined with triggers and other functionalities to create embedded logic analyzers (ELAs). As trace buffers allow a debugger to capture many clock cycles, these are usually preferred for localizing bugs when the location of the bug is still not well known. For further debugging, scan chains or trace buffers at a lower abstraction level may be used.

1.5 Motivation

1.5.1 Current Approaches to Trace Signal Selection

As mentioned before, trace buffers are limited in the number of signals that can be simultaneously observed, which has led to the question of what signals should be prioritized for tracing. Many previous works have focused on increasing the observability of gate-level signals. One metric for determining how well a selection of trace signals improves observability is restoration ratio [25,27–29,33]. This metric is used by selecting a set of flip-flops from a gate-level netlist specification, and then, by knowing the outputs and inputs of certain gates, we can infer the values of other signals within the circuit. For example, if we trace the output value of an AND gate to 1, then we can infer that both inputs are also 1, which may allow us to infer other signals. Other examples of restoration are shown in Figure 1.4. However, this metric fails to include the notion of an error or bug and the essence of the design, so while we may be able to infer a larger set of signals, we are not guaranteed to observe an error or bug in the design with any greater capacity.

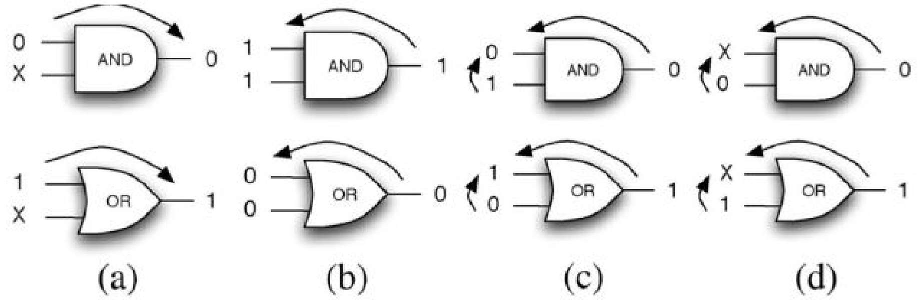

Figure 1.4: Principle operations for state restoration. (a) Forward. (b) Backward. (c) Combined. (d) Not defined.

Other gate-level signal selection methods have used the notion of bugs or errors in their metrics [34, 41, 60, 62–64]. Some of these techniques still rely on restoration to observe these errors, while others do not. In either case, the observability of errors is limited to a gate-level analysis in these methods, which does not provide any information on the expected higher level functionality of the circuit. As a result, these methods usually focus on finding electrical bugs. These electrical bugs usually manifest themselves as some deviation from the specified behavior of the design in the higher abstraction levels, so to find these electrical bugs, we must first localize at a higher level. In addition to localizing electrical bugs, localizing a bug to a small portion of the design such that it can be replicated using pre-silicon is a widely used method for finding functional bugs.

For complete and efficient validation, we must be able to localize bugs at a higher level before using gate-level traces. High-level debug architectures are used within a design to attempt to observe these bugs [6, 12, 17, 65–69]. Many of these architectures introduce run-stop mechanisms that increase the complexity and can be intrusive to the original design. Run-stop mechanisms are also only helpful if the debugger knows what trigger conditions to set to observe a bug. As is the case with many bugs, the erroneous behavior that has occurred may not give any hint about the root cause of a problem and an initial localization is needed so that further debug can begin. This initial debug effort should show system operation at a high level that is easy for the debugger to understand without much effort so that this first-level localization can happen quickly and allow the debugger to move into further localization. Also, the choice of where to place these structures to allow them to trace signals that are important for localizing bugs is currently an ad-hoc procedure where designers attempt to place these structures within the design to the best of their knowledge. The increased use of reusable IP blocks in modern SoC (System-on-chip) designs has made this choice easier for designers as bugs are less likely to appear within the functionality of these areas; however, the communication between IPs has become a riskier area because this changes with each design.

1.5.2 Message Passing Communication

In traditional SoCs, communication has been conducted along a bus or possibly a small number of buses that are connected in some manner. To observe the communication between masters and slaves, one has simply needed to observe the signals on the bus. However, because of the increasing number of IPs used in modern high-end SoC designs and the lack of scalability of buses, designs have shifted to networks-on-chip (NoCs) for IP communication. In this configuration, messages between IPs are packetized and sent along a network of routers and switches until they reach their final destination. Groups of packets along the network will form a message, which will give the receiving IP information that it should react to. The communication between IPs is not as easily observed as it is on a bus because there is no centralized communication point. To observe all possible communication, one would need to observe all incoming and outgoing channels at each IP. A *channel* is the physical, traceable location where portions of each message can be observed as they arrive or leave an IP. However, because communication is done in predefined sequences of messages, called *protocols*, between functional units to perform a specific task, a small subset of all channels may be able to observe a large portion of all expected communication. This subset would be helpful to a debugger and should help observe the maximum amount of bugs that appear in the communication across IPs. In order to know what this subset is, we propose a method that uses protocol specifications to select trace signals. These protocols are specified early in the design process and determine the sequence of messages that should occur to complete a high-level action, similar to a message sequence chart (MSC). MSCs have been used in the past to verify communication protocols and therefore are a natural format from which to extract information when attempting to validate protocols as well [70–72]. We define a textual format to specify MSCs and analyze all protocols and find a subset of messages that allows maximum observability of bugs. The goals of our selection method are:

1. Provide a selection of trace signals that can localize bugs (specific bugs types will be defined later on) by observing the receipt and sending of messages between IPs in an NoC-based digital design.

2. Constrain our trace signal selection to a fixed trace buffer width size. We assume that the length of the buffer is unlimited, as methods exist to either offload trace buffer data in real-time or store the trace buffer data to main memory [73, 74]. While we assume a fixed-width buffer size, compression methods exist that can effectively increase this size, although guarantees on available sizes still need to be made. While these compression techniques allow an overall decrease in trace size, specialized architectures may be needed to utilize this compression to increase the buffer width for our purposes.

1.6 Contributions

To reduce debug time, we propose a method that leverages system-level communication information to create traces that allow for quick localization. Using system-level information allows debuggers to use behavioral specifications as a means for debug and, because of increased IP (intellectual property) usage in modern day SoCs, many bugs appear in the communication between already well validated IP blocks. The contributions of this work are:

1. An automated system-level trace selection method. Previous work in the area of trace signal selection has been focused on gate-level analysis of digital designs, which do not consider the high-level functionality of the design. Our method considers only the high-level functionality from the messages passed between functional units within an NoC.

2. A probability and information theory based formulation of this problem. Currently, this high-level, message-based trace signal selection is done manually by system designers who must somehow determine

what functionality needs to be captured from all correct behaviors of their system. Even a moderately sized SoC design will have many behaviors captured in the messages sent between functional units than a system designer can understand, to accurately select trace signals. As mentioned earlier, communication is done in predefined sequences of messages, called *protocols*, between functional units to perform a specific task. We define protocols in more detail in subsequent chapters. There are a large number of protocols in a system at any point of time, which are inter-leaved with each other. All of the individual protocols might be at different stages in their sequence of messages, and can move ahead independent of each other. This is what we call an inter-leaving of protocols. We define it in more detail later. To be able to localize a bug for debugging, it is important to identify and narrow down on an inter-leaving as much as possible. Our information theory formulation selects trace signals to maximize mutual information over a formal structure representing inter-leaving of different protocols. We then show how these selected signals are beneficial for bug localization and debugging.

Our high-level trace signal selection will allow debuggers to identify the inter-leaving of protocols in the system and narrow down on it. This will allow them to quickly localize bugs in a system using the provided traces. Quickly localizing a bug to a smaller portion of a design is one way to help reduce the overall time spent debugging. Once a bug is localized to a smaller portion of a design, lower level debug structures or simulation can be used to further localize and debug. The narrow protocol inter-leaving can be replayed in simulation where all the signals are visible. And since the scenario being replayed is localized and narrow, it can be simulated in reasonable time.

1.7 Outline

The rest of the thesis is organized as follows:

1. Chapter 2 provides preliminaries of NoC communication between IPs. It describes what are messages, protocols and protocol inter-leavings. It also gives a brief introduction to information theory and mutual

information.

2. Chapter 3 describes our protocol specification format and construction of the structure representing protocol inter-leaving. Information theory formulation of the problem and the method to select trace signals is then presented.

3. Chapter 4 presents results with examples and OpenSPARC T2 SoC from Oracle.

4. Chapter 5 concludes the thesis with a brief discussion of possible future work.

CHAPTER 2

PRELIMINARIES

2.1 Overview

In digital systems with NoC-based communication, communication between functional units is done in the form of packetized messages sent along the network. Messages are sent within *protocols*, which we define as predefined sequences of messages between functional units that perform specific tasks. An example of a simple protocol is shown in Figure 2.1. This protocol defines the messages and the sequence in which they should occur in order to power on the Radio functional unit. We define a functional unit as any design component that receives and/or sends messages on the on-chip network. This term may be used interchangeably with the term *block* throughout the rest of this work for brevity. A reference list of the terms used in this work is presented in Table 2.1.

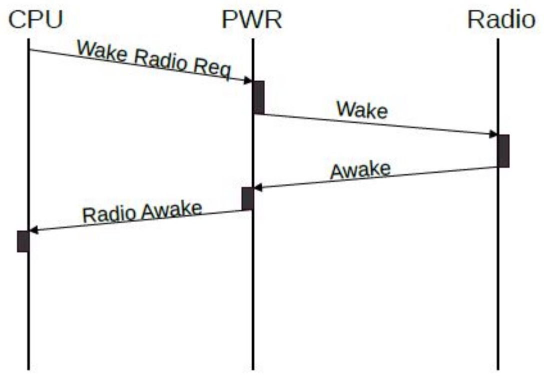

Figure 2.1: Protocol for power-on of radio block from CPU request. The vertical lines represent the three functional units and the lines between them represent messages sent on the interconnection network. Time progresses downward, hence the downward slope of the message lines.

Table 2.1: Reference list of terms

functional unit - a module within a design that is able to send and receive messages to and from other functional units. Also called IP or block.

message - a single unit of information sent from one functional unit to another with header information and one or more payload fields. In this thesis, payload fields are either command, address, or data.

protocol - a predefined sequence of messages between functional units that perform a specific task

protocol family - a group of protocols that achieve the same function, but have different initiators and/or targets

protocol inter-leaving - at any point of execution in the system, there are several protocols which have been initiated by different functional units and are in action. They are referred to as an instance of the protocol. All of these live protocol instances are at different stages in the predefined sequence of messages. These protocol instances at different stages during any point of execution in the system are called a protocol inter-leaving.

2.2 Messages

An example of a message with header and three possible payload fields of command, address, and data is shown in Figure 2.2. The header information is used by the network and contains the source, destination, and other information needed to route the message, while one, two, or all three of the other fields are used by the receiving block. The payload field contains information based on which the receiving block acts, performs certain tasks and may send a message forward to one of the blocks in the system. In this work, we assume the three payload fields of command, address, and data, and we assume each field has a fixed bit length and can be traced independently of the other fields. One or more signals (single bit or word) make up these messages. Furthermore, these messages could be sent over a single clock cycle or could last for multiple clock cycles. Trace signal selection for system level, protocol-based debug in these systems is achieved by selecting a set of messages, or in other words the signals that compose these messages. An illustration of a message and its tracing is shown in Figure 2.3.

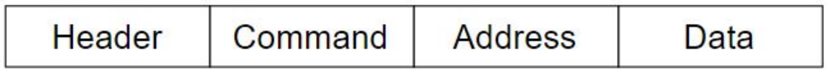

Figure 2.2: Example message format

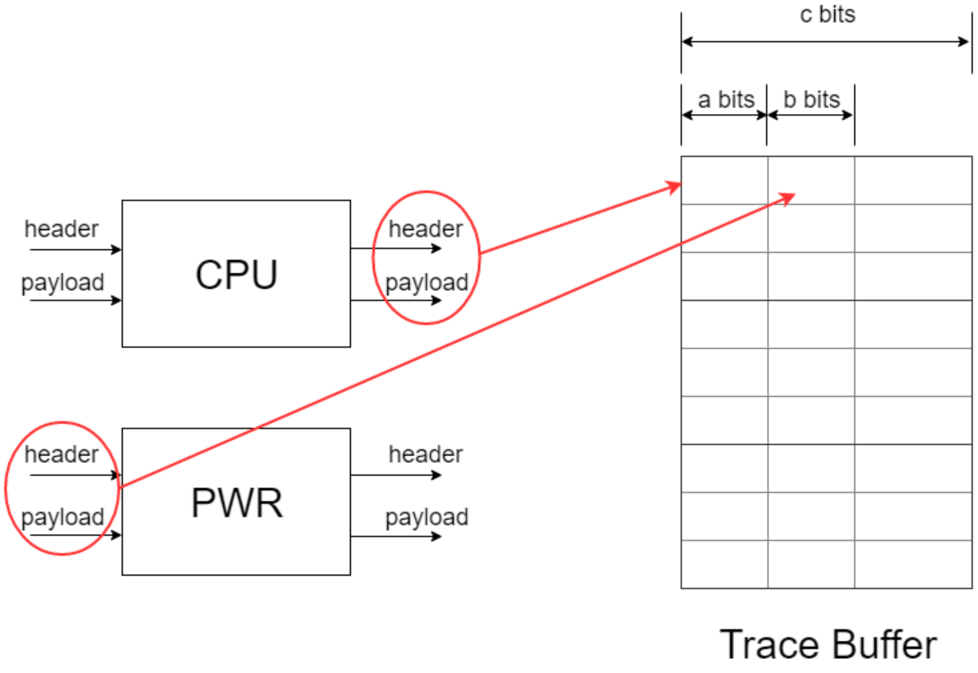

Figure 2.3: Example of message tracing

In the design of NoCs, communication for each block is managed by a network interface (NI) that lies between the functional unit and the network. Each NI would covert incoming packetized messages that flow across the NoC into specific signals that the functional unit can understand, and, vice-versa, convert internal communication signals into packetized messages to send across the NoC. In most cases, this involves some sort of translation between packetized messages to a set of bus signals of a specific bus protocol. One example may be the AHB bus protocol [75]. The NI may communicate with the functional unit by asserting certain signals on the AHB interface. To capture the incoming and outgoing messages to and from a specific functional unit, a designer simply needs to trace the signals within the NI or a simple extension can be added to the NI to make these signals visible. Tracing only a portion of a message is very rarely useful, so in our work we focus on tracing entire incoming or outgoing messages. Each field in a message is always a predefined amount of bits. The number of fields and length of each field vary from message to message and across different NoC architectures. For example, an NoC that has separate high and low power fabrics can denote the difference between these two fabrics by adding more field types. In our problem formulation, we assume the interconnection network does not lose or corrupt messages as they are sent from one functional unit to another. Separate debugging solutions can be used to debug the interconnection network itself for the cases when this is the root cause [17, 65, 76].

2.3 Protocols

Protocols are predefined sequences of messages between functional units that perform specific tasks. Examples of such tasks could be memory read, memory write, cache coherence, interrupt handling, system boot, power management etc. An example of a simple protocol is shown in Figure 2.1, which defines the messages and the sequence in which they should occur in order to power on the Radio functional unit.

2.4 Trace Buffer Architecture

Currently, selecting messages to trace is a manual process undertaken by a system designer. A common approach in trace signals is using a multiplexer to select between different sets of trace signals [37, 60, 62, 77]. Other approaches do so dynamically, but for this trace signal solution, we will assume a multiplexer to allow the debugger to choose between different sets of signals. A system designer or team of system designers use their knowledge of the system's protocols to select sets of messages that can help debug in post-silicon. Each set of messages, or *view*, can be selected together using the selection bits of multiplexers as shown in Figure 2.4. The example shown has only 2 views and n-bits, but typically a system would include more views.

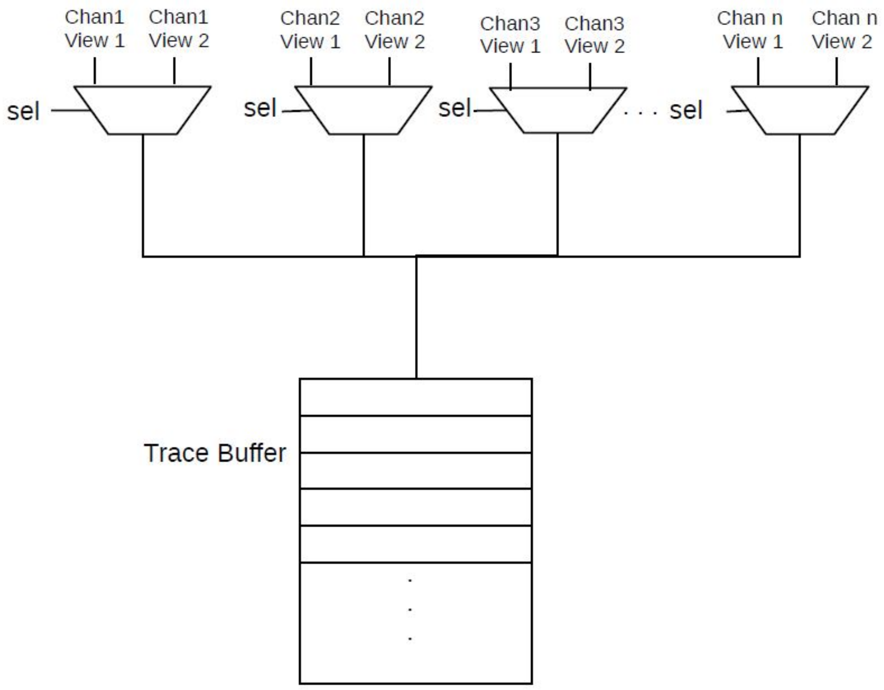

Figure 2.4: Trace buffer architecture for a 2-view, n-bit architecture

In this approach, each view would observe a different portion of the design in such a way that each view will allow a debugger to focus on the messages to and from a specific block within the design. An example of the debug procedure undertaken by a system debugger is shown in Figure 2.5.

This debug methodology is used as a first pass debug that can allow a system level debugger to localize bugs to a single block or a portion of a single block to be further analyzed using block-specific debug structures. However, because the width of the trace buffer is usually less than the number of bits needed for all the messages of even a single block, creating a selection that can allow a post silicon debugger to observe erroneous behavior caused by bugs in the system is an arduous task. The vast number of protocols that are defined in most systems and the subtle communication patterns within those protocols mean that a manual selection is not done in a systematic fashion and, therefore, the quality of selection can be very poor as no definitive statements can be made about its ability to aid in the debug at the system level. In addition, a manual selection takes a considerable time as the system designer or team must carefully look over each and every protocol to attempt

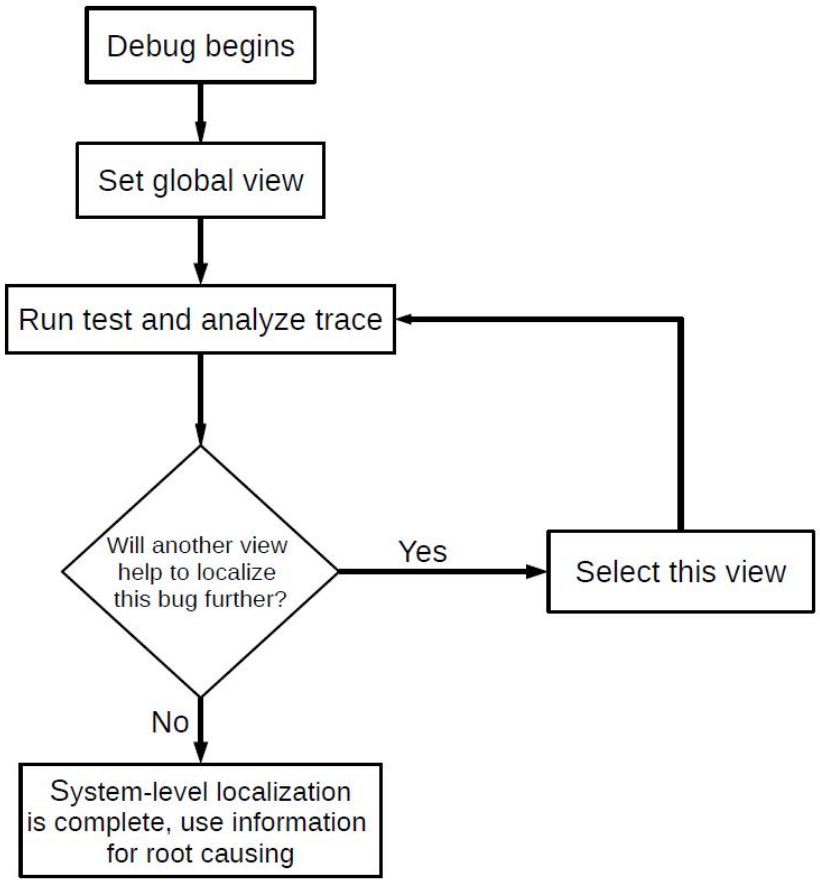
Debug begins
Set global view
Run test and analyze trace
Will another view help to localize this bug further?
Yes
Select this view
No
System-level localization is complete, use information for root causing

Figure 2.5: Usage flow chart for system-level debugger using system-level signal selection with views

to capture the important features. We present an automated, systematic, hierarchical, block-level, selection method for these messages that can aid in identifying and narrowing the protocol inter-leaving and hence localizing the bug. This will be described in detail later. Given a protocol specification and trace buffer width constraint, this selection method will provide trace signals to aid system-level post-silicon debug.

2.5 Information Theory

Information theory studies the transmission, processing, utilization, and extraction of information. Abstractly, information can be thought of as the resolution of uncertainty. Information theory is based on probability theory and statistics. Information theory often concerns itself with measures of information of the distributions associated with random variables. Important quantities of information are entropy, a measure of information in a single random variable, and mutual information, a measure of information in common between two random variables.

A random variable can take on a set of possible different values, each with an associated probability. The mathematical function describing the possible values of a random variable and their associated probabilities is known as a probability distribution. Entropy gives a measure of the uncertainty of the random variable. It is sometimes called the missing information: the larger the entropy, the less a priori information one has on the value of the random variable. Entropy of a discrete random variable X with probability mass function $p(x)$ is defined as

$$H(X) = -\sum_{x\in\chi} p(x)\log_2 p(x) \tag{2.1}$$

In case of a continuous random variable, sum is replaced by an integral and probability mass function by probability density function. The entropy is meant to measure the uncertainty in a realization of X. Now, we want to quantify how much uncertainty does the realization of a random variable X have if the outcome of another random variable Y is known. We define conditional entropy as

$$H(X|Y) = -\sum_{x,y} p(x,y)\log\frac{p(x,y)}{p(y)} = \sum_{y} p(y)H(X|Y=y) \tag{2.2}$$

Here $p(y)$ is the marginal distribution of Y, so $p(y) = \sum_x p(x,y)$. It is easy to see that the conditional entropy can be written as $H(X|Y) = H(X,Y) - H(Y)$, where

$$H(X,Y) = -\sum_{x,y} p(x,y)\log p(x,y) \tag{2.3}$$

Mutual information measures the amount of information that can be obtained about one random variable by observing another. The mutual information of X relative to Y is given by

$$I(X;Y) = \sum_{x,y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)} \tag{2.4}$$

where $p(x, y)$ is the joint probability distribution function of X and Y, and $p(x)$ and $p(y)$ are the marginal probability distribution functions of X and Y respectively. A basic property of the mutual information (Figure 2.6) is that

$$I(X;Y) = H(X) - H(X|Y) = H(X) + H(Y) - H(X,Y) \tag{2.5}$$

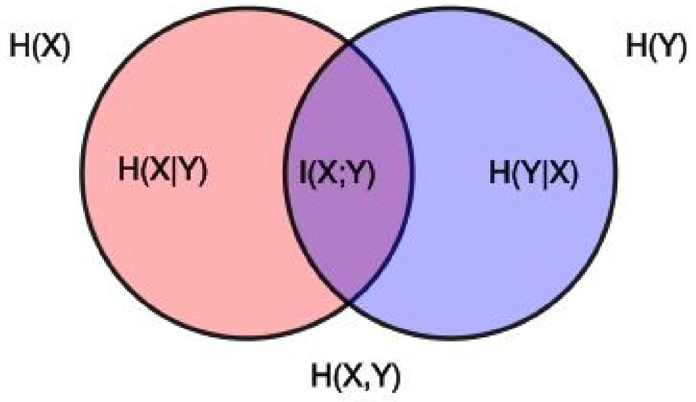

Figure 2.6: Venn diagram for various information measures associated with correlated random variables X and Y. The area contained by both circles is the joint entropy $H(X,Y)$. The circle on the left (red and violet) is the individual entropy $H(X)$, with the red being the conditional entropy $H(X|Y)$. The circle on the right (blue and violet) is $H(Y)$, with the blue being $H(Y|X)$. The violet is the mutual information $I(X;Y)$.

Intuitively, mutual information measures the information that X and Y share: it measures how much knowing one of these variables reduces uncertainty about the other. For example, if X and Y are independent, then knowing X does not give any information about Y and vice versa, so their mutual information is zero. At the other extreme, if X is a deterministic function of Y and Y is a deterministic function of X then all information conveyed by X is shared with Y: knowing X determines the value of Y and vice versa. Mutual information is a measure of the inherent dependence expressed in the joint distribution of X and Y relative to the joint distribution of X and Y under the assumption of independence. Mutual information therefore measures dependence in the following sense: $I(X;Y) = 0$ if and only if X and Y are independent random variables.

In many applications, one wants to maximize mutual information (thus increasing dependencies), which is often equivalent to minimizing conditional entropy. Examples include search engine technology, telecommunications, clustering of data set, Bayesian networks and decision tree learning.

CHAPTER 3

PROTOCOL DIRECTED TRACE SIGNAL SELECTION

3.1 System-on-Chip and Message Specification

A system-on-chip consists of a set of IPs or functional units and has a set of communication fabrics connecting different IP components. An interface across two different IPs is such that at least one communication fabric exists between the IP pair. Each of the IP comprises of a set of interface signals which delivers data/command payload to the communication fabric. The communication fabric consists of routers to route the signals to appropriate destination.

As an example for experiments, we will be using the OpenSPARC T2 processor which is an open source system-on-chip (SoC) consisting of several heterogeneous IPs. A block-level diagram of OpenSPARC processor is shown in Figure 3.1.

A message m is a 3-tuple $m = \langle o, d, \mathcal{S} \rangle$ where:

- o is the origin IP
- d is the destination IP
- $\mathcal{S} \subseteq S_o \cap S_d$, called the sets of signals

A message may consist of a control payload or a data payload along with error correcting code or commands to be executed.

3.2 Protocol Specification

To aid the specification of protocols, we define a family of protocols. A family of protocols may contain similar messages to perform a general task, but have a different initiator and target block. The example in Figure 2.1 may belong

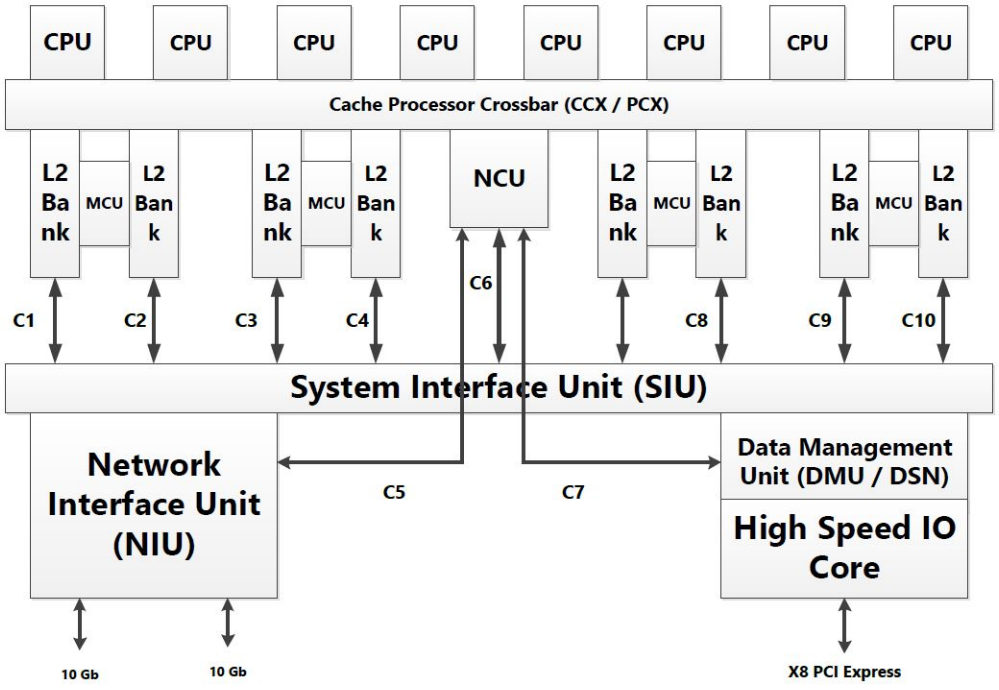

Figure 3.1: Block diagram of OpenSPARC T2 processor

to the protocol family that defines the power-on of all units. In this case, the initiator is the CPU and the target is the Radio. As another example in Figure 3.2 shows, an upstream memory write from the GFX block to the USB block may be contained within the family of protocols that defines upstream memory writes, but for this specific protocol, the GFX block is the initiator and the USB block is the target. Thinking of protocols in this manner makes them easier to specify and use as an input into the trace signal selection problem. Instead of trying to identify all possible protocols that may occur to complete a certain task, we can group them into families and specify initiators and targets.

The goal for our trace signal selection at this level of abstraction is to quickly identify and narrow down on the protocol inter-leaving. This would allow the debuggers to localize any observed bugs in the design to a small set of possibly buggy functional units. Once a bug can be localized to only a small subset of all the functional units in a design, more specific debug tools can be used to further root-cause the problem if needed. In some cases, the goal may be to identify the possible stimulus that can be used to recreate the bug in pre-silicon from our trace signal selection information. In either case, analyzing all the protocols to select our signals can help trace selection for the localization of specific bug types.

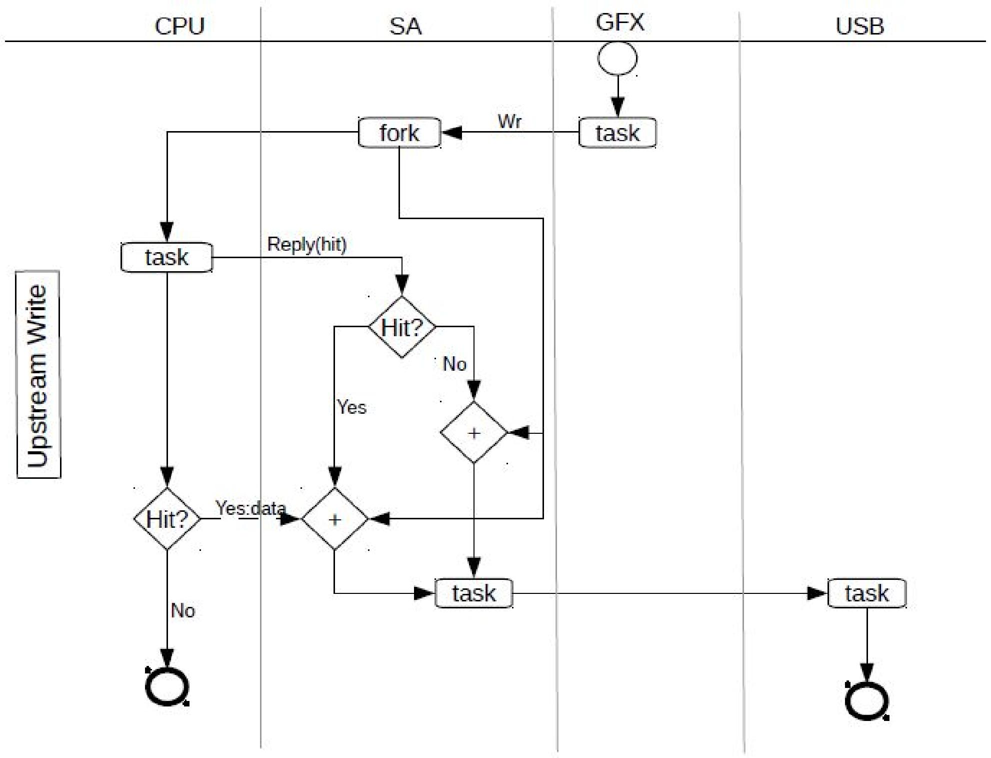

Figure 3.2: Protocol diagram for an upstream write. The symbols marked with + indicate synchronization points where all the incoming messages must have arrived before continuing.

Some common bug types from the system level that can be captured are halting bugs, data bugs, and control bugs. A halting bug is any bug that causes a protocol to fail to complete. A halting bug can be caused by a variety of internal issues from a functional block simply failing to send a message, to an incorrect field on a sent message, to setting an incorrect recipient on a message, which may cause the recipient to ignore the message and halt the protocol. A data bug passes along incorrect values in any field during the execution of a protocol. This can be caused by any incorrect handling of data at any point in the protocol. A control bug is when a functional block sends a message to an incorrect block which may cause unspecified final behavior. From the above definitions, one can observe that a single bug may belong to multiple types. For example, a bug that halts a protocol by sending a message to an incorrect block would classify as both a halting bug and control bug. In either case, if we attempt to capture traces that show either a control bug or halting bug, we will be able to localize this bug to some extent.

To select trace signals based on protocol specifications, we need some method to specify protocols in a format that can be analyzed. Protocols are often defined in diagrams such as those in Figure 2.1 and Figure 3.2.

These diagrams provide information on the flow of the protocol: the timing sequence, the information sent on each message, the sender and receiver of each message, and any possible conditional messages. Here the vertical columns are different functional units or IPs involved in the protocol. A circle represents the starting point of the protocol, whereas a bold circle represents the end of the protocol. Tasks performed by an IP as part of the protocol are shown within rectangles in the vertical column corresponding to the IP. Diagonal boxes are decision points. Diagonal boxes with + sign are synchronization points, where forward movement happens only after all incoming edges have been completed. Arrows within a vertical column show in what order the tasks are performed by an IP as the protocol moves forward. Arrows across vertical column represent messages travelling from source IP/block to destination IP/block.

A protocol $\mathcal{R}$ can be defined as a 4-tuple $\mathcal{R} = \langle \mathcal{V}, \mathcal{L}, \delta, v_0 \rangle$ where:

- $\mathcal{V} \neq \emptyset$ called the set of states. $\mathcal{V} = \mathcal{T} \cup \mathcal{D}$ where $\mathcal{T}$ = set of nodes representing a task in an IP, $\mathcal{D}$ = set of nodes representing pre-condition internal to an IP to initiate a messages.

- $\mathcal{L}$ is the set of all *observable messages* across all valid interfaces. $\mathcal{L}$ does not contain the internal events e of an IP.

- $\delta \subseteq \mathcal{S} \times (\mathcal{L} \cup e) \times \mathcal{S}$, where $(s, \alpha, s') \in \delta$ and is written as $s \overset{\alpha}{\to} s'$.

- $v_0 \in \mathcal{V}$ is the initial state of the protocol.

For a set of protocols $\{\mathcal{R}_k\}$ $k \geq 1$, an interleaving $\mathcal{G}$ of the protocols is defined as an ordered sequence of multiple instances of concurrently executing protocols.

For our running example we will consider the three protocols in Figure 3.3. These protocols are for a railroad crossing. On receipt of a signal indicating that a train is approaching, it closes the gates and only opens them after the train has sent a signal indicating that it crossed the road. For simplicity, it is assumed that all trains pass the relevant track section in the same direction from left to right. The states of the protocol for the *Train* have the following intuitive meaning: in state *far* the train is not close to the crossing, in state *near* it is approaching the crossing and has just sent a signal to notify this, and in state *in* it is at the crossing. The states of the *Gate* have the obvious interpretation. The state changes of the *Controller* stand for handshaking with the trains (via the actions *approach* and *exit*) and the *Gate* (via the actions *lower* and *raise* via which the *Controller* causes the gate to close or to open, respectively).

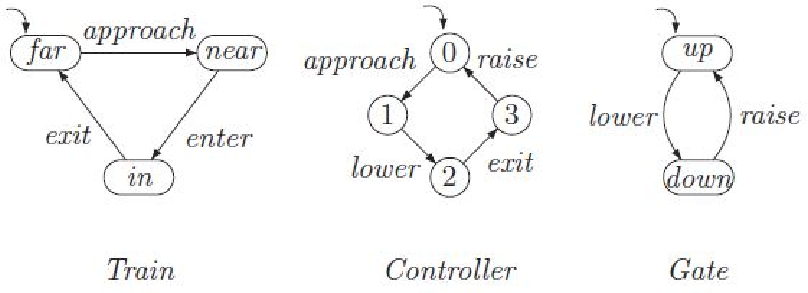

Figure 3.3: The components of the railroad crossing

3.3 Labelled Transition System

A labelled transition system (LTS) $\mathcal{T}$ can be defined as a 4-tuple $\mathcal{T} = \langle \mathcal{S}, \mathcal{I}, \mathcal{A}, \delta \rangle$ where:

- $\mathcal{S}$ is the set of states
- $\mathcal{I}$ is the set of initial states
- $\mathcal{A}$ is the set of actions
- δ is a total transition relation: $\delta \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$

We construct a LTS to represent the overall system with concurrent execution of protocols. The LTS captures all the different possible interleavings of the set of protocols. Concurrency is represented by interleaving, that is, the non-deterministic choice between activities of the simultaneously acting protocols. Consider the transition systems of two traffic lights for non-intersecting (i.e., parallel) roads. It is assumed that the traffic lights switch completely independent of each other. For example, the traffic lights may be controlled by pedestrians who would like to cross the road. Each traffic light is modeled as a simple transition system with two states, one state modeling a red light, the other one modeling a green light. The transition system of the parallel composition of both traffic lights is sketched at the bottom of Figure 3.4 where ||| denotes the interleaving operator. In principle, any form of interlocking of the actions of the two traffic lights is possible. For instance, in the initial state where both traffic lights are red, there is a non-deterministic choice between which of the lights turns green. Note that this non-determinism is descriptive, and does not model a scheduling problem between the traffic lights.

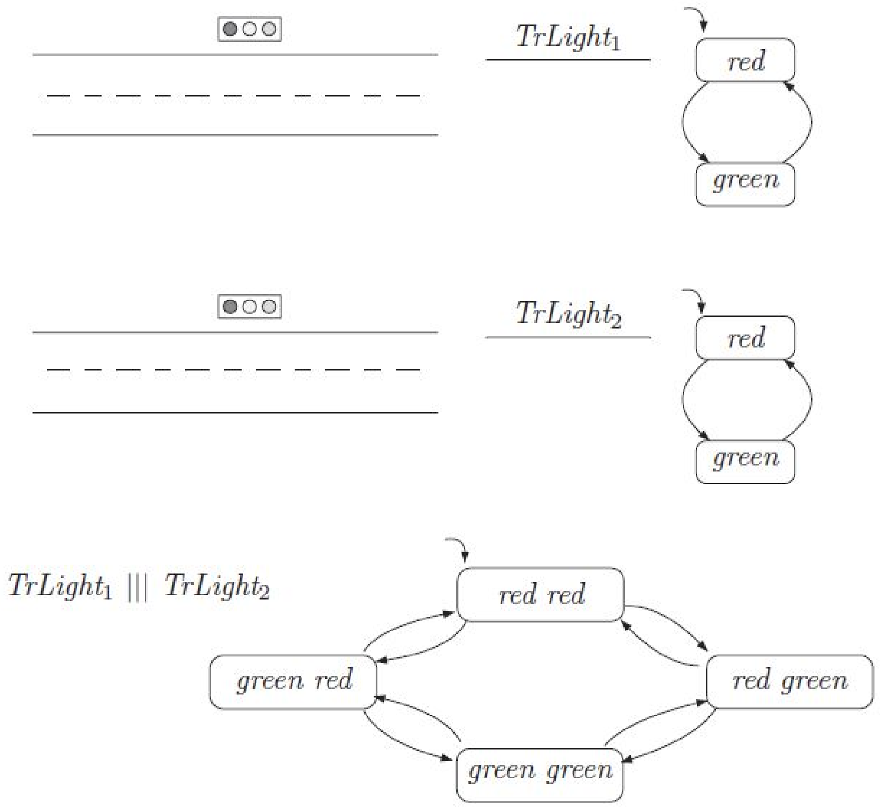

Figure 3.4: Example of interleaving operator for transition systems

We are now in a position to formally define the interleaving (denoted |||) of transition systems. The transition system $TS_1|||TS_2$ represents an interleaving resulting from the weaving (or merging) of the actions of the components as described by TS_1 and TS_2. The states of $TS_1|||TS_2$ are pairs $\langle s_1, s_2\rangle$ consisting of states s_i of the components TS_i. The outgoing transitions of the global state $\langle s_1, s_2\rangle$ consist of the outgoing transitions of s_1 together with those of s_2. Accordingly, whenever the composed system is in state $\langle s_1, s_2\rangle$, a non-deterministic choice is made between all outgoing transitions of local state s_1 and those of local state s_2.

Algorithm 1 *ConstructInterleavedLTS*

```
$T_1$ and $T_2$ be individual transition systems
$T = T_1 ||| T_2$
for each node $s_i$ in $T_1$ do
    for each node $s_j$ in $T_2$ do
        for each outgoing edge from $s_i$ to $s_k$ do
            draw corresponding outgoing edge from $\langle s_i, s_j\rangle$ to $\langle s_k, s_j\rangle$
        end for
        for each outgoing edge from $s_j$ to $s_k$ do
            draw corresponding outgoing edge from $\langle s_i, s_j\rangle$ to $\langle s_i, s_k\rangle$
        end for
    end for
end for
```

Algorithm 1 describes the procedure for construction of the interleaved LTS for two protocols represented by T_1 and T_2. For more than two protocols, the algorithm can be modified suitably. Interleaving of the first two systems can be constructed, which can then be interleaved with the third system and so on until all the individual systems have been inter-leaved. For our running example of railroad crossing, the individual protocols are shown in Figure 3.3. The interleaved LTS for the three protocols is shown in Figure 3.5.

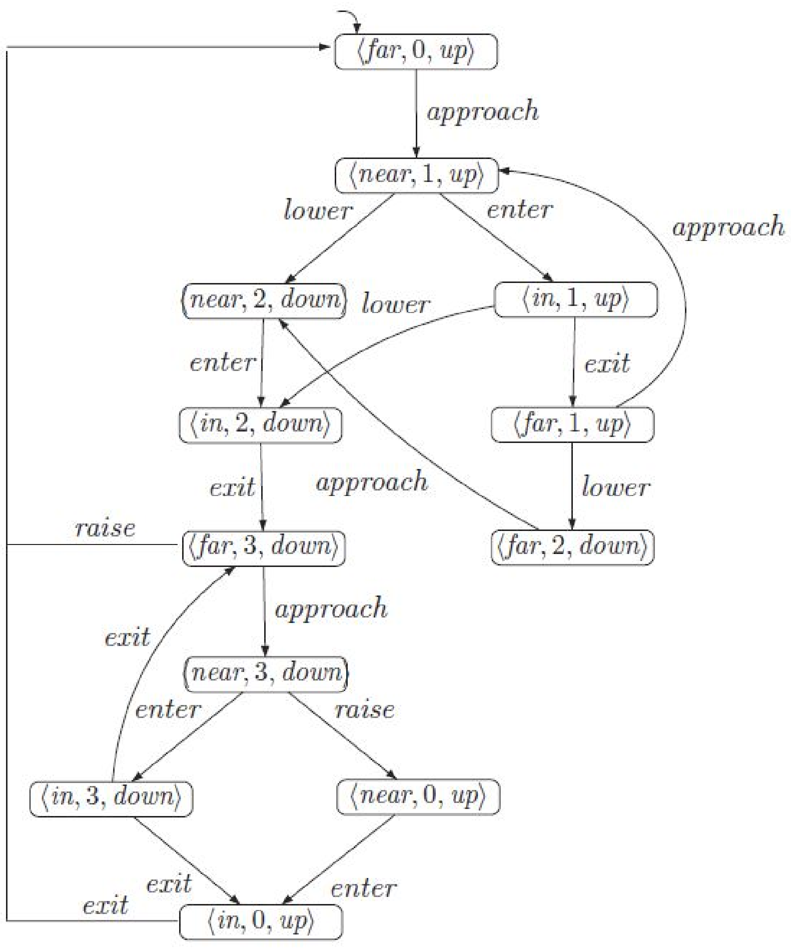

Figure 3.5: Interleaved LTS for railroad crossing

3.4 Problem Formulation

The LTS constructed represents the interleaving of individual protocols. If we have the capacity to trace all the messages then we can deterministically say what state of the LTS we are in, that is, identify all the individual protocol instances and their states. We can also tell the complete path taken in the LTS during the execution. However, due to limited trace buffer width constraint we can only select a few messages to trace. So we cannot identify the LTS state or protocol interleaving exactly but can try our best to identify it. We want to select messages which can help identify and narrow down on the protocol interleavings. We associate two random variables with the LTS structure representing the protocol interleaving.

Let X be a random variable representing different states in the LTS, i.e., it can take any value in the set $\mathcal{S}$ of the different states of the LTS. Let $\mathcal{A}$ be the set of all possible messages in the LTS. Let Y_i be a random variable representing a candidate set of messages. Out of all possible messages, we can only trace a few that satisfy the trace buffer width constraint, i.e., whose combined width is less than the trace buffer width. Several sets of messages may satisfy the trace buffer width constraint and are candidates for tracing. $Y_i,\ i \in \mathcal{I}$ are random variables representing all such candidates for tracing. The values taken by any such Y_i are a subset of $\mathcal{A}$. In our working example from Figure 3.5 we have:

$\mathcal{S} = \{(far, 0, up),\ (near, 1, up),\ (near, 2, down),\ (in, 1, up),\ (in, 2, down),$
$(far, 1, up),\ (far, 3, down),\ (far, 2, down),\ (near, 3, down),\ (in, 3, down),$
$(near, 0, up),\ (in, 0, up)\}.$

$\mathcal{A} = \{approach,\ lower,\ enter,\ exit,\ raise\}$

Suppose we cannot trace all five messages in $\mathcal{A}$ and the trace buffer width is 3, that is, we can only trace three messages. There are $\binom{5}{3} = 10$ possible candidates for tracing which satisfy the trace buffer width:

Y_1 takes values in the set $\mathcal{A}_1 = \{approach,\ lower,\ enter\}$
Y_2 takes values in the set $\mathcal{A}_2 = \{approach,\ lower,\ exit\}$
Y_3 takes values in the set $\mathcal{A}_3 = \{approach,\ lower,\ raise\}$
Y_4 takes values in the set $\mathcal{A}_4 = \{approach,\ enter,\ exit\}$
Y_5 takes values in the set $\mathcal{A}_5 = \{approach,\ enter,\ raise\}$
Y_6 takes values in the set $\mathcal{A}_6 = \{approach,\ exit,\ raise\}$
Y_7 takes values in the set $\mathcal{A}_7 = \{lower,\ enter,\ exit\}$
Y_8 takes values in the set $\mathcal{A}_8 = \{lower,\ enter,\ raise\}$
Y_9 takes values in the set $\mathcal{A}_9 = \{lower,\ exit,\ raise\}$
Y_{10} takes values in the set $\mathcal{A}_{10} = \{enter,\ exit,\ raise\}$

$Y_i,\ 1 \leq i \leq 10$ are the random variables representing various candidates for tracing. We want to select the best candidate for tracing which helps us to identify and narrow the protocol interleaving. Protocol interleaving is represented by various states of the LTS. So in other words we want to select a Y_i which helps identify X as best as possible. This notion is captured in mutual information which one random variable conveys about another. The best candidate for tracing is $Y_i,\ 1 \leq i \leq 10$, which maximizes $I(X;Y_i)$.

All values of X are equally probable since the LTS can be in any state. Therefore

$$p_X(x) = \frac{1}{total\ number\ of\ states\ in\ the\ LTS} \quad \forall x \in \mathcal{S}$$

For finding the marginal distribution of Y_i, we count the number of occurrences of each message in the set $\mathcal{A}$ over the entire LTS. Then the marginal distribution of Y_i is:

$$p_{Y_i}(y) = \frac{number\ of\ occurrences\ of\ y\ in\ the\ LTS}{\sum_{z \in \mathcal{A}_i} number\ of\ occurrences\ of\ z\ in\ the\ LTS}$$

For finding the joint probability, we use the conditional probability and the marginal distributions.

$$p(x,y) = p(x|y)p(y) = p(y|x)p(x) \tag{3.1}$$

$p(x)$ and $p(y)$ are known and $p(x|y)$ can be calculated from the fraction of times the LTS state x is reached after the message $Y_i = y$ has been observed, that is, has y as an incoming edge in the LTS. Or in other words, $p(x|y)$ is the fraction of times x is reached, from the total number of occurrences of the message y in the LTS.

$$p_{X|Y_i}(x|y) = \frac{number\ of\ occurrences\ of\ y\ in\ the\ LTS\ leading\ to\ state\ x}{total\ number\ of\ occurrences\ of\ y\ in\ the\ LTS}$$

We can now apply the formula to calculate $I(X;Y_i)$ and select the candidate Y_i that maximizes the mutual information over X. Calculation of $I(X;Y_1)$ for our example from Figure 3.5 is shown below:

$$p_X(x) = \frac{1}{12} \quad \forall x \in \mathcal{S}$$

$$p_{Y_1}(y) = \begin{cases} \frac{4}{11} & \text{if } y = \text{ approach} \\ \frac{3}{11} & \text{if } y = \text{ lower} \\ \frac{4}{11} & \text{if } y = \text{ enter} \end{cases}$$

$$p_{X|Y_1}(x|approach) = \begin{cases} \frac{2}{4} & \text{if } x = \text{ (near,1,up)} \\ \frac{1}{4} & \text{if } x = \text{ (near,2,down)} \\ \frac{1}{4} & \text{if } x = \text{ (near,3,down)} \end{cases}$$

$$p_{X|Y_1}(x|lower) = \begin{cases} \frac{1}{3} & \text{if } x = \text{ (near,2,down)} \\ \frac{1}{3} & \text{if } x = \text{ (in,2,down)} \\ \frac{1}{3} & \text{if } x = \text{ (far,2,down)} \end{cases}$$

$$p_{X|Y_1}(x|enter) = \begin{cases} \frac{1}{4} & \text{if } x = \text{ (in,1,up)} \\ \frac{1}{4} & \text{if } x = \text{ (in,2,down)} \\ \frac{1}{4} & \text{if } x = \text{ (in,3,down)} \\ \frac{1}{4} & \text{if } x = \text{ (in,0,up)} \end{cases}$$

$$p_{X,Y_1}(x, approach) = \begin{cases} \frac{2}{11} & \text{if } x = \text{ (near,1,up)} \\ \frac{1}{11} & \text{if } x = \text{ (near,2,down)} \\ \frac{1}{11} & \text{if } x = \text{ (near,3,down)} \end{cases}$$

$$p_{X,Y_1}(x, lower) = \begin{cases} \frac{1}{11} & \text{if } x = \text{ (near,2,down)} \\ \frac{1}{11} & \text{if } x = \text{ (in,2,down)} \\ \frac{1}{11} & \text{if } x = \text{ (far,2,down)} \end{cases}$$

$$p_{X,Y_1}(x, enter) = \begin{cases} \frac{1}{11} & \text{if } x = \text{ (in,1,up)} \\ \frac{1}{11} & \text{if } x = \text{ (in,2,down)} \\ \frac{1}{11} & \text{if } x = \text{ (in,3,down)} \\ \frac{1}{11} & \text{if } x = \text{ (in,0,up)} \end{cases}$$

$$\begin{aligned} I(X;Y_1) &= \sum_{x,y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)} \\ &= \frac{2}{11} \log \frac{\frac{2}{11}}{\frac{1}{12}\frac{4}{11}} + \frac{6}{11} \log \frac{\frac{1}{11}}{\frac{1}{12}\frac{4}{11}} + \frac{3}{11} \log \frac{\frac{1}{11}}{\frac{1}{12}\frac{3}{11}} \\ &= 0.3765 \end{aligned}$$

Similarly we have:

$I(X;Y_2) = 0.5683$
$I(X;Y_3) = 0.7308$
$I(X;Y_4) = 0.5324$
$I(X;Y_5) = 0.5975$
$I(X;Y_6) = 0.5972$
$I(X;Y_7) = 0.5181$
$I(X;Y_8) = 0.5856$
$I(X;Y_9) = 0.5865$
$I(X;Y_{10}) = 0.5425$

Hence the set represented by Y_3 should be selected because $I(X;Y_i)$ is maximum for $i = 3$.

3.5 Cross Validation

In order to cross-validate our method, we need to keep in mind the fact that we can trace only a few messages out of the total. In other words, due to the messages that are not traced there are some unobserved actions. During an execution we do not know how many times one of the un-traced messages occurred before we observe a traced message. Since we cannot deterministically say how the system is progressing or how the execution moves forward in the LTS, we can only try to fill in the gaps in the sequence of observed messages and try to predict the path taken during execution. We can associate a probability with which a path might have been taken in the LTS during execution. We have to try and infer from the sequence of traced messages observed during an execution.

In our case, we have some messages that are being traced while others are not. During an execution, we will be able to observe the traced messages as the execution run moves forward. However we do not know which of the untraced messages might have happened at different stages of the execution. These are the unobserved messages during an execution run. Therefore an execution run corresponds to a sequence of messages, some of which are observed (recorded on trace buffer) and some unobserved (not traced). Due to the unobserved messages, we cannot deterministically say what path is taken in the LTS during an execution, what state of the LTS the system is presently in and hence what interleaving of protocols is present. We can draw conclusions from the observed messages, make predictions on the unobserved messages and try to assign a probability to what path might have been taken in the LTS.

We apply the depth-first search (DFS) algorithm on the LTS representing the interleaving of protocols to get all the paths from the start node to the end node. We extract the message sequence for each such path. From these message sequences we delete the untraced messages so that they now represent the message sequence which would be recorded on the trace buffer if the actual execution is along the corresponding path in the LTS. So now we have pairs of LTS paths and the corresponding message sequence (without un-traced messages).

During a post-silicon execution of the chip the traced messages are recorded on the trace buffer. This sequence of messages from the trace buffer is matched against the message sequences extracted from the LTS. If there is a unique match then we know what path in the LTS was followed during the execution. This information is passed on to the debuggers who replay this in a pre-silicon environment. Replaying all the protocols in a pre-silicon environment is not possible due to slow speed. However, once we are able to identify the path or state from the LTS, the debugger can replay this smaller scenario in simulation and look at it in more detail to debug since simulation has full observability of signals.

If there are multiple matches, then we cannot be certain which path was followed in the LTS. As such, all the matching paths are possible candidates for further investigation by the debugger, but even this information is very valuable to the debugger as we have given him several possible paths, one of which was followed during the actual execution and resulted in the bug. This set is much smaller than the complete massive design.

CHAPTER 4

IMPLEMENTATION AND RESULTS

4.1 Implementation

Our experiments are performed on the OpenSPARC T2 SoC from Oracle. OpenSPARC T2 contains eight SPARC physical processor cores. The eight physical cores are connected through a crossbar to an on-chip unified 4 Mbyte, 16-way associative L2 cache. The L2 cache connects to four on-chip DRAM controllers, which directly interface to a pair of fully buffered DIMM (FBD) channels. Each SPARC physical core is supported by system-on-chip hardware components.

OpenSPARC T2 has four memory control units (MCUs), one for each memory branch with a pair of L2 banks interacting with exactly one dynamic random-access memory (DRAM) branch. The data management unit (DMU) manages transaction layer packet (TLP) to/from the PCI-express unit (PEU) and maintains the ordering from the PEU and then to the system interface unit (SIU). The network interface unit (NIU) connects a pair of on-chip 10 Gb/s Ethernet MACs to the rest of the system. The SIU connects the NIU, DMU and L2 cache. SIU is the L2 cache access point for the network and PCI-express subsystems. The non-cacheable unit (NCU) performs an address decode on I/O-addressable transactions and directs them to the appropriate block (for example, NIU, DMU, clock control unit). A block level diagram of OpenSPARC processor is shown in Figure 3.1.

The complete specification and design of the OpenSPARC T2 SoC is available from Oracle [78]. The specification document gives details on each of the individual IPs. It gives information on how two IPs communicate with each other for different purposes and what interface they use. It describes the message structure, signals involved in the message and its timing diagram. We use all this information to construct complete protocols for different tasks.

The protocols are shown in Appendix A.

Various regression suites are provided along with the OpenSPARC design by Oracle. Regression tests are used to test the functionality of different parts of the design. These regression suites stimulate different blocks and IPs in the SoC, thus instantiating different protocols. We use monitors written in Verilog to keep track of different interface signals between IPs. These monitors record various signals that make up messages between IPs along with their time stamp. Or in other words they record the sequence of messages during a simulation. This mimics an actual recording from post-silicon execution on a trace buffer and is used for cross-validation.

We have implemented our LTS construction and method selection methodology in Python. The user or debugger provides the specifications of different protocols, the bit width of individual messages and the trace buffer width budget. The tool reads the protocol specifications provided by the user and parses them. The tool then performs the complete analysis, processing through the protocols iteratively. It merges them iteratively to construct the LTS (Algorithm 1) and then performs the analysis to select the trace signals as described in Sections 3.3 and 3.4 respectively. It gives the set of messages for tracing which satisfy the trace buffer width.

4.2 Experiments

We perform experiments on different groups of protocols. Individual protocols from OpenSPARC are shown in Appendix A. We group together protocols which have similar IPs and perform similar tasks. We then apply our LTS construction and message selection algorithm to select the messages which should be traced. We then apply the cross-validation method. We do not have access to an actual OpenSPARC chip with post-silicon debug features. So instead we simulate the SoC using regression suites which stimulate different IPs and hence instantiate various protocols. These simulations are extremely slow as compared to an actual chip execution. This is the reason we make groups of protocols for our analysis, as a simulation which sensitizes all the IPs and instantiates all protocols is not possible. The monitors which we have written record the traced messages in a sequence. We then match it against the message sequences extracted from the LTS. These results for different groups of protocols are presented next.

4.3 Results for Message Selection

4.3.1 PIO Read, PIO Write and Mondo Interrupt

PIO Write has 3 nodes, PIO Read has 7 nodes and Mondo Interrupt has 8 nodes. There are 168 nodes in the constructed LTS. The set of all the messages is given below.

$\mathcal{A} = \{pioreq,\ piowcrd,\ dmusiidata,\ reqtot,\ grant,\ siincu,\ monacknack\}$

We are using shorthand for messages. A more detailed description can be found in Appendix A. The bit widths of the messages are listed below in the same order.

$\mathcal{A}\_width = \{pioreq:\ 65\,bits,\ piowcrd:\ 6\,bits,\ dmusiidata:\ 130\,bits,\ reqtot:\ 1\,bit,\ grant:\ 1\,bit,\ siincu:\ 32\,bits,\ monacknack:\ 8\,bits\}$

Next we present the analysis for message selection for different sizes of the trace buffer.

Trace Buffer Width: 32
Candidate message sets are listed below.

Y_1 takes values in the set $\mathcal{A}_1 = \{reqtot,\ piowcrd,\ monacknack\}$
Y_2 takes values in the set $\mathcal{A}_2 = \{reqtot,\ piowcrd,\ grant\}$
Y_3 takes values in the set $\mathcal{A}_3 = \{reqtot,\ monacknack,\ grant\}$
Y_4 takes values in the set $\mathcal{A}_4 = \{piowcrd,\ monacknack,\ grant\}$
Y_5 takes values in the set $\mathcal{A}_5 = \{reqtot,\ piowcrd,\ monacknack,\ grant\}$

Mutual information for candidate message sets are listed below.

$I(X;Y_1) = 1.3821$
$I(X;Y_2) = 1.2903$

$I(X;Y_3) = 1.3364$
$I(X;Y_4) = 1.3821$
$I(X;Y_5) = 1.3896$

Figure 4.1 shows the mutual information for different message sets. $\mathcal{A}_5 = \{reqtot,\ piowcrd,\ monacknack,\ grant\}$ is selected.

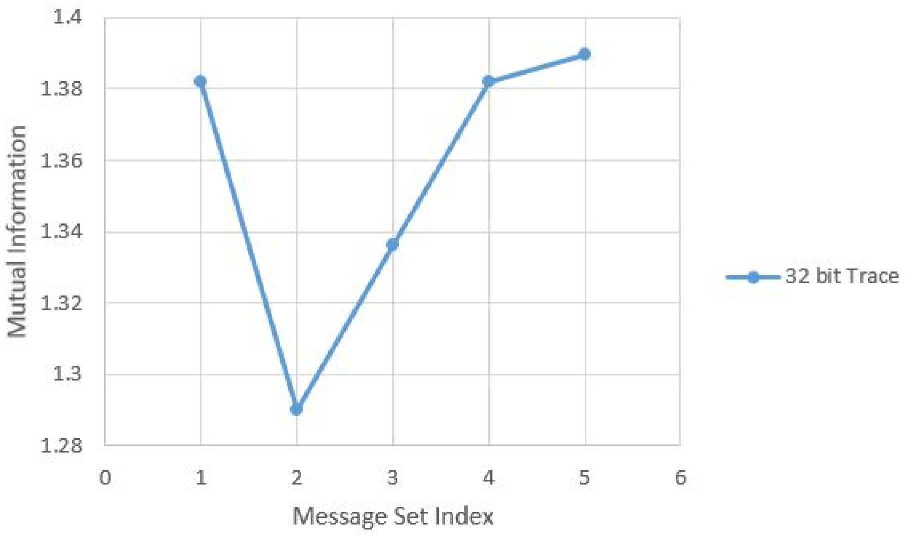

Figure 4.1: Mutual information for different message sets

Trace Buffer Width: 64

There are a large number of candidate message sets, some of which are listed below.

Y_1 takes values in the set $\mathcal{A}_1 = \{grant,\ piowcrd,\ monacknack\}$
Y_2 takes values in the set $\mathcal{A}_2 = \{siincu,\ reqtot,\ piowcrd,\ monacknack\}$
Y_3 takes values in the set $\mathcal{A}_3 = \{siincu,\ reqtot,\ piowcrd,\ grant\}$
Y_4 takes values in the set $\mathcal{A}_4 = \{siincu,\ reqtot,\ monacknack,\ grant\}$
Y_5 takes values in the set $\mathcal{A}_5 = \{siincu,\ piowcrd,\ monacknack,\ grant\}$
Y_6 takes values in the set $\mathcal{A}_6 = \{reqtot,\ piowcrd,\ monacknack,\ grant\}$
Y_7 takes values in the set $\mathcal{A}_7 = \{siincu,\ reqtot,\ piowcrd,\ monacknack,\ grant\}$

Mutual information for candidate message sets are listed below.

$I(X;Y_1) = 1.3821$

$I(X;Y_2) = 1.2692$
$I(X;Y_3) = 1.2187$
$I(X;Y_4) = 1.3589$
$I(X;Y_5) = 1.2692$
$I(X;Y_6) = 1.3896$
$I(X;Y_7) = 1.3963$

Figure 4.2 shows the mutual information for different message sets. $\mathcal{A}_7 = \{siincu,\ reqtot,\ piowcrd,\ monacknack,\ grant\}$ is selected.

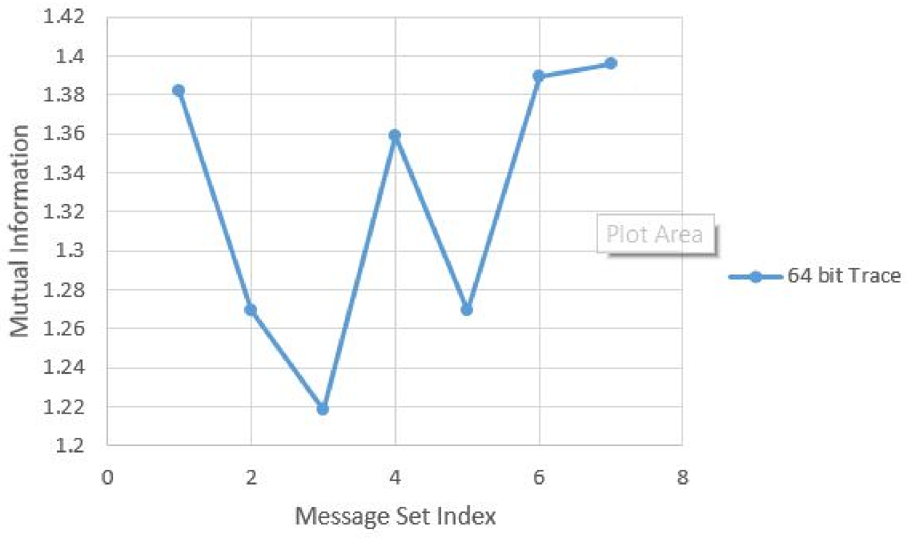

Figure 4.2: Mutual information for different message sets

Trace Buffer Width: 128

Some of the candidate message sets are listed below.

Y_1 takes values in the set $\mathcal{A}_1 = \{pioreq,\ siincu,\ reqtot,\ grant,\ piowcrd,\ monacknack\}$
Y_2 takes values in the set $\mathcal{A}_2 = \{pioreq,\ siincu,\ reqtot,\ piowcrd,\ monacknack\}$
Y_3 takes values in the set $\mathcal{A}_3 = \{pioreq,\ siincu,\ reqtot,\ piowcrd,\ grant\}$
Y_4 takes values in the set $\mathcal{A}_4 = \{pioreq,\ siincu,\ reqtot,\ monacknack,\ grant\}$
Y_5 takes values in the set $\mathcal{A}_5 = \{pioreq,\ siincu,\ piowcrd,\ monacknack,\ grant\}$
Y_6 takes values in the set $\mathcal{A}_6 = \{pioreq,\ reqtot,\ piowcrd,\ monacknack,\ grant\}$
Y_7 takes values in the set $\mathcal{A}_7 = \{siincu,\ reqtot,\ piowcrd,\ monacknack,\ grant\}$

Mutual information for candidate message sets are listed below.

$I(X; Y_1) = 1.4103$
$I(X; Y_2) = 1.1531$
$I(X; Y_3) = 1.1261$
$I(X; Y_4) = 1.2100$
$I(X; Y_5) = 1.1531$
$I(X; Y_6) = 1.2247$
$I(X; Y_7) = 1.3963$

Figure 4.3 shows the mutual information for different message sets. $\mathcal{A}_1 = \{pioreq,\ siincu,\ reqtot,\ grant,\ piowcrd,\ monacknack\}$ is selected.

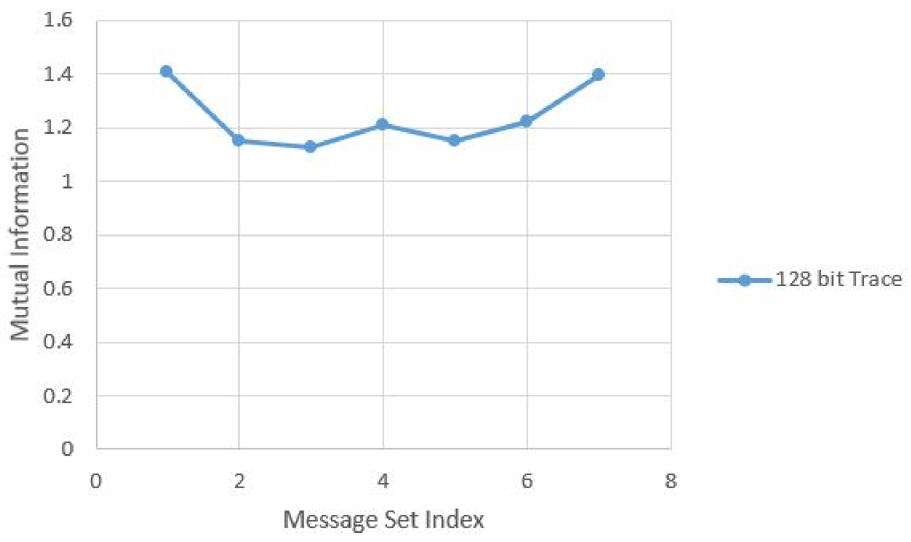

Figure 4.3: Mutual information for different message sets

4.3.2 PIO Read, PIO Write and Network Data Packet Processing

PIO Write has 3 nodes, PIO Read has 7 nodes and Network Data Packet Processing has 7 nodes. There are 147 nodes in the constructed LTS. The set of all the messages is given below.

$\mathcal{A} = \{pioreq,\ piowcrd,\ dmusiidata,\ reqtot,\ grant,\ siincu,\ niusiidata\}$

We are using shorthand for messages. A more detailed description can be found in Appendix A. The bit widths of the messages are listed below in the same order.

$\mathcal{A}\_width = \{pioreq: \ 65\, bits,\ piowcrd: \ 6\, bits,\ dmusiidata: \ 130\, bits,\ reqtot: \ 1\ bit,\ grant: \ 1\ bit,\ siincu: \ 32\ bits,\ niusiidata: \ 130\ bits\}$

Next we present the analysis for message selection for different sizes of the trace buffer.

Trace Buffer Width: 32

Candidate message sets are listed below.

Y_1 takes values in the set $\mathcal{A}_1 = \{reqtot,\ piowcrd\}$
Y_2 takes values in the set $\mathcal{A}_2 = \{reqtot,\ grant\}$
Y_3 takes values in the set $\mathcal{A}_3 = \{piowcrd,\ grant\}$
Y_4 takes values in the set $\mathcal{A}_4 = \{reqtot,\ piowcrd,\ grant\}$

Mutual information for candidate message sets are listed below.

$I(X;Y_1) = 1.2154$
$I(X;Y_2) = 1.2517$
$I(X;Y_3) = 1.2154$
$I(X;Y_4) = 1.2585$

Figure 4.4 shows the mutual information for different message sets. $\mathcal{A}_4 = \{reqtot,\ piowcrd,\ grant\}$ is selected.

Trace Buffer Width: 64

Some of the candidate message sets are listed below.

Y_1 takes values in the set $\mathcal{A}_1 = \{siincu,\ reqtot,\ piowcrd\}$
Y_2 takes values in the set $\mathcal{A}_2 = \{siincu,\ reqtot,\ grant\}$
Y_3 takes values in the set $\mathcal{A}_3 = \{reqtot,\ piowcrd,\ grant\}$
Y_4 takes values in the set $\mathcal{A}_4 = \{siincu,\ piowcrd,\ grant\}$

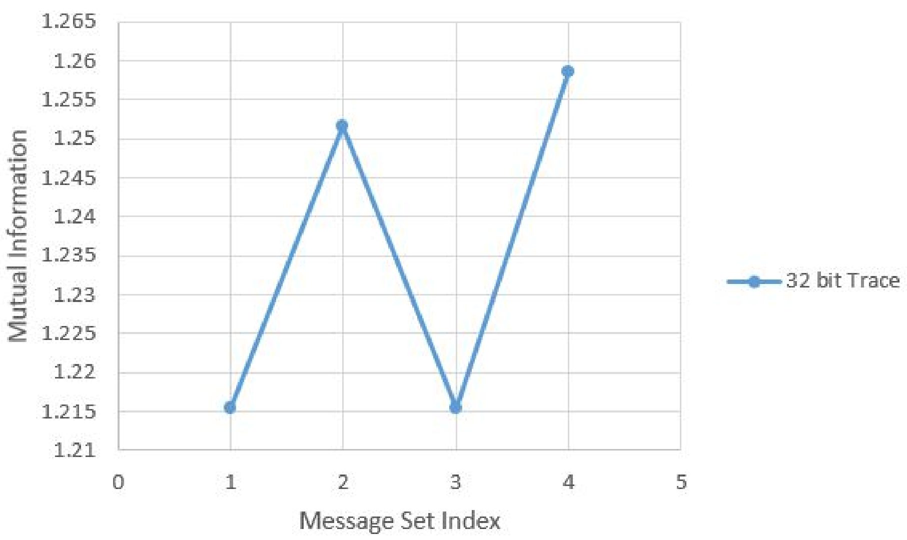

Figure 4.4: Mutual information for different message sets

Y_5 takes values in the set $\mathcal{A}_5 = \{reqtot,\ siincu,\ piowcrd,\ grant\}$

Mutual information for candidate message sets are listed below.

$I(X;Y_1) = 1.1188$
$I(X;Y_2) = 1.1921$
$I(X;Y_3) = 1.2585$
$I(X;Y_4) = 1.1188$
$I(X;Y_5) = 1.2687$

Figure 4.5 shows the mutual information for different message sets. $\mathcal{A}_5 = \{reqtot,\ siincu,\ piowcrd,\ grant\}$ is selected.

Trace Buffer Width: 128
Some of the candidate message sets are listed below.

Y_1 takes values in the set $\mathcal{A}_1 = \{pioreq,\ siincu,\ reqtot,\ piowcrd\}$
Y_2 takes values in the set $\mathcal{A}_2 = \{pioreq,\ siincu,\ reqtot,\ grant\}$
Y_3 takes values in the set $\mathcal{A}_3 = \{pioreq,\ reqtot,\ piowcrd,\ grant\}$
Y_4 takes values in the set $\mathcal{A}_4 = \{pioreq,\ siincu,\ piowcrd,\ grant\}$
Y_5 takes values in the set $\mathcal{A}_5 = \{reqtot,\ siincu,\ piowcrd,\ grant\}$
Y_6 takes values in the set $\mathcal{A}_6 = \{pioreq,\ reqtot,\ siincu,\ piowcrd,\ grant\}$

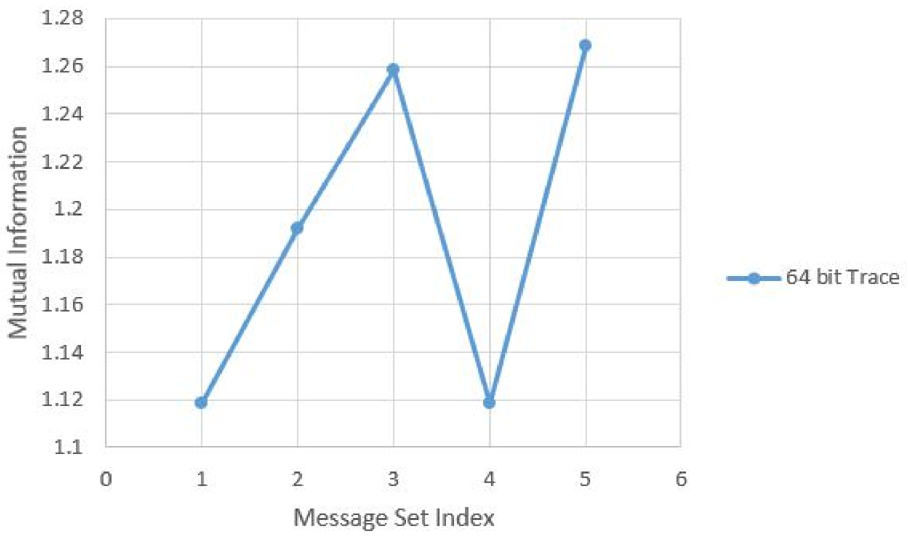

Figure 4.5: Mutual information for different message sets

Mutual information for candidate message sets are listed below.

$I(X;Y_1) = 1.0443$
$I(X;Y_2) = 1.0916$
$I(X;Y_3) = 1.1281$
$I(X;Y_4) = 1.0443$
$I(X;Y_5) = 1.2687$
$I(X;Y_6) = 1.2929$

Figure 4.6 shows the mutual information for different message sets. $\mathcal{A}_6 = \{pioreq,\ reqtot,\ siincu,\ piowcrd,\ grant\}$ is selected.

4.3.3 PIO Read, PIO Write, Mondo Interrupt and Network Data Packet Processing

PIO Write has 3 nodes, PIO Read has 7 nodes, Mondo Interrupt has 8 nodes and Network Data Packet Processing has 7 nodes. There are 1176 nodes in the constructed LTS. The set of all the messages is given below.

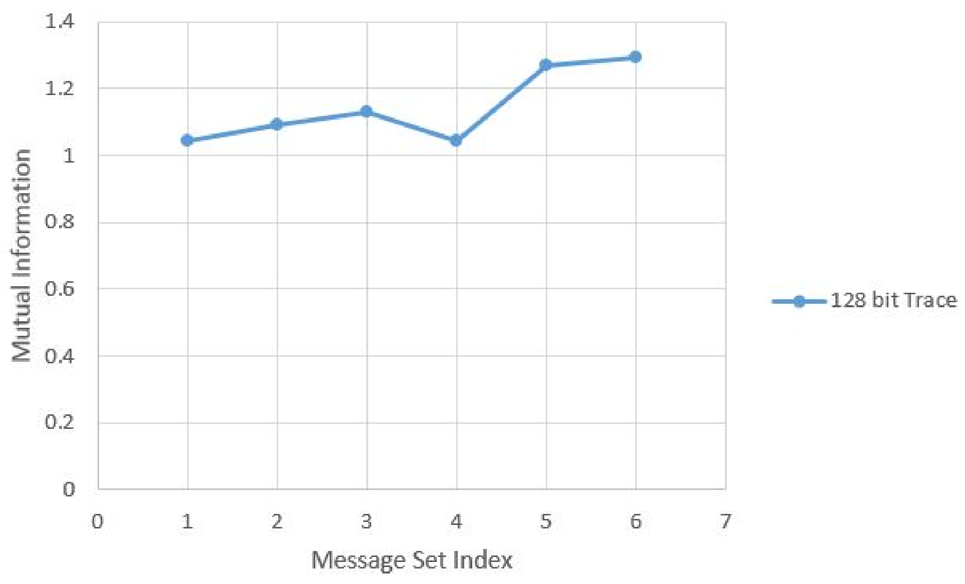

Figure 4.6: Mutual information for different message sets

$\mathcal{A} = \{pioreq,\ piowcrd,\ dmusiidata,\ reqtot,\ grant,\ siincu,\ monacknack,\ niusiidata\}$

We are using shorthand for messages. A more detailed description can be found in Appendix A. The bit widths of the messages are listed below in the same order.

$\mathcal{A}\_width = \{pioreq:\ 65\,bits,\ piowcrd:\ 6\,bits,\ dmusiidata:\ 130\,bits,\ reqtot:\ 1\ bit,\ grant:\ 1\ bit,\ siincu:\ 32\ bits,\ monacknack:\ 8\ bits,\ niusiidata:\ 130\ bits\}$

Next we present the analysis for message selection for different sizes of the trace buffer.

Trace Buffer Width: 32
Candidate message sets are listed below.

Y_1 takes values in the set $\mathcal{A}_1 = \{reqtot,\ grant,\ monacknack\}$
Y_2 takes values in the set $\mathcal{A}_2 = \{reqtot,\ piowcrd,\ grant\}$
Y_3 takes values in the set $\mathcal{A}_3 = \{reqtot,\ monacknack,\ piowcrd\}$
Y_4 takes values in the set $\mathcal{A}_4 = \{piowcrd,\ monacknack,\ grant\}$

Y_5 takes values in the set $\mathcal{A}_5 = \{reqtot,\ piowcrd,\ monacknack,\ grant\}$

Mutual information for candidate message sets are listed below.

$I(X;Y_1) = 1.1067$
$I(X;Y_2) = 1.0810$
$I(X;Y_3) = 1.1280$
$I(X;Y_4) = 1.1280$
$I(X;Y_5) = 1.1785$

Figure 4.7 shows the mutual information for different message sets. $\mathcal{A}_5 = \{reqtot,\ piowcrd,\ monacknack,\ grant\}$ is selected.

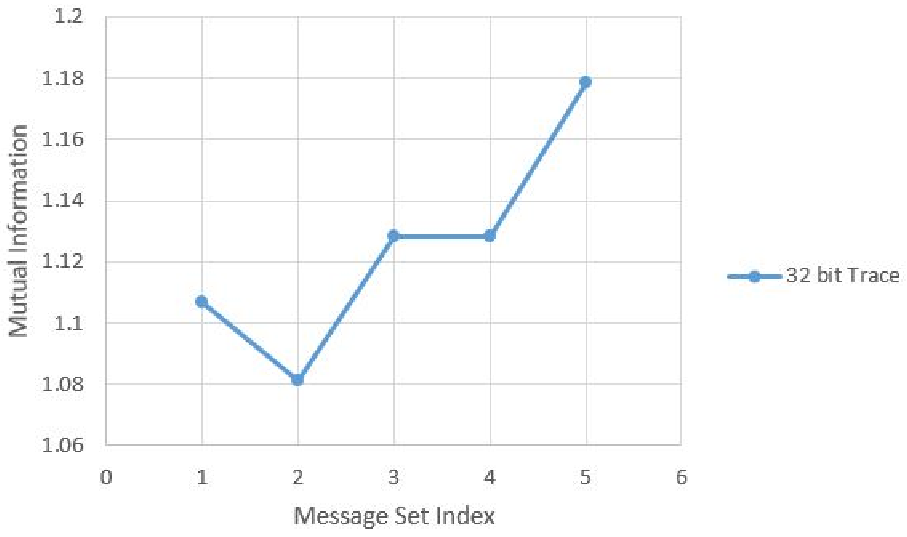

Figure 4.7: Mutual information for different message sets

Trace Buffer Width: 64

There are a large number of candidate message sets, some of which are listed below.

Y_1 takes values in the set $\mathcal{A}_1 = \{reqtot,\ grant,\ siincu,\ monacknack\}$
Y_2 takes values in the set $\mathcal{A}_2 = \{grant,\ reqtot,\ piowcrd,\ monacknack\}$
Y_3 takes values in the set $\mathcal{A}_3 = \{siincu,\ reqtot,\ piowcrd,\ grant\}$
Y_4 takes values in the set $\mathcal{A}_4 = \{siincu,\ reqtot,\ monacknack,\ piowcrd\}$
Y_5 takes values in the set $\mathcal{A}_5 = \{siincu,\ piowcrd,\ monacknack,\ grant\}$

Y_6 takes values in the set $\mathcal{A}_6 = \{siincu,\ reqtot,\ piowcrd,\ monacknack,\ grant\}$

Mutual information for candidate message sets are listed below.

$I(X;Y_1) = 0.9855$
$I(X;Y_2) = 1.1785$
$I(X;Y_3) = 0.9315$
$I(X;Y_4) = 0.9864$
$I(X;Y_5) = 0.9864$
$I(X;Y_6) = 1.2047$

Figure 4.8 shows the mutual information for different message sets. $\mathcal{A}_6 = \{siincu,\ reqtot,\ piowcrd,\ monacknack,\ grant\}$ is selected.

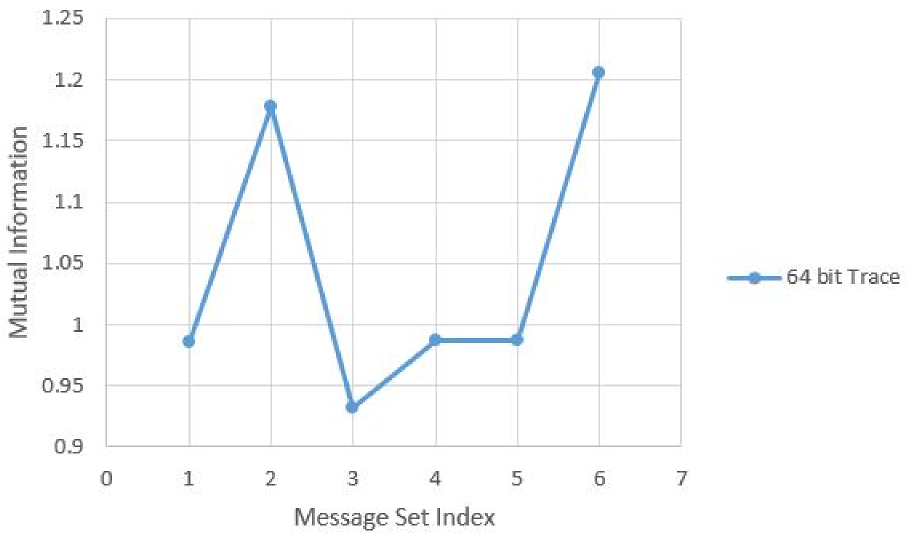

Figure 4.8: Mutual information for different message sets

Trace Buffer Width: 128

Some of the candidate message sets are listed below.

Y_1 takes values in the set $\mathcal{A}_1 = \{pioreq,\ siincu,\ reqtot,\ grant,\ piowcrd,\ monacknack\}$
Y_2 takes values in the set $\mathcal{A}_2 = \{pioreq,\ siincu,\ reqtot,\ grant,\ monacknack\}$
Y_3 takes values in the set $\mathcal{A}_3 = \{pioreq,\ monacknack,\ reqtot,\ piowcrd,\ grant\}$
Y_4 takes values in the set $\mathcal{A}_4 = \{pioreq,\ siincu,\ reqtot,\ piowcrd,\ grant\}$
Y_5 takes values in the set $\mathcal{A}_5 = \{pioreq,\ siincu,\ piowcrd,\ monacknack,\ reqtot\}$

Y_6 takes values in the set $\mathcal{A}_6 = \{pioreq,\ siincu,\ piowcrd,\ monacknack,\ grant\}$
Y_7 takes values in the set $\mathcal{A}_7 = \{siincu,\ reqtot,\ piowcrd,\ monacknack,\ grant\}$

Mutual information for candidate message sets are listed below.

$I(X;Y_1) = 1.2804$
$I(X;Y_2) = 0.9617$
$I(X;Y_3) = 1.0977$
$I(X;Y_4) = 0.9210$
$I(X;Y_5) = 0.9615$
$I(X;Y_6) = 0.9615$
$I(X;Y_7) = 1.2047$

Figure 4.9 shows the mutual information for different message sets. $\mathcal{A}_1 = \{pioreq,\ siincu,\ reqtot,\ grant,\ piowcrd,\ monacknack\}$ is selected.

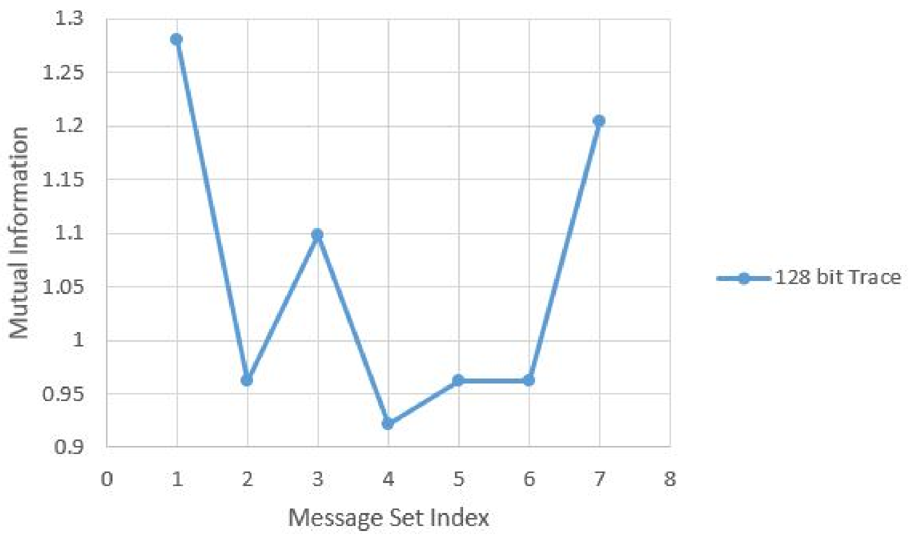

Figure 4.9: Mutual information for different message sets

Trace Buffer Width: 256

Some of the candidate message sets are listed below.

Y_1 takes values in the set $\mathcal{A}_1 = \{pioreq,\ siincu,\ reqtot,\ grant,\ piowcrd,\ monacknack,\ niusiidata\}$
Y_2 takes values in the set $\mathcal{A}_2 = \{pioreq,\ siincu,\ reqtot,\ grant,\ piowcrd,\ monacknack,\ dmusiidata\}$

Y_3 takes values in the set $\mathcal{A}_3 = \{piowcrd,\ siincu,\ reqtot,\ grant,\ monacknack,\ niusiidata\}$
Y_4 takes values in the set $\mathcal{A}_4 = \{piowcrd,\ monacknack,\ reqtot,\ siincu,\ grant,\ dmusiidata\}$
Y_5 takes values in the set $\mathcal{A}_5 = \{pioreq,\ siincu,\ monacknack,\ piowcrd,\ grant,\ niusiidata\}$
Y_6 takes values in the set $\mathcal{A}_6 = \{pioreq,\ siincu,\ piowcrd,\ monacknack,\ grant,\ dmusiidata\}$
Y_7 takes values in the set $\mathcal{A}_7 = \{pioreq,\ siincu,\ piowcrd,\ monacknack,\ reqtot,\ niusiidata\}$
Y_8 takes values in the set $\mathcal{A}_8 = \{siincu,\ reqtot,\ piowcrd,\ pioreq,\ grant,\ niusiidata\}$
Y_9 takes values in the set $\mathcal{A}_9 = \{monacknack,\ reqtot,\ piowcrd,\ pioreq,\ grant,\ niusiidata\}$

Mutual information for candidate message sets are listed below.

$I(X;Y_1) = 1.4495$
$I(X;Y_2) = 1.4090$
$I(X;Y_3) = 1.3363$
$I(X;Y_4) = 0.9267$
$I(X;Y_5) = 0.9975$
$I(X;Y_6) = 0.9296$
$I(X;Y_7) = 0.9975$
$I(X;Y_8) = 0.9575$
$I(X;Y_9) = 1.1194$

Figure 4.10 shows the mutual information for different message sets. $\mathcal{A}_1 = \{pioreq,\ siincu,\ reqtot,\ grant,\ piowcrd,\ monacknack,\ niusiidata\}$ is selected.

4.3.4 DMA Read Request and CPU-Main Memory Load

DMA Read Request has 11 nodes and CPU-Main Memory Load has 9 nodes. There are 99 nodes in the constructed LTS. The set of all the messages is

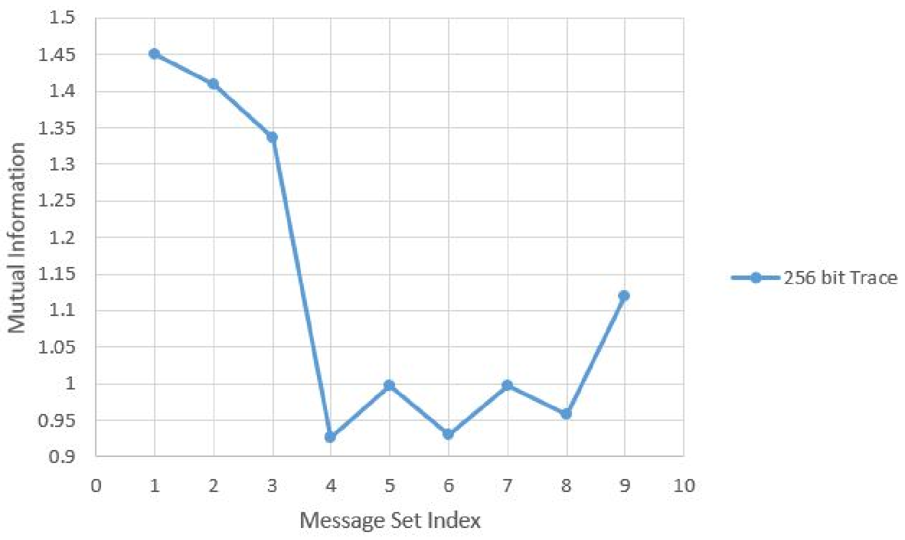

Figure 4.10: Mutual information for different message sets

given below.

$\mathcal{A} = \{dmusiidata,\ siil2t,\ l2tmcu,\ mcul2tack,\ mcul2tdata,\ l2bsio,\ siodmu,\ pcxl2t,\ pcxl2tdata,\ l2tcpxreq,\ l2tcpxdata,\ cpxl2t\}$

We are using shorthand for messages. A more detailed description can be found in Appendix A. The bit widths of the messages are listed below in the same order.

$\mathcal{A}\_width = \{dmusiidata:\ 130\,bits,\ siil2t:\ 32\,bits,\ l2tmcu:\ 41\,bits,\ mcul2tack:\ 1\ bit,\ mcul2tdata:\ 134\ bits,\ l2bsio:\ 34\ bits,\ siodmu:\ 130\ bits,\ pcxl2t:\ 2\,bits,\ pcxl2tdata:\ 130\,bits,\ l2tcpxreq:\ 8\,bits,\ l2tcpxdata:\ 146\,bits,\ cpxl2t:\ 8\ bits\}$

Next we present the analysis for message selection for different sizes of the trace buffer.

Trace Buffer Width: 32
Candidate message sets are listed below.

Y_1 takes values in the set $\mathcal{A}_1 = \{mcul2tack,\ l2tcpxreq,\ pcxl2t\}$
Y_2 takes values in the set $\mathcal{A}_2 = \{mcul2tack,\ l2tcpxreq,\ cpxl2t\}$

Y_3 takes values in the set $\mathcal{A}_3 = \{mcul2tack,\ pcxl2t,\ cpxl2t\}$
Y_4 takes values in the set $\mathcal{A}_4 = \{l2tcpxreq,\ pcxl2t,\ cpxl2t\}$
Y_5 takes values in the set $\mathcal{A}_5 = \{mcul2tack,\ l2tcpxreq,\ pcxl2t,\ cpxl2t\}$

Mutual information for candidate message sets are listed below.

$I(X;Y_1) = 1.9455$
$I(X;Y_2) = 1.9455$
$I(X;Y_3) = 1.9455$
$I(X;Y_4) = 1.8972$
$I(X;Y_5) = 1.9977$

Figure 4.11 shows the mutual information for different message sets. $\mathcal{A}_5 = \{mcul2tack,\ l2tcpxreq,\ pcxl2t,\ cpxl2t\}$ is selected.

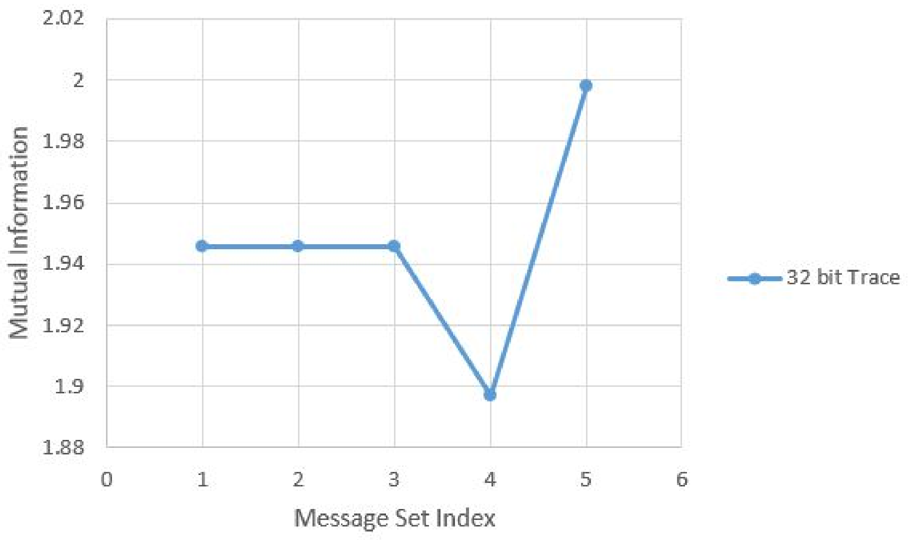

Figure 4.11: Mutual information for different message sets

Trace Buffer Width: 64
Some of the candidate message sets are listed below.

Y_1 takes values in the set $\mathcal{A}_1 = \{l2tcpxreq,\ pcxl2t,\ siil2t,\ cpxl2t\}$
Y_2 takes values in the set $\mathcal{A}_2 = \{l2tcpxreq,\ pcxl2t,\ l2bsio,\ cpxl2t\}$
Y_3 takes values in the set $\mathcal{A}_3 = \{l2tcpxreq,\ pcxl2t,\ cpxl2t,\ l2tmcu\}$

Y_4 takes values in the set $\mathcal{A}_4 = \{mcul2tack,\ l2tcpxreq,\ pcxl2t,\ siil2t,\ cpxl2t\}$
Y_5 takes values in the set $\mathcal{A}_5 = \{mcul2tack,\ l2tcpxreq,\ pcxl2t,\ l2bsio,\ cpxl2t\}$
Y_6 takes values in the set $\mathcal{A}_6 = \{mcul2tack,\ l2tcpxreq,\ pcxl2t,\ cpxl2t,\ l2tmcu\}$

Mutual information for candidate message sets are listed below.

$I(X;Y_1) = 1.9402$
$I(X;Y_2) = 1.9234$
$I(X;Y_3) = 1.9977$
$I(X;Y_4) = 2.0558$
$I(X;Y_5) = 1.9234$
$I(X;Y_6) = 1.9076$

Figure 4.12 shows the mutual information for different message sets. $\mathcal{A}_4 = \{mcul2tack,\ l2tcpxreq,\ pcxl2t,\ siil2t,\ cpxl2t\}$ is selected.

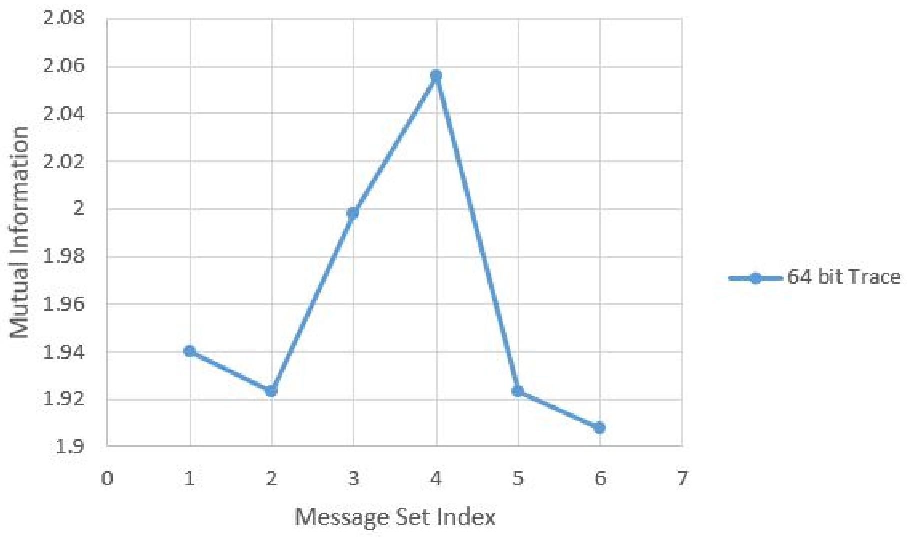

Figure 4.12: Mutual information for different message sets

Trace Buffer Width: 128

Some of the candidate message sets are listed below.

Y_1 takes values in the set $\mathcal{A}_1 = \{mcul2tack,\ l2tcpxreq,\ pcxl2t,\ siil2t,\ l2bsio,\ cpxl2t\}$

Y_2 takes values in the set $\mathcal{A}_2 = \{mcul2tack,\ l2tcpxreq,\ pcxl2t,\ siil2t,\ l2bsio,\ l2tmcu\}$
Y_3 takes values in the set $\mathcal{A}_3 = \{mcul2tack,\ l2tcpxreq,\ pcxl2t,\ siil2t,\ cpxl2t,\ l2tmcu\}$
Y_4 takes values in the set $\mathcal{A}_4 = \{mcul2tack,\ l2tcpxreq,\ pcxl2t,\ l2bsio,\ cpxl2t,\ l2tmcu\}$
Y_5 takes values in the set $\mathcal{A}_5 = \{mcul2tack,\ l2tcpxreq,\ siil2t,\ l2bsio,\ cpxl2t,\ l2tmcu\}$
Y_6 takes values in the set $\mathcal{A}_6 = \{mcul2tack,\ pcxl2t,\ siil2t,\ l2bsio,\ cpxl2t,\ l2tmcu\}$
Y_7 takes values in the set $\mathcal{A}_7 = \{l2tcpxreq,\ pcxl2t,\ siil2t,\ l2bsio,\ cpxl2t,\ l2tmcu\}$
Y_8 takes values in the set $\mathcal{A}_8 = \{mcul2tack,\ l2tcpxreq,\ pcxl2t,\ siil2t,\ l2bsio,\ cpxl2t,\ l2tmcu\}$

Mutual information for candidate message sets are listed below.

$I(X;Y_1) = 1.9768$
$I(X;Y_2) = 1.8803$
$I(X;Y_3) = 1.9614$
$I(X;Y_4) = 1.8674$
$I(X;Y_5) = 1.8803$
$I(X;Y_6) = 1.8803$
$I(X;Y_7) = 1.9768$
$I(X;Y_8) = 1.9952$

Figure 4.13 shows the mutual information for different message sets. $\mathcal{A}_8 = \{mcul2tack,\ l2tcpxreq,\ pcxl2t,\ siil2t,\ l2bsio,\ cpxl2t,\ l2tmcu\}$ is selected.

4.4 Cross-validation of Selected Message Set

In this section, we describe the cross-validation flow for one set of protocols, namely Mondo Interrupt, CPU-Main Memory Load, and NCU-XBAR Upstream & Downstream Protocols. For this purpose we perform simulation of

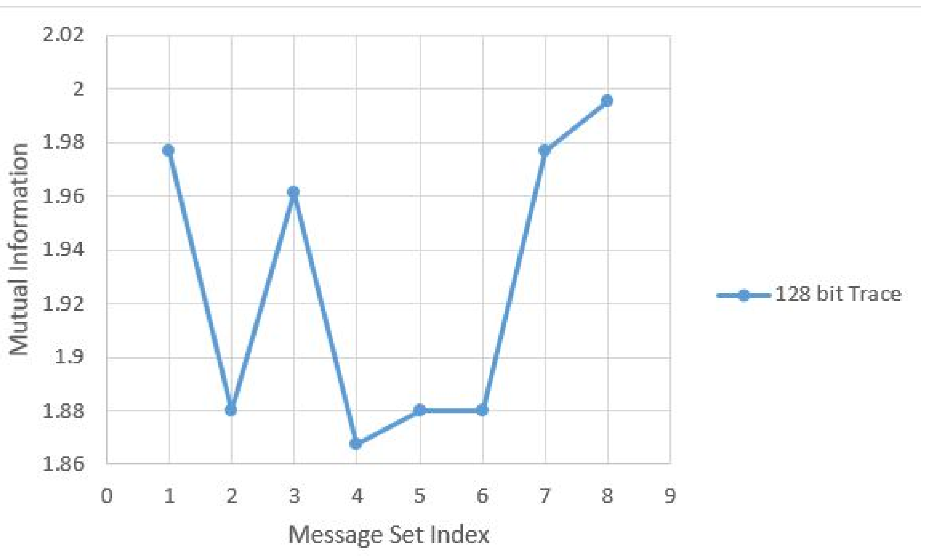

Figure 4.13: Mutual information for different message sets

the OpenSPARC design using regression suites which trigger these protocols. We use monitors to record messages during the simulation. We then follow the steps described in Section 3.5 and use the message sequence recorded on the trace buffer to come up with possible paths in the LTS which could have been followed during the execution. We check if the actual simulation path is in the set of reconstructed execution paths or not.

The list of all possible messages and their frequency of occurrence is shown below. They are arranged in decreasing order of frequency of occurrence. The bit width of each message is shown in brackets.

$\mathcal{A} = \{dmusiidata(130\, bits) : 108,\ siincu(32\, bits) : 108,\ cpxncugnt(8\, bits) : 72,\ ncucpxdata(146\, bits) : 72,\ ncucpxreq(8\, bits) : 72,\ pcxncudata(130\, bits) : 72,\ pcxncurdy(1\, bit) : 72,\ grant(1\, bit) : 54,\ reqtot(1\, bit) : 54,\ monacknack(8\ bits) : 54,\ l2tcpxdata(146\ bits) : 48,\ cpxl2t(8\ bits) : 48,\ mcul2tdata(134\, bits) : 48,\ mcul2tack(1\, bit) : 48,\ pcxl2tdata(130\, bits) : 48,\ l2tmcu(41\, bits) : 48,\ pcxl2t(2\, bits) : 48,\ l2tcpxreq(8\, bits) : 48\}$

The complete sequence of messages recorded during a simulation is given below. The corresponding message shorthands are shown in angular brackets.

PCX initiating data transfer to NCU $\langle pcxncurdy \rangle$
PCX Sending data to NCU = 21000ff8800641020000000000000040 $\langle pcxncudata \rangle$
NCU sending Request to CPX for CPU = 01 $\langle ncucpxreq \rangle$
NCU transferring data = 28200040000ff000000000000000000000040 to CPX for CPU = 1 $\langle ncucpxdata \rangle$
CPX indicating packet reached at CPU = 01 $\langle cpxncugnt \rangle$
L2T7 receiving a Request from PCX $\langle pcxl2t \rangle$
DATA Cycle started at L2T7 from PCX $\langle pcxl2tdata \rangle$
L2T7 Read Request to MCU3 $\langle l2tmcu \rangle$
MCU3 to L2T7 Read Request Acknowledgement $\langle mcul2tack \rangle$
MCU3 to L2T7 Read Data $\langle mcul2tdata \rangle$
L2T7 sending request to CPX $\langle l2tcpxreq \rangle$
Data Cycle started from L2T7 to CPX $\langle l2tcpxdata \rangle$
ACKNOWLEDGE from CPX Detected $\langle cpxl2t \rangle$
Mondo Interrupt Request Sent to SIU Ordered Queue $\langle dmusiidata \rangle$
DMU to SIU Mondo Interrupt Request / PIO Read Data Return Payload Cycle $\langle dmusiidata \rangle$
SIU Signalling a Transfer to NCU $\langle reqtot \rangle$
NCU Granting SIU for Transfer $\langle grant \rangle$
NCU Granting SIU for Transfer $\langle grant \rangle$
SIU to NCU Header and Payload Cycle $\langle siincu \rangle$
NCU sending Mondo Packet acknowledge (ack) to DMU for Mondo ID = 00 $\langle monacknack \rangle$

Figure 4.14 shows a section of the LTS and the progress of simulation along a path. The path taken during the simulation is marked in red. The messages which are traced and recorded on the trace buffer during the simulation are marked in blue. $(mon\_1, cpu\_1, xbar\_1)$ is the start state and $(mon\_8, cpu\_9, xbar\_6)$ is the end state. Multiple transitions may be possible from a state. The transition taken during the simulation and the next state reached is shown in red. The message is shown on the corresponding transition. If it is a traced message and is recorded on the trace buffer during simulation, it is shown in blue. A dotted transition means that a few states and transitions have not been shown due to limited space.

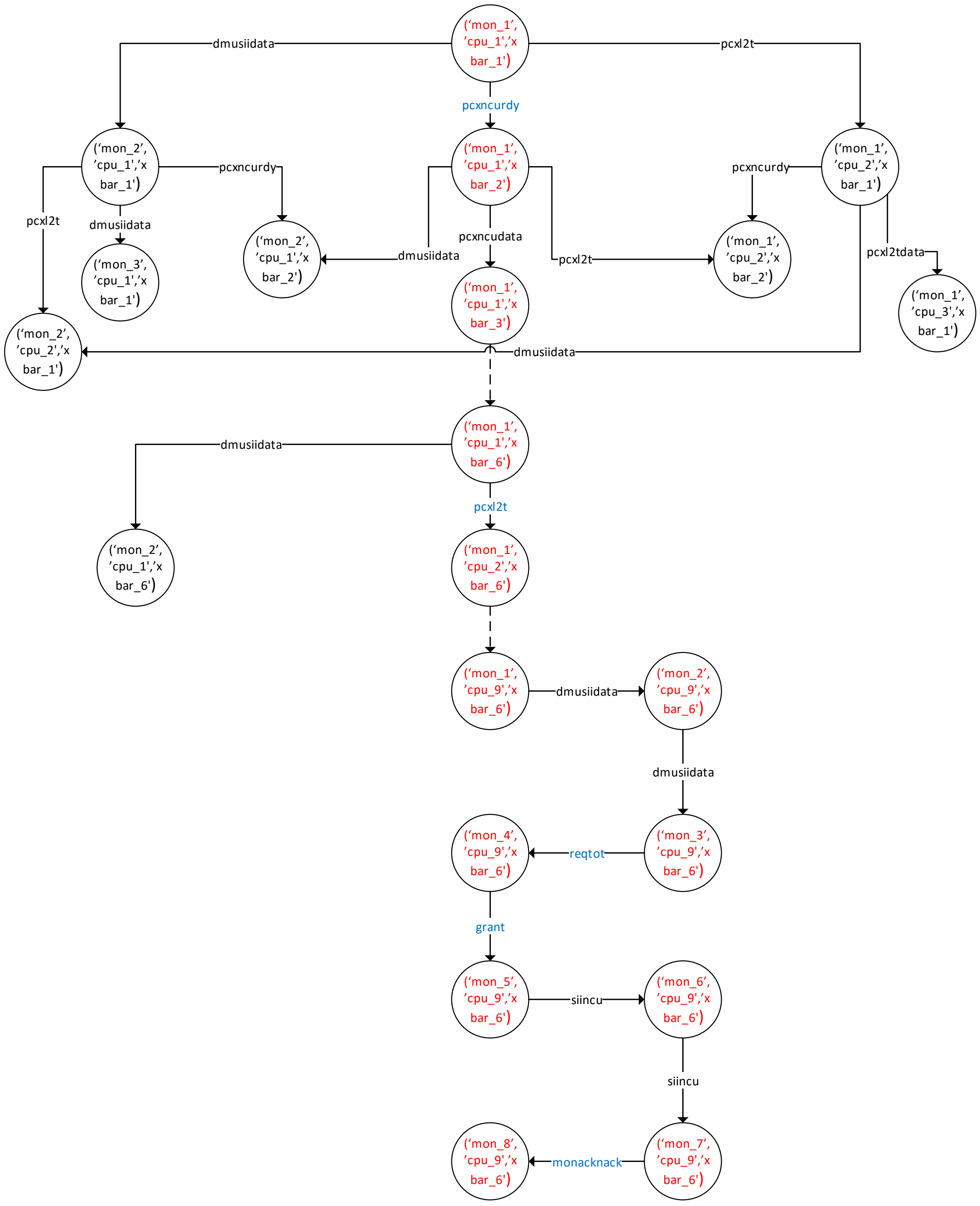
('mon_1', 'cpu_1','x bar_1')
dmusiidata
pcxl2t
pcxncurdy
('mon_2', 'cpu_1','x bar_1')
('mon_1', 'cpu_1','x bar_2')
('mon_1', 'cpu_2','x bar_1')
pcxl2t
dmusiidata
pcxncurdy
('mon_3', 'cpu_1','x bar_1')
('mon_2', 'cpu_1','x bar_2')
dmusiidata
pcxncudata
pcxl2t
('mon_1', 'cpu_2','x bar_2')
pcxl2tdata
('mon_1', 'cpu_3','x bar_1')
('mon_1', 'cpu_1','x bar_3')
('mon_2', 'cpu_2','x bar_1')
dmusiidata
('mon_1', 'cpu_1','x bar_6')
dmusiidata
('mon_2', 'cpu_1','x bar_6')
pcxl2t
('mon_1', 'cpu_2','x bar_6')
('mon_1', 'cpu_9','x bar_6')
dmusiidata
('mon_2', 'cpu_9','x bar_6')
dmusiidata
('mon_4', 'cpu_9','x bar_6')
reqtot
('mon_3', 'cpu_9','x bar_6')
grant
('mon_5', 'cpu_9','x bar_6')
siincu
('mon_6', 'cpu_9','x bar_6')
siincu
('mon_8', 'cpu_9','x bar_6')
monacknack
('mon_7', 'cpu_9','x bar_6')

Figure 4.14: Progress of simulation along a LTS path

4.4.1 32 bit Trace Buffer

Message set selected by our method for tracing: {*cpxl2t*, *grant*, *pcxncurdy*, *mcul2tack*, *monacknack*, *pcxl2t*, *reqtot*, *l2tcpxreq*}

Sequence of messages recorded on trace buffer during simulation: {*pcxncurdy*, *pcxl2t*, *mcul2tack*, *l2tcpxreq*, *cpxl2t*, *reqtot*, *grant*, *monacknack*}

Out of 10,000 different paths extracted from the LTS in Figure 4.14, 81 paths had matching message sequences and could have been followed during execution. The actual simulation path was also among the 81 matching LTS paths.

4.4.2 64 bit Trace Buffer

Message set selected by our method for tracing: {*cpxncugnt*, *cpxl2t*, *grant*, *pcxncurdy*, *mcul2tack*, *ncucpxreq*, *monacknack*, *pcxl2t*, *reqtot*, *l2tcpxreq*}

Sequence of messages recorded on trace buffer during simulation: {*pcxncurdy*, *ncucpxreq*, *cpxncugnt*, *pcxl2t*, *mcul2tack*, *l2tcpxreq*, *cpxl2t*, *reqtot*, *grant*, *monacknack*}

Out of 10,000 different paths extracted from the LTS in Figure 4.14, 45 paths had matching message sequences and could have been followed during execution. The actual simulation path was also among the 45 matching LTS paths.

4.4.3 128 bit Trace Buffer

Message set selected by our method for tracing: {*cpxncugnt*, *cpxl2t*, *siincu*, *grant*, *pcxncurdy*, *mcul2tack*, *ncucpxreq*, *monacknack*, *l2tmcu*, *pcxl2t*, *reqtot*, *l2tcpxreq*}

Sequence of messages recorded on trace buffer during simulation: {*pcxncurdy*, *ncucpxreq*, *cpxncugnt*, *pcxl2t*, *l2tmcu*, *mcul2tack*, *l2tcpxreq*, *cpxl2t*, *reqtot*, *grant*, *siincu*, *siincu*, *monacknack*}

Out of 10,000 different paths extracted from the LTS in Figure 4.14, 45 paths had matching message sequences and could have been followed during execution. The actual simulation path was also among the 45 LTS paths.

4.4.4 Evaluation of Different Message Sets

In this sub-section we evaluate different message sets which can be traced. Our method selects the message set with highest mutual information for tracing over message sets with lower mutual information. To evaluate different message sets, we follow the procedure described in Section 3.5 for each of them. We consider each set for tracing and calculate the number of LTS paths which could have been followed during execution. It is expected that a message set with higher mutual information should help us more in identifying the possible LTS path than a message set with lower mutual information. That is, a message set with higher mutual information should give fewer LTS paths which could have been followed during execution.

On the X-axis we plot mutual information for different message sets. The number of possible LTS paths for each message set is plotted on the Y-axis. Figures 4.15, 4.16 and 4.17 show the graphs for 32 bit trace buffer, 64 bit trace buffer and 128 bit trace buffer respectively. It can be seen in the figures that as the mutual information of a message set increases, the number of possible LTS paths for it decreases. This implies that a message set with higher mutual information gives a more definitive selection of set of paths which could have been followed during execution.

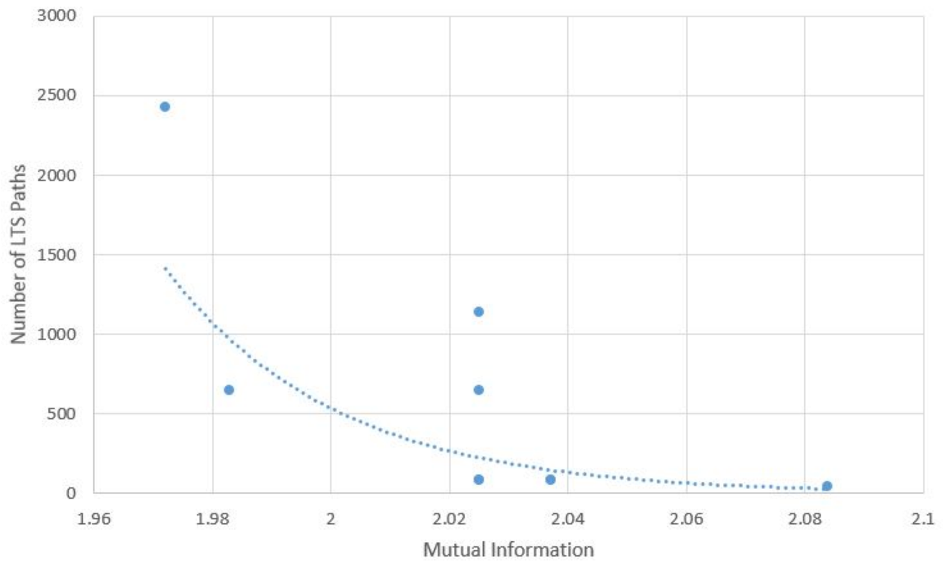

Figure 4.15: Evaluation of different message sets for 32 bit trace buffer

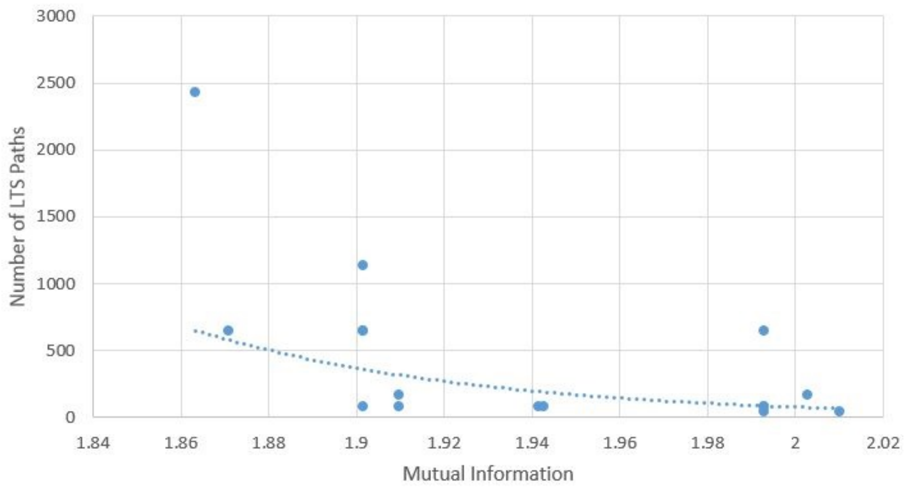

Figure 4.16: Evaluation of different message sets for 64 bit trace buffer

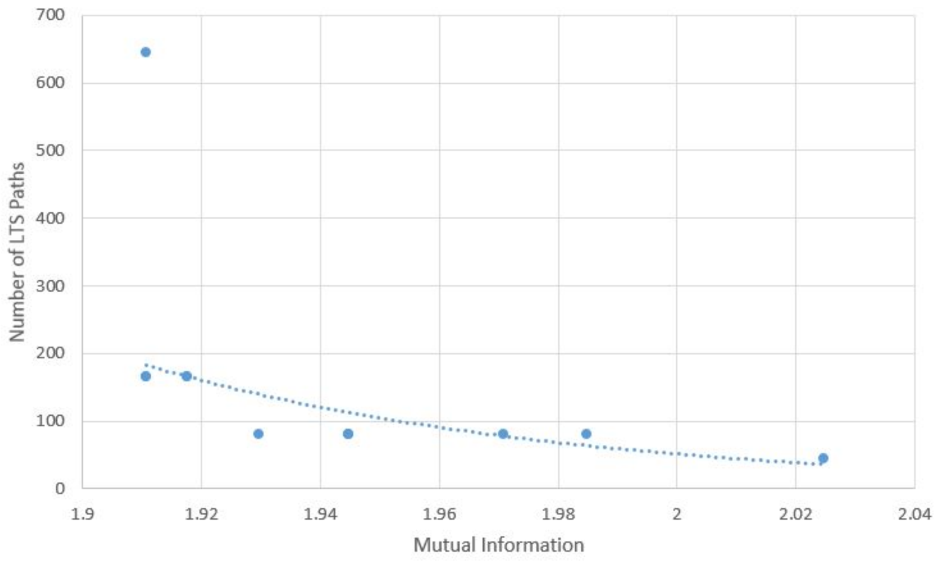

Figure 4.17: Evaluation of different message sets for 128 bit trace buffer

CHAPTER 5

CONCLUSION

5.1 Conclusion

The results of the experiments above demonstrate that our selection method does the following:

1. Provides an automated selection of trace signals that capture system-level behavior
2. Provides a trace signal selection method which can be used early in the design process unlike the gate-level methods which work on synthesized netlists
3. Provides the engineer specifying the protocols some degree of guidance in the final solution
4. Is effective at localizing bugs using a realistic trace buffer width relative to the overall design size

We have performed experiments on a real world complex SoC, namely OpenSPARC. We have used the available specifications to construct the protocols. The experiments have been performed on the groups of these protocols. These results have been cross-validated by comparing against full chip simulation of the SoC, obtained from Oracle.

5.2 Future Work

As part of further work we can try scaling the solution. At present the LTS constructed from individual protocols can blow up very quickly. We could come up with an algorithm to construct and analyze the LTS on the fly.

Future work could include a better cross-validation method as compared to running simulations of the design. Simulations are very slow and hence limit the scope of possible chip behaviors that can be analyzed. We could try implementing the design on FPGA and use signal tap hardware to trace the signals. We could also try a more formal approach to proving the method using theorems and proofs.

We could test the method on multiple designs to further substantiate the method. At present OpenSPARC is the only available open-source SoC with documentation and specifications to construct the protocols and do the analysis. Lack of open-source SoC is a big constraint. We do not really know what kind of hardware specifications, IPs, NoC architecture and protocols exist for real world SoCs such as those in our smart phones. Using multiple designs can help improve our method.

APPENDIX A

PROTOCOLS

Shown in this appendix are the protocols used in our experiments.

A.1 Mondo Interrupt

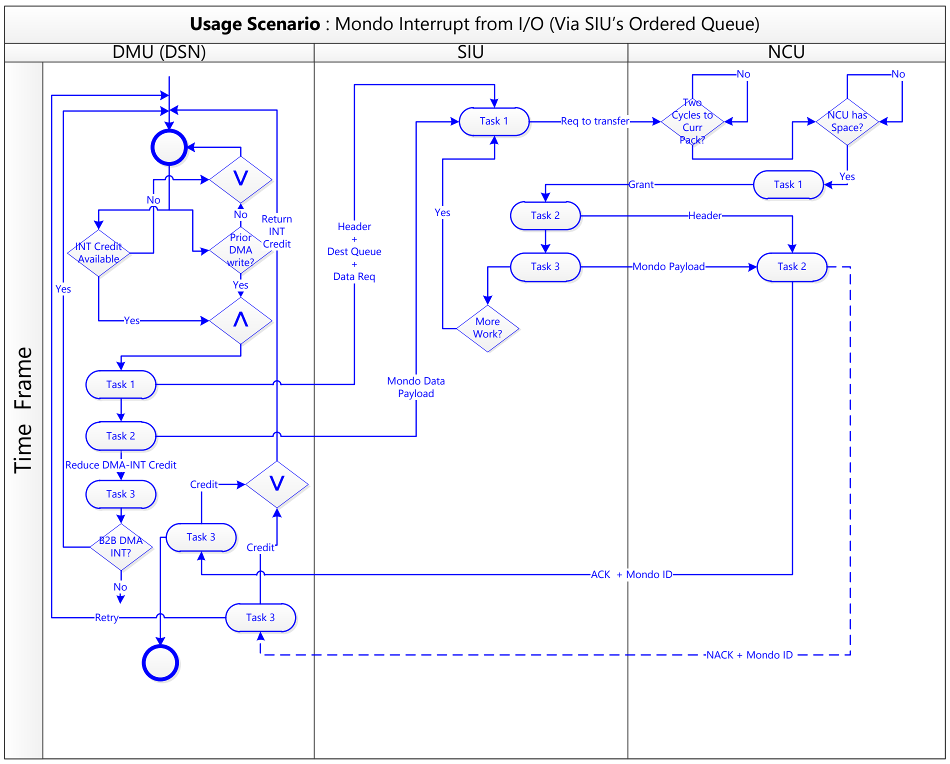

Figure A.1: Mondo Interrupt

A.2 PIO Read

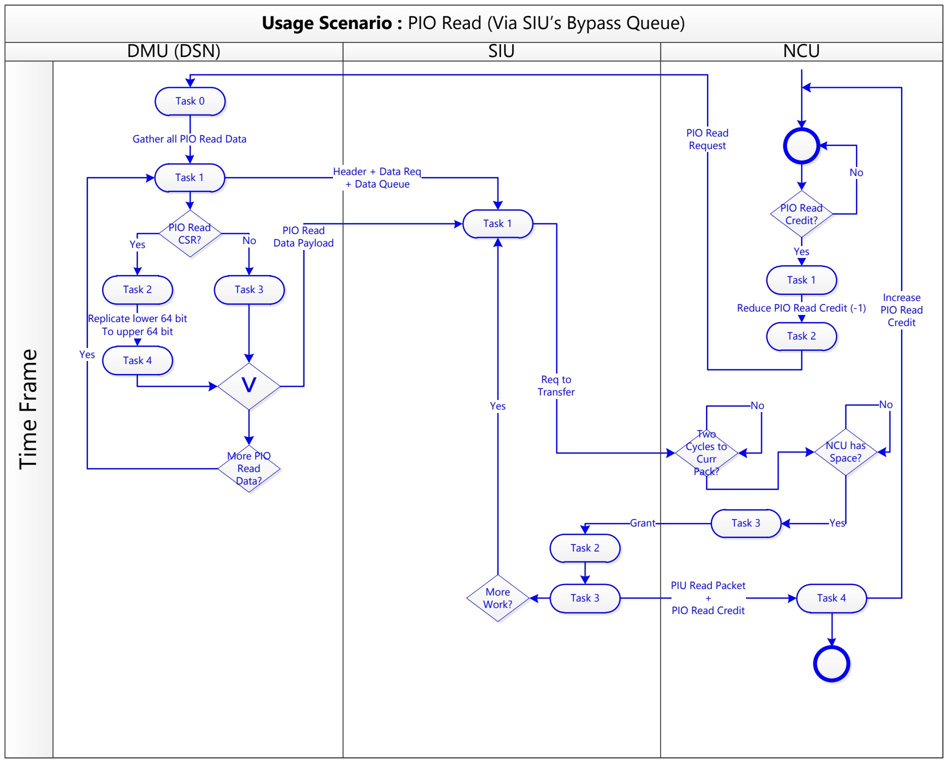

Figure A.2: PIO Read

A.3 PIO Write

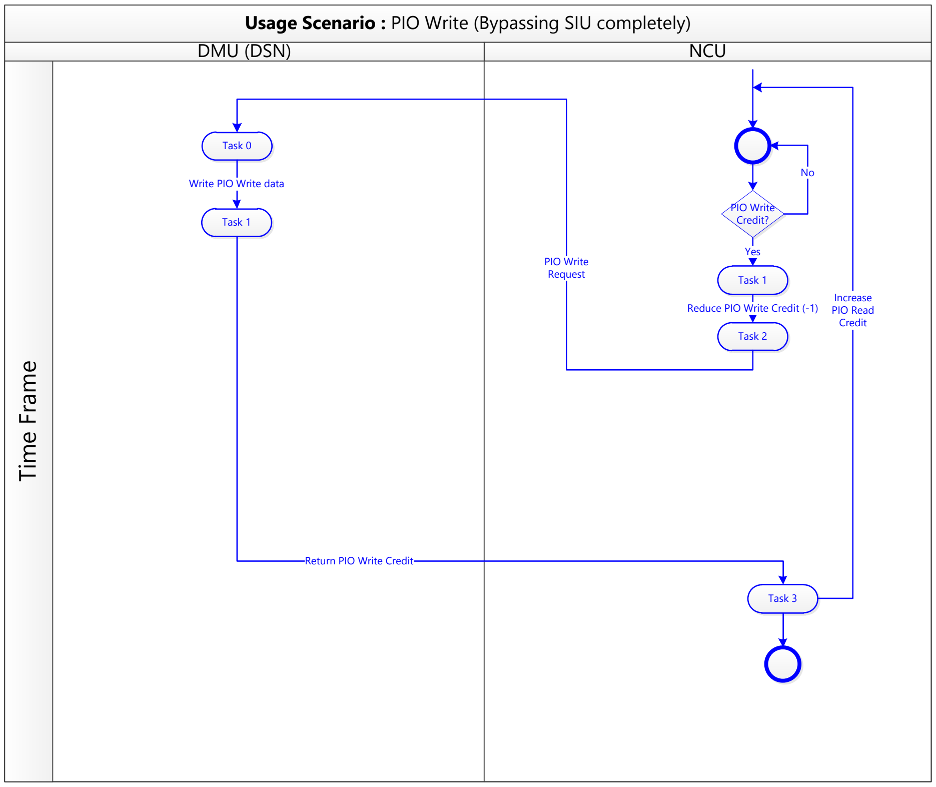

Figure A.3: PIO Write

A.4 Network Data Packet Processing

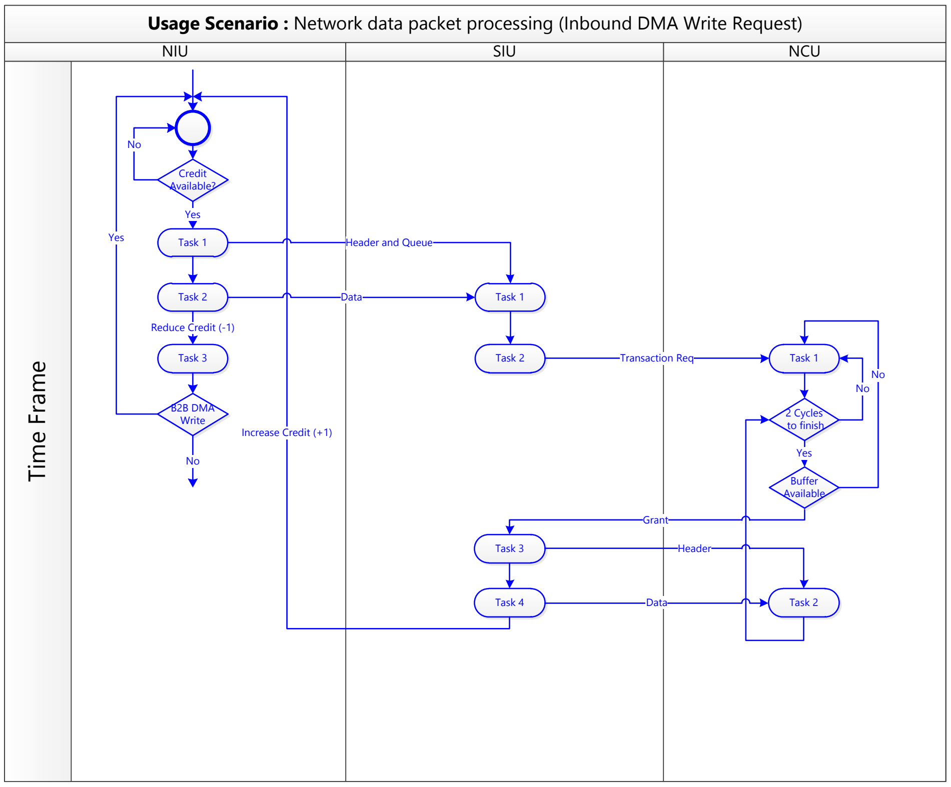

Figure A.4: Network Data Packet Processing

A.5 DMA Read Request

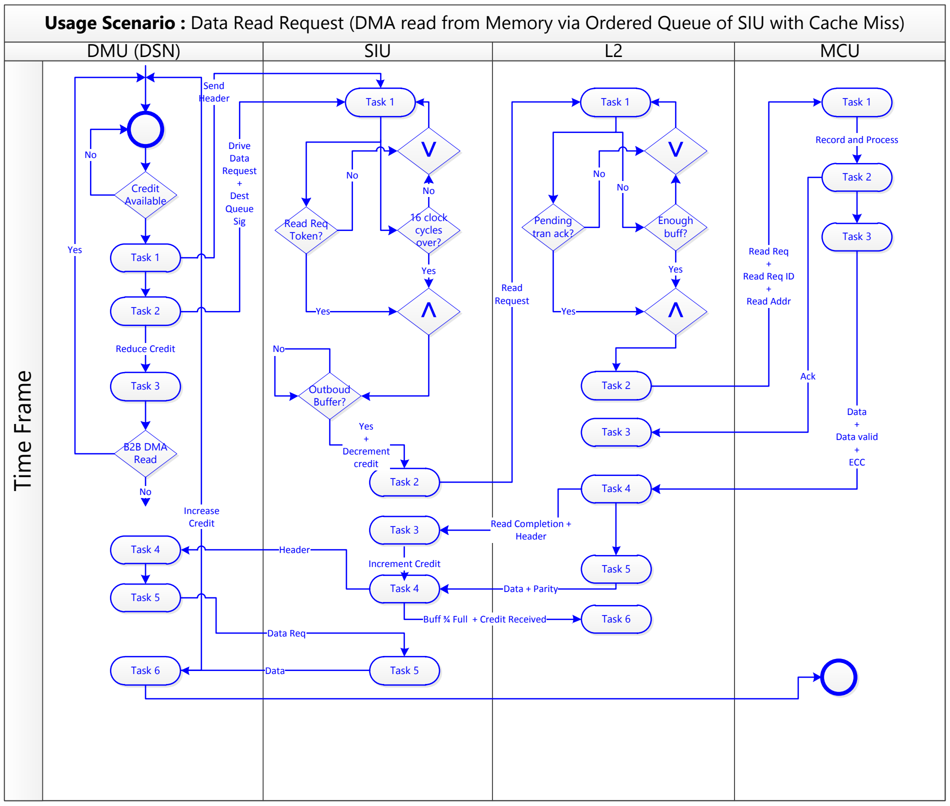

Figure A.5: DMA Read Request

A.6 CPU-Main Memory Load

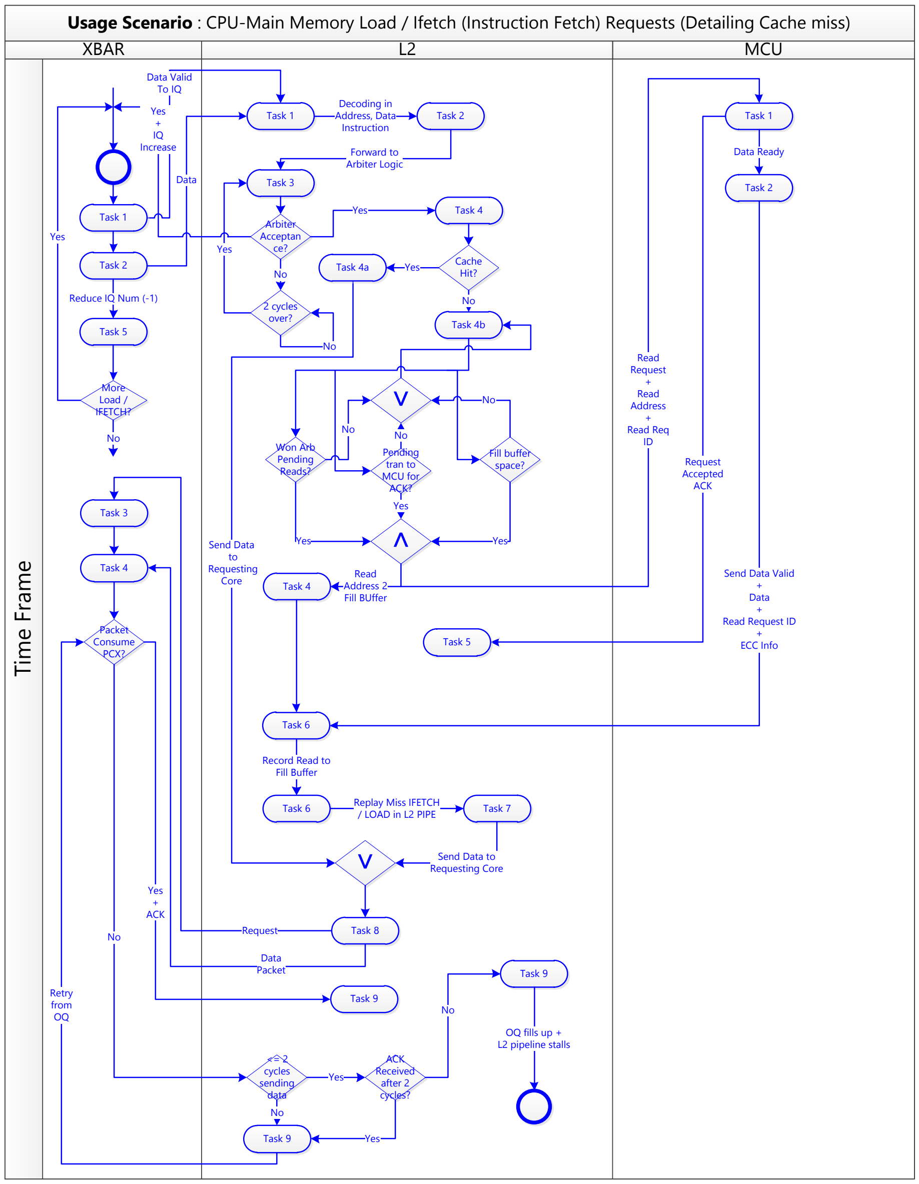

Figure A.6: CPU-Main Memory Load

A.7 NCU-XBAR Upstream & Downstream Protocols

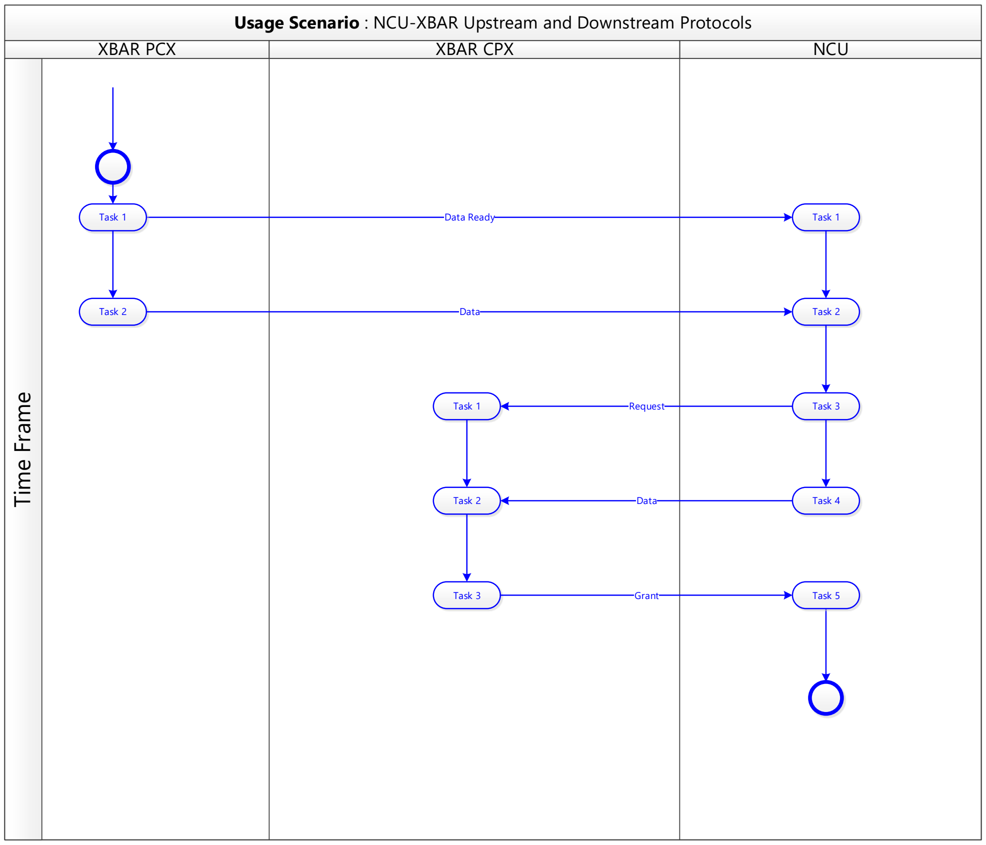

Figure A.7: NCU-XBAR Upstream & Downstream Protocols

REFERENCES

[1] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011*, pp. 1–638, Jan 2012.

[2] *IEEE Standard for System Verilog*, IEEE Standard, 2012.

[3] H. D. Foster, "Why the Design Productivity Gap Never Happened," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 581–584, IEEE Press, 2013.

[4] S. Mitra, S. A. Seshia, and N. Nicolici, "Post-silicon Validation Opportunities, Challenges and Recent Advances," in *Proceedings of the 47th Design Automation Conference*, pp. 12–17, ACM, 2010.

[5] 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*, pp. 3–7, ACM, 2010.

[6] 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*, pp. 7–12, ACM, 2006.

[7] P. Patra, "On the Cusp of a Validation Wall," *Design & Test of Computers, IEEE*, vol. 24, no. 2, pp. 193–196, 2007.

[8] S. Yerramilli, "Addressing Post-silicon Validation Challenge: Leverage Validation & Test Synergy (invited address)," in *Intl. Test Conf*, 2006.

[9] P. T. Wolkotte, J. H. Rutgers, P. K. Hölzenspies, M. Westmijze, R. Blumink, and G. J. Smit, "An Automated Design-flow for FPGA-based Sequential Simulation," *Technology Foundation STW*, 2008.

[10] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, *et al.*, "Intel Nehalem Processor Core Made FPGA Synthesizable," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 3–12, ACM, 2010.

[11] D. Lin and S. Mitra, "QED Post-silicon Validation and Debug: Frequently Asked Questions," in *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pp. 478–482, IEEE, 2014.

[12] X. Liu and Q. Xu, "Interconnection Fabric Design for Tracing Signals in Post-silicon Validation," in *Proceedings of the 46th Annual Design Automation Conference*, pp. 352–357, ACM, 2009.

[13] D. Josephson, "The Good, the Bad, and the Ugly of Silicon Debug," in *Proceedings of the 43rd annual Design Automation Conference*, pp. 3–6, ACM, 2006.

[14] S.-B. Park and S. Mitra, "IFRA: Instruction Footprint Recording and Analysis for Post-silicon Bug Localization in Processors," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pp. 373–378, IEEE, 2008.

[15] S.-B. 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*, pp. 368–373, ACM, 2010.

[16] 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*, pp. 1–10, IEEE, 2010.

[17] C. Ciordas, K. Goossens, T. Basten, A. Radulescu, and A. Boon, "Transaction Monitoring in Networks on Chip: The On-chip Run-time Perspective," in *Industrial Embedded Systems, 2006. IES'06. International Symposium on*, pp. 1–10, IEEE, 2006.

[18] I. Wagner and V. Bertacco, "Reversi: Post-silicon Validation System for Modern Microprocessors," in *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pp. 307–314, IEEE, 2008.

[19] I. Silas, I. Frumkin, E. Hazan, E. Mor, and G. Zobin, "System-level Validation of the Intel Pentium M Processor," *Intel Technology Journal*, vol. 7, no. 2, pp. 37–43, 2003.

[20] T. J. Foster, D. L. Lastor, and P. Singh, "First Silicon Functional Validation and Debug of Multicore Microprocessors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 15, no. 5, pp. 495–504, 2007.

[21] P. Lisherness and K.-T. T. Cheng, "An Instrumented Observability Coverage Method for System Validation," in *High Level Design Validation*

and Test Workshop, 2009. HLDVT 2009. IEEE International, pp. 88–93, IEEE, 2009.

[22] A. Adir, A. Nahir, A. Ziv, C. Meissner, and J. Schumann, "Reaching Coverage Closure in Post-silicon Validation," in *Hardware and Software: Verification and Testing*, pp. 60–75, Springer, 2010.

[23] H. F. Ko and N. Nicolici, "Algorithms for State Restoration and Trace-signal Selection for Data Acquisition in Silicon Debug," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 2, pp. 285–297, 2009.

[24] X. Liu and Q. Xu, "Trace Signal Selection for Visibility Enhancement in Post-silicon Validation," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 1338–1343, European Design and Automation Association, 2009.

[25] 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.

[26] H. F. Ko and N. Nicolici, "Automated Trace Signals Selection Using the RTL Descriptions," in *Test Conference (ITC), 2010 IEEE International*, pp. 1–10, IEEE, 2010.

[27] 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.

[28] 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*, pp. 352–357, IEEE, 2011.

[29] 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*, pp. 595–601, IEEE, 2011.

[30] A. Nahir, A. Ziv, R. Galivanche, A. Hu, M. Abramovici, A. Camilleri, B. Bentley, H. Foster, V. Bertacco, and S. Kapoor, "Bridging Pre-silicon Verification and Post-silicon Validation," in *Proceedings of the 47th Design Automation Conference*, pp. 94–95, ACM, 2010.

[31] H. F. Ko and N. Nicolici, "Combining Scan and Trace Buffers for Enhancing Real-time Observability in Post-silicon Debugging," in *European Test Symposium*, pp. 62–67, 2010.

[32] K. Basu, P. Mishra, and P. Patra, "Constrained Signal Selection for Post-silicon Validation," in *High Level Design Validation and Test Workshop (HLDVT), 2012 IEEE International*, pp. 71–75, IEEE, 2012.

[33] 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*, pp. 302–307, IEEE, 2013.

[34] 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*, pp. 308–313, IEEE, 2013.

[35] M. Li and A. Davoodi, "A Hybrid Approach for Fast and Accurate Trace Signal Selection for Post-silicon Debug," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 33, no. 7, pp. 1081–1094, 2014.

[36] A. DeOrio, J. Li, and V. Bertacco, "Bridging Pre-and Post-silicon Debugging with BiPeD," in *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, pp. 95–100, IEEE, 2012.

[37] M. Li and A. Davoodi, "Multi-mode Trace Signal Selection for Post-silicon Debug," in *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pp. 640–645, IEEE, 2014.

[38] S. Prabhakar and M. Hsiao, "Using Non-trivial Logic Implications for Trace Buffer-based Silicon Debug," in *Asian Test Symposium, 2009. ATS'09.*, pp. 131–136, IEEE, 2009.

[39] K. Zhao and J. Bian, "Pruning-based Trace Signal Selection Algorithm," in *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, pp. 639–644, IEEE Press, 2011.

[40] K. Rahmani, P. Mishra, and S. Ray, "Scalable Trace Signal Selection Using Machine Learning," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pp. 384–389, IEEE, 2013.

[41] H. Shojaei and A. Davoodi, "Trace Signal Selection to Enhance Timing and Logic Visibility in Post-silicon Validation," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 168–172, IEEE Press, 2010.

[42] N. Nicolici and H. F. Ko, "Design-for-debug for Post-silicon Validation: Can High-level Descriptions Help?," in *High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International*, pp. 172–175, IEEE, 2009.

[43] K. Olukotun, M. Heinrich, and D. Ofelt, "Digital System Simulation: Methodologies and Examples," in *Proceedings of the 35th annual Design Automation Conference*, pp. 658–663, ACM, 1998.

[44] C. R. Ho, M. Theobald, B. Batson, J. Grossman, S. C. Wang, J. Gagliardo, M. M. Deneroff, R. O. Dror, and D. E. Shaw, "Post-silicon Debug Using Formal Verification Waypoints," in *Design and Verification Conf*, 2009.

[45] 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*, pp. 320–325, IEEE, 2013.

[46] D. Lin, T. Hong, Y. Li, S. Eswaran, S. Kumar, F. Fallah, N. Hakim, D. S. Gardner, and S. Mitra, "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.

[47] 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*, pp. 561–566, IEEE, 2012.

[48] 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.

[49] S.-B. Park and S. Mitra, "IFRA: Post-silicon Bug Localization in Processors," in *High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International*, pp. 154–159, IEEE, 2009.

[50] K.-h. Chang, I. L. Markov, and V. Bertacco, "Automating Post-silicon Debugging and Repair," in *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*, pp. 91–98, IEEE, 2007.

[51] A. Krstic, L.-C. Wang, K.-T. Cheng, and T. Mak, "Diagnosis-based Post-silicon Timing Validation Using Statistical Tools and Methodologies," in *International Test Conference*, p. 339, IEEE, 2003.

[52] 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*, pp. 513–531, Springer, 2012.

[53] F. M. De Paula, M. Gort, A. J. Hu, S. J. Wilton, and J. Yang, "Backspace: Formal Analysis for Post-silicon Debug," in *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, p. 5, IEEE Press, 2008.

[54] M. Gort, F. M. De Paula, J. J. Kuan, T. M. Aamodt, A. J. Hu, S. J. Wilton, and J. Yang, "Formal Analysis-based Trace Computation for Post-silicon Debug," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 20, no. 11, pp. 1997–2010, 2012.

[55] M. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-signal VLSI Circuits*, vol. 17. Springer Science & Business Media, 2000.

[56] X. Liu and Q. Xu, "On Reusing Test Access Mechanisms for Debug Data Transfer in SoC Post-silicon Validation," in *Asian Test Symposium, 2008. ATS'08. 17th*, pp. 303–308, IEEE, 2008.

[57] D. Josephson and B. Gottlieb, "The Crazy Mixed up World of Silicon Debug," in *Custom Integrated Circuits Conference*, pp. 665–670, 2004.

[58] H. F. Ko and N. Nicolici, "Functional Scan Chain Design at RTL for Skewed-load Delay Fault Testing," in *Asian Test Symposium*, pp. 454–459, IEEE, 2004.

[59] G. J. Van Rootselaar and B. Vermeulen, "Silicon Debug: Scan Chains Alone Are Not Enough," in *Test Conference, 1999. Proceedings. International*, pp. 892–902, IEEE, 1999.

[60] X. Liu and Q. Xu, "On Multiplexed Signal Tracing for Post-silicon Validation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, no. 5, pp. 748–759, 2013.

[61] E. Anis and N. Nicolici, "On Using Lossless Compression of Debug Data in Embedded Logic Analysis," in *Test Conference, 2007. ITC 2007. IEEE International*, pp. 1–10, IEEE, 2007.

[62] 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*, pp. 62–67, IEEE, 2013.

[63] X. Liu and Q. Xu, "On Signal Tracing for Debugging Speedpath-related Electrical Errors in Post-silicon Validation," in *Test Symposium (ATS), 2010 19th IEEE Asian*, pp. 243–248, IEEE, 2010.

[64] Y. Lee, T. Matsumoto, and M. Fujita, "On-chip Dynamic Signal Sequence Slicing for Efficient Post-silicon Debugging," in *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, pp. 719–724, IEEE Press, 2011.

[65] B. Vermeulen, K. Goossens, R. van Steeden, and M. Bennebroek, "Communication-centric SoC Debug Using Transactions," in *Test Symposium, 2007. ETS'07. 12th IEEE European*, pp. 69–76, IEEE, 2007.

[66] H. F. Ko, A. B. Kinsman, and N. Nicolici, "Design-for-debug Architecture for Distributed Embedded Logic Analysis," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 19, no. 8, pp. 1380–1393, 2011.

[67] R. Abdel-Khalek and V. Bertacco, "DiAMOND: Distributed Alteration of Messages for On-chip Network Debug," in *Networks-on-Chip (NoCS), 2014 Eighth IEEE/ACM International Symposium on*, pp. 127–134, IEEE, 2014.

[68] E. Singerman, Y. Abarbanel, and S. Baartmans, "Transaction Based Pre-to-post Silicon Validation," in *Proceedings of the 48th Design Automation Conference*, pp. 564–568, ACM, 2011.

[69] K. Goossens, B. Vermeulen, R. Van Steeden, and M. Bennebroek, "Transaction-based Communication-centric Debug," in *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, pp. 95–106, IEEE, 2007.

[70] H. Vranken, T. G. Garciá, S. Mauw, and L. Feijs, "IC Design Validation Using Message Sequence Charts," in *Proceedings of the 26th Euromicro Conference, 2000.*, vol. 1, pp. 122–127, IEEE, 2000.

[71] A. Bunker, G. Gopalakrishnan, and K. Slind, "Live Sequence Charts Applied to Hardware Requirements Specification and Verification," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 4, pp. 341–350, 2005.

[72] R. Kumar and E. G. Mercer, "Verifying Communication Protocols Using Live Sequence Chart Specifications," *Electronic Notes in Theoretical Computer Science*, vol. 250, no. 2, pp. 33–48, 2009.

[73] M. Barnum, L. Lambrecht, and T. Ozguner, "Method and Apparatus for Guaranteeing Memory Bandwidth for Trace Data," Feb. 3 2006. US Patent App. 11/347,415.

[74] B. Mihajlović and Ž. Žilić, "Real-time Address Trace Compression for Emulated and Real System-on-Chip Processor Core Debugging," in *Proceedings of the 21st edition of the Great lakes symposium on VLSI*, pp. 331–336, ACM, 2011.

[75] *AMBA Documentation*, ARM Holdings, 2011.

[76] H. Yi, S. Park, and S. Kundu, "On-chip Support for NoC-based SoC Debugging," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 57, no. 7, pp. 1608–1617, 2010.

[77] S. Prabhakar and M. S. Hsiao, "Multiplexed Trace Signal Selection Using Non-trivial Implication-based Correlation," in *Quality Electronic Design (ISQED), 2010 11th International Symposium on*, pp. 697–704, IEEE, 2010.

[78] *OpenSPARC T2 SoC Documentation*, Oracle Corporation, 2007.