SYSTEM-LEVEL TRACE SIGNAL SELECTION FOR POST-SILICON DEBUG USING LINEAR PROGRAMMING

BY

MATTHEW AMREIN

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, 2015

Urbana, Illinois

Adviser:

Assistant 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 found in post-silicon validation has proven to be much more difficult than in pre-silicon because of the lack of the observability of all signals on chip. Trace buffers are an often-used structure in post-silicon debug that stores the state of a selected signal into an on-chip buffer, where it can be offloaded for a debugger to observe. However, because of area limitations for debug structures on chip and routing concerns, the signals that are selected to be traced must be 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 as designers can no longer fully understand the complexities that are involved within these designs. Recent work has concentrated on automating the selection of low-level signals from a gate-level analysis. In this work, we present the first automated system-level, message-based trace selection where the guiding principle is functional coverage of system-level messages. We use a linear programming formulation to find multiple solutions that allow tracing of the high-frequency messages and then further analyze these solutions using a message interval heuristic. 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. In addition, this method drastically reduces the time needed to select signals, as we automate a currently manual process.

To my parents, for their love and support.

ACKNOWLEDGMENTS

First, I thank my friends and family for providing me with encouragement while I moved through all phases of this work. I am not sure that this work would be possible without this encouragement.

I thank Sandip Ray for his help in formulating this problem and sharing his expertise in this area. Without his help, this work would never have been conceived. I thank Sai Ma for sharing her views in the area of post-silicon validation and for providing technical tools that allowed me to complete experiments. Rui Jiang also helped in providing tools that I later used. I thank Debjit Pal for his work in generating netlists and help in obtaining the restoration-based trace selection.

Finally, I thank my advisor, Dr. Shobha Vasudevan. Without her, none of this is possible. She provided me with the required connections, encouragement, and direction to complete this project. I am truly grateful for all her help and expertise that she provided for this project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

NoC	Network on Chip
SoC	System on Chip
DfD	Design for Debug
FPGA	Field Programmable Gate Array
RTL	Register Transfer Language
IC	Integrated Circuit
IP	Intellectual Property
NI	Network Interface
FC	Frequency Coverage

CHAPTER 1

INTRODUCTION

In this thesis, we first present background information on the area of post-silicon debug and the problems currently facing the area. Then, an overview of current debug structures used and current research in trace signal selection is presented. After that, we present our method and the results of our method on a test system implemented in SystemC.

1.1 IC Design Process

Integrated circuits (ICs) are designed from an initial customer requirements specification, which 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

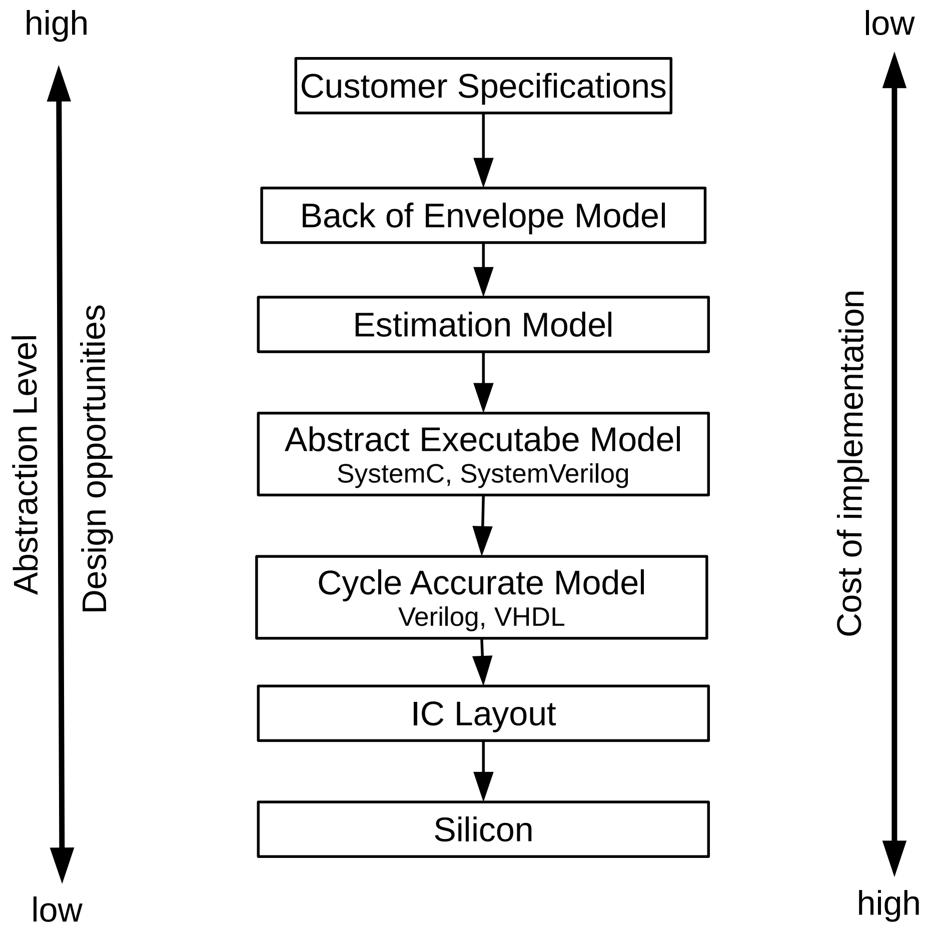

Figure 1.1: Design flow of digital IC

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 the right-hand axis indicates, the further down the design flow a design is, the more costly in terms of both time and money it is to make changes. This 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- and Post-silicon Validation

As Moore's law continues, integrated circuits become more complex and difficult to debug. Recent studies have shown that validation in modern IC designs takes up to 70% of design time in current designs and is increasing [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 validation, where access to the entire design is available for checking. However, without a substantial increase in the speed at which these tests can be run on FPGAs or emulation machines, tests for functionality have been pushed to post-silicon where they are faster to run. This, in addition to tests for electrical bugs, defined as bugs that appear due to electrical reasons such as crosstalk or vdroop, has made post-silicon validation the bottleneck in many designs [4, 5].

Pre-silicon validation can be defined as the process that is undertaken by validation engineers to ensure the correctness of a design before the design is fabricated into an actual silicon chip. Pre-silicon validation occurs at every phase of design until fabrication, but for the purposes of this work, pre-silicon validation will refer mainly to the validation 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 validation is concerned with finding and fixing functional bugs, defined as bugs in the specified behavior of the design or its implementation in RTL.

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. 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. Both pre-silicon and post-silicon validation have their own challenges and advantages that contribute to 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 only suitable 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 [6]. 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 [7]. 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 it only involves 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.

1.3 Validation Process

According to [8], 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 [9, 10].

1. **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.

2. **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 [11, 12], reducing fault latency in the initial detection [13], and other methods [14, 15].

3. **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.

4. **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.

Methods to reduce the time to find bugs have been well researched. Methods such as constrained random tests [16] and automated directed tests [17] have been successful in improving the coverage and time spent both writing and running tests. Also, research into coverage monitoring and test coverage statistics have eased the decision of when validation is complete. Fixing a bug once the root cause is known, while not always necessarily straightforward, does not consume a large portion of design time. Many times the fixes are very small, equating to a few lines in RTL or routing a single wire in the layout. Even for larger changes, the design process from the change (in either RTL or layout) is well defined and follows standard design flow. However, the process of debugging (both localizing and finding the root cause of a bug) is a less developed area in research. This, combined with the observability in post-silicon issues mentioned above, has contributed to the 35% figure for post-silicon debug cited earlier.

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

Scan chains are a DfD structure that are 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, then run the clock to output the values of a large set of flip-flops serially on a single output pin. An example of a scan chain is shown in Fig. 1.2.

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

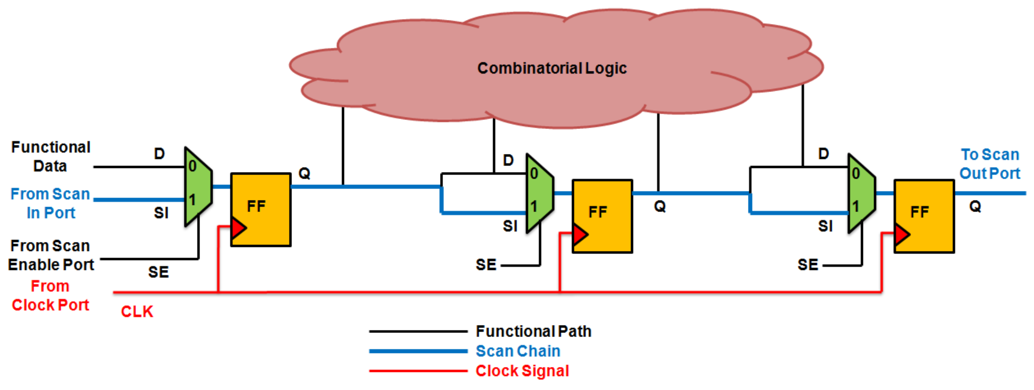

Figure 1.2: Scan chain

known, so they can also be used to detect faulty gates. Scan chains are also used in signature checking schemes and in manufacturing testing. If it is possible to restart the design from the halted state, scan chains can be used to see multiple cycles of a large set of flip flops. Unfortunately, in modern designs it is often hard enough to halt a design in a consistent manner because of issues such as multiple clock domains, let alone restart execution after halting. For this reason, scan changes are not well suited to gain temporal observability in a design. This is usually left to trace buffers.

1.4.2 Trace Buffers

Trace buffers are a DfD storage structure used to store the value of some internal signal so that it can later be offloaded for a debugger to view. 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 $N = width \cdot length$. 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. The implementation of trace buffers can vary from tracing when certain triggers activate to simply tracing at every clock cycle. Compared to scan chains, trace buffers allow us to achieve temporal observability in a design, but because of the aforementioned width restrictions, trace buffers can only observe a small number of signals, so spacial observability is sacrificed. Trace buffers are used extensively in industry and usually combined with triggers and other functionalities to create embedded logic analyzers (ELAs) [18, 19, 20]. 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, 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. 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 a restoration ratio [21, 22, 23, 24, 25]. 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. However, this metric fails to include the notion of an error or bug, 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.

Other gate-level signal selection methods have used the notion of bugs or errors in their metrics [26, 27, 28, 29, 30, 31]. 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 [8, 32, 9, 33, 34, 35, 36, 14, 37]. 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 a 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 been 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 between functional units to perform a specific task we will call *protocols*, 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 [38, 39, 40]. We define a textual format to specify MSCs and analyze all protocols and find a subset of channels 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 [41, 42]. 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 ICs, many bugs appear in the communication between already well validated IP blocks. The contributions of this work are:

1. The first 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 linear programming 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 moderate 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. Our linear programming formulation highlights key properties that we have determined to be beneficial to debug, and then selects trace signals to maximize these properties.

Our high-level trace signal selection will allow debuggers 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 can be used to further localize and debug.

The effectiveness of our method and the shortcomings of current research into this area will be shown using experimental results presented in Chapter 4. These experiments include analysis of our method using various inputs, a comparison of message-level observability between our method and current trace signal selection methods, and finally a bug case study that demonstrates the ability to localize bugs to a small portion of the design using our traces.

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 Fig. 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 are presented in Fig. 2.2.

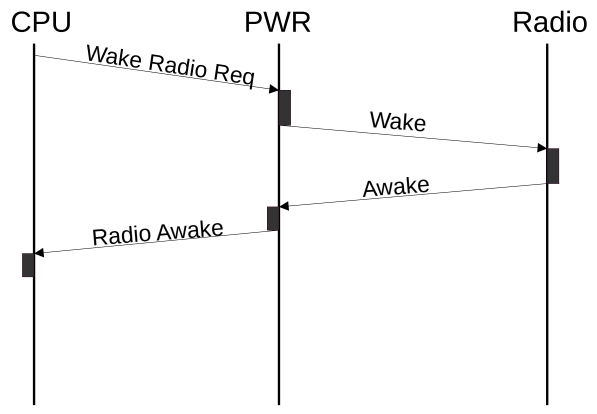

Figure 2.1: Protocol for power-on of radio block from CPU request. The vertical lines represent the three functional units and the lines between them represent messages sent on the interconnection network. Time progresses downward, hence the downward slope of the message lines.

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

channel - the physical, traceable location where the payload fields of a message pass through as they enter or leave a functional unit

view - a set of channels that are traced at the same time

conditionality - the probability that a message is sent in a given execution of a protocol

frequency coverage - value between 0 and 1 that represents the percentage of all messages that can be traced based on the frequency of occurrence

interval score - a weighted average of the average number of missing messages between traced messages for a given selection of channels

decision point - where a functional unit makes a decision where to send the next message

Figure 2.2: Reference of terms used

2.2 Messages and Channels

An example of a message with three possible payload fields of command, address, and data is shown in Fig 2.3. The header information is used by the network and contains and 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. 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. We define the locations where a field either leaves or enters a block from the network as a *channel*. For example, tracing the incoming channel of a block would mean that the command field of every incoming message to the CPU block is put into the trace buffer to later be offloaded and observed. Trace signal selection for system level, protocol-based debug in these systems is achieved by selecting a set of incoming and outgoing channels. An illustration of a channel and its tracing is shown in Fig. 2.4.

Header	Command	Address	Data

Figure 2.3: Example message format

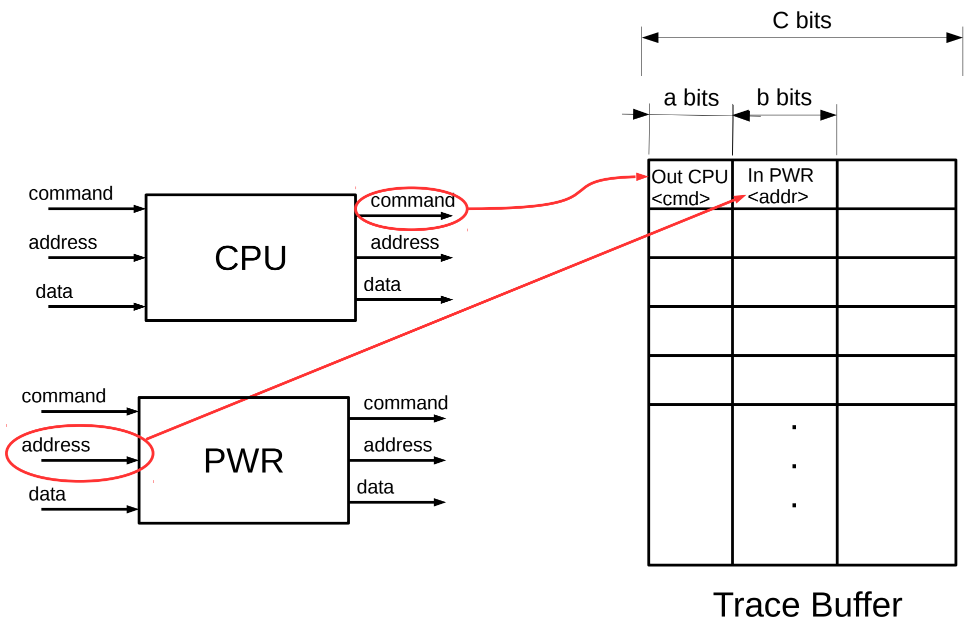

Figure 2.4: Example of how each channel is individually traced

In the design of NoCs, each channel of a block would be 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 [43]. 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 payload field is very rarely useful, so in our work we focus on tracing entire incoming or outgoing fields on messages. Each field is always a predefined amount of bits. We have defined the lengths of those three fields as follows: 8 bits for the command field and 32 bits for both the data and address fields. The number of fields and length of each field can be changed to accommodate 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 [14, 32, 44].

2.3 Trace Buffer Architecture

Currently, the process of selecting incoming and outgoing channels to trace is a manual selection process undertaken by a system designer. A common approach in trace signals is using a multiplexer to select between different sets of trace signals [26, 45, 46, 28]. Other approaches do so dynamically, but for this trace signal solution, we will use 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 channels that can help debug in post-silicon. Each set of channels, or *view*, can be selected together using the selection bits of multiplexers as shown in Fig. 2.5. The example shown has only 2 views and n-bits, but typically a system would include more views. The implementation presented later in this thesis has a total of 8 views.

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

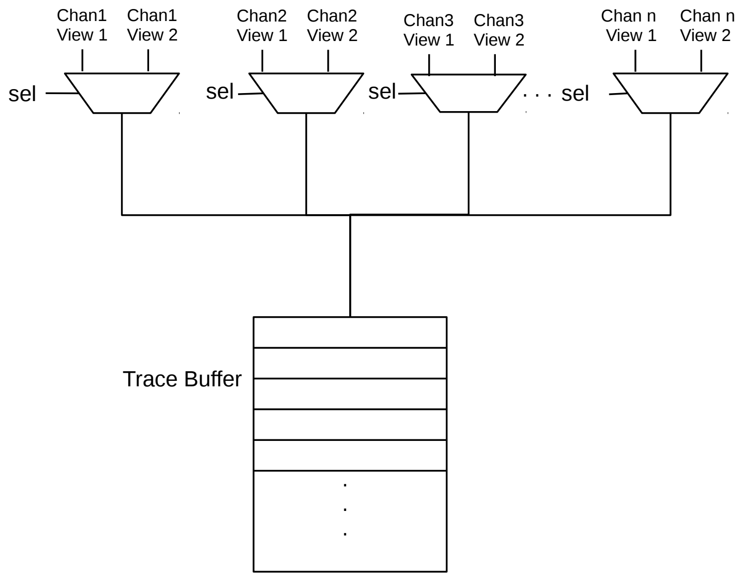

Figure 2.5: Trace buffer architecture for a 2-view, n-bit architecture

procedure undertaken by a system debugger is shown in Fig. 2.6.

The trace presented to the debugger will include the block of the channel, the value of the channel, and whether the message was outgoing or incoming. An example of the trace given to the debugger assuming the protocol in Fig. 2.1 and the trace selection is 2.4 is shown in Fig. 2.7. 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 channels 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 amount of time as the system designer or team must carefully look over each and every protocol to attempt to capture the important features. We present an automated, systematic, hierarchical, block-level, selection method for these channels that can provide

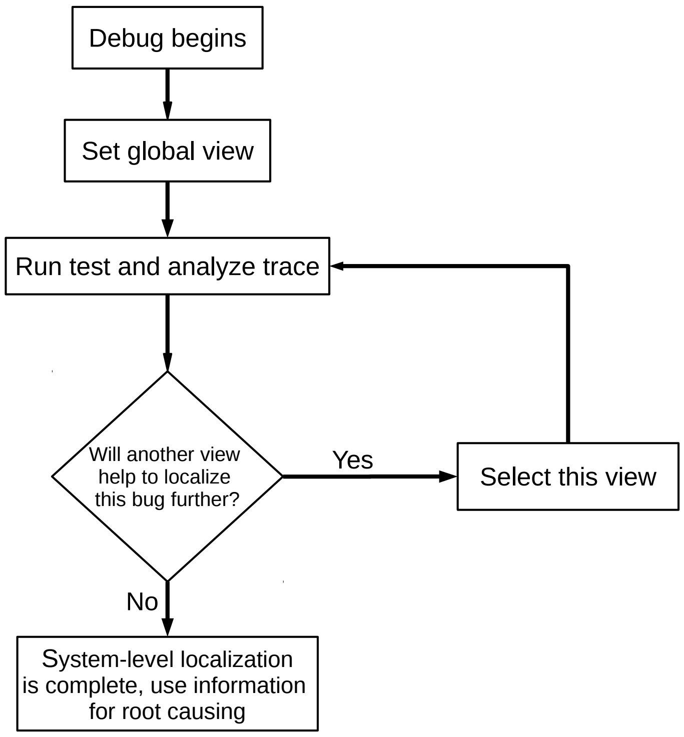
Debug begins
Set global view
Run test and analyze trace
Will another view help to localize this bug further?
Yes
Select this view
No
System-level localization is complete, use information for root causing

Figure 2.6: Usage flow chart for system-level debugger using system-level signal selection with views

high message frequency coverage and high message interval properties, both of which will be described in detail later as desirable properties for the ability to debug. Given a protocol specification and trace buffer width constraint, this selection method will provide views on a global level and block-specific level to aid system-level post-silicon debug.

Each view in our selection provides high message frequency coverage and

```
Out CPU = <Wake_Radio, [], []>
In PWR = <Wake_Radio, [], []>
In PWR = <Awake_Radio, [], []>
```

Figure 2.7: An example of the trace that would be presented to a debugger. The messages are in the form `<CMD, DATA, ADDR>`. In this case, the value of each command field has been presented as a more descriptive command name. A value of `[]` indicates a channel is not traces. A value of `XX` indicates the channel is traces, but no value has been assigned to this field for this particular message.

high message interval properties, so it is likely that each trace will provide information, in the form of missing or incorrect messages, to help localize the bug. Once this information is observed, a debugger can use this information and the known channel selection of each view to choose another view to localize the bug further. Each block-specific view also provides high message frequency coverage and high message interval properties, but these properties are on a constrained set of messages (the messages to and from the block), so the effect is a trace that captures more information specific to that block, leading to a finer-grained debug effort. In addition to global and block-specific views, we create a control view that allows protocols with large numbers of decision points to be isolated in a similar manner as we do with each block. The result of our method is an automated trace selection that not only releases system designers from the considerable amount of time needed to select signals, but also selects signals of a higher quality.

2.4 Linear Programming

In order to select the optimal channel selection based on some defined metrics, *linear programming* is used. Linear programming is a method by which some optimization, either a maximization or minimization, is achieved by creating a mathematical model that is comprised of linear inequalities and some function that is to be maximized or minimized. In other words, linear programming is a method to find a maximal or minimal optimization of some outcome given some constraints. A linear program in its canonical form is shown in Eq. 2.1.

$$\begin{aligned} \text{maximize: } & \mathbf{c^T x} \\ \text{subject to: } & A\mathbf{x} \leq \mathbf{b} \\ \text{and } & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{2.1}$$

This linear program finds a vector $\mathbf{x}$ that maximizes the $objective\ function$, $\mathbf{c^T x}$. A is a matrix of coefficients and b is a vector. The two inequalities are constraints that the solution must satisfy.

Linear programming has many applications [47] and as a result, there are a multitude of both open source and commercial linear program solvers that can efficiently solve linear programs [48, 49, 50, 51]. In this work, the open source GLPK Linear Program Solver is used [48].

CHAPTER 3

PROTOCOL BASED SIGNAL SELECTION

An overview of our selection method is shown in Fig. 3.1. The following sections will describe each step in detail.

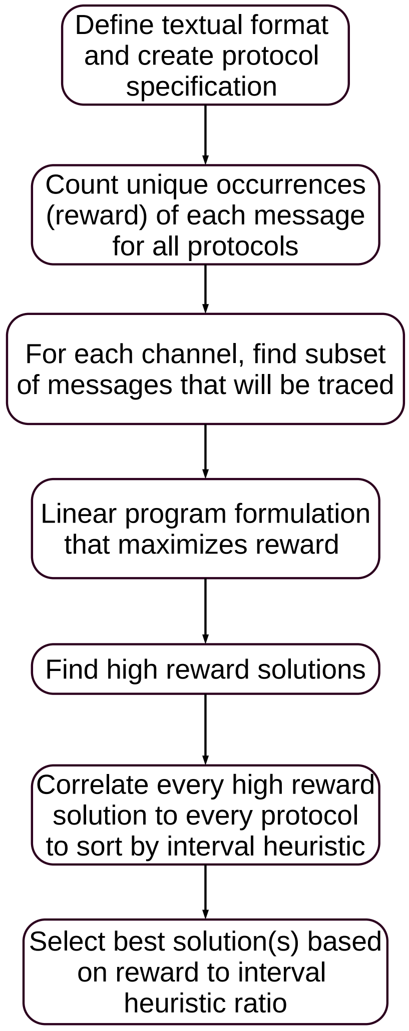
Define textual format and create protocol specification
Count unique occurrences (reward) of each message for all protocols
For each channel, find subset of messages that will be traced
Linear program formulation that maximizes reward
Find high reward solutions
Correlate every high reward solution to every protocol to sort by interval heuristic
Select best solution(s) based on reward to interval heuristic ratio

Figure 3.1: Step-by-step flow of global view and block view selection method

3.1 Textual Format and 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 Fig. 2.1 may belong 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 Fig. 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 localize any observed bugs in our design to a small set of possible problem 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.

We define three common bug types from the system level that our trace selection method will attempt to capture: 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 including a functional block simply failing to send a message, to an incorrect field on a sent message, or the setting the incorrect recipient on a message, which may cause the recipient to ignore the message and halt the protocol. A data bug is a bug that 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 could be defined 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. We will keep these bug types in mind as we continue through the explanation of our signal selection method.

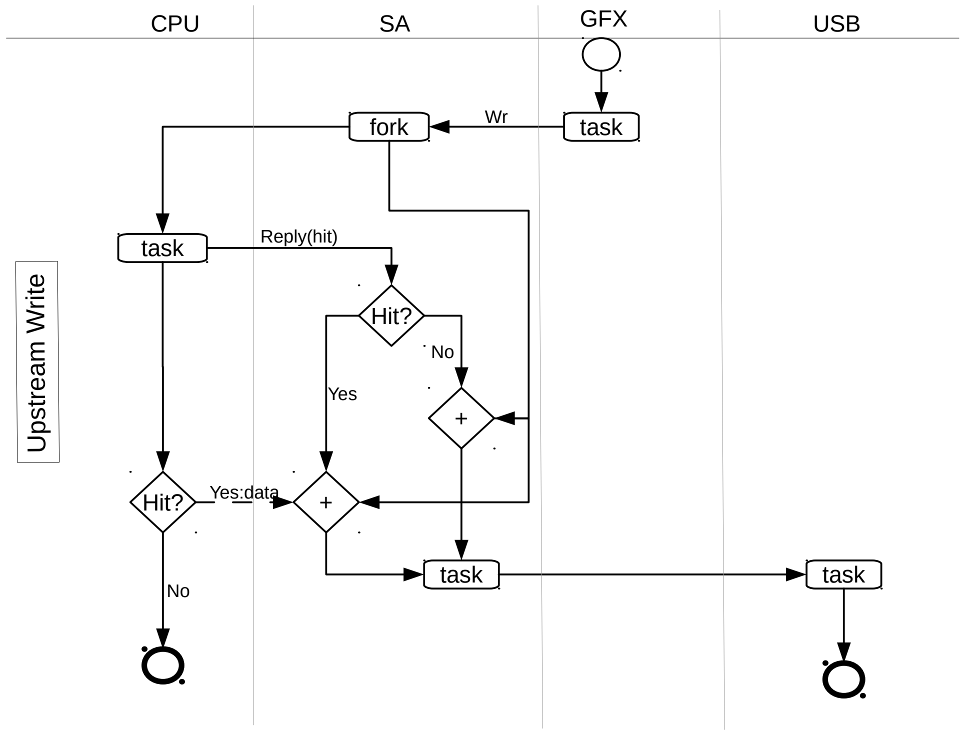

Figure 3.2: Protocol diagram for and upstream write. The symbols with marked with "+" indicate synchronization points where the all incoming messages must have arrived before continuing.

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 the ones shown in Fig. 2.1 and Fig. 3.2.

These diagrams give certain 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. We have developed a textual format to capture all this information, as shown below:

$m_i =< s_i, d_i, r_i, t_i, w_i >$
where,
$s_i =$ sender of message
$d_i =$ payload field of message
$r_i =$ recipient of message
$t_i =$ sequence time of message
$w_i =$ conditionality of message

Each message is defined as a tuple with up to five elements. The first and third element are the sender and receiver of the message, respectively. The second element is the payload field(s) that the information of the message is contained in. This must contain at least one data field, but can contain up to all the data fields that are defined. The sequence time element specifies where this message occurs in the sequence of the protocol. Multiple messages can have the same sequence time if they are being sent concurrently. The final element is the conditionality of message. This element is any number between 0 and 1 and is used to represent conditional messages as the probability that the message will be sent. If a single message in the execution of a protocol is only sent 50% of the time, then the weight for that message would be 0.5. If this element is ignored, it is assumed that the weight is 1.

The format (examples shown in Table 3.1) gives the design engineer specifying the input some flexibility in the importance of certain messages and the sequence of messages. This is intended as a way to guide the final selection if the engineer wants to emphasize certain properties of the protocol. For example, the design engineer may choose to model all conditional messages with a higher conditionality than they actually appear with in execution in order to guide the selection to treat these messages with a higher priority. This would sacrifice the observability of the common case (when the conditional message is not sent), but would provide more observability for the case when the message is sent. In addition, the engineer can change conditionality on entire protocols to emphasize either the common case or attempt to emphasize the less common cases.

While the textual format allows some flexibility, in order to understand the important payload fields in each message, the engineer creating this specification must have some knowledge of the format of the information being transmitted. For example, in Fig. 2.1, the first message is described as a

Table 3.1: Textual representation of protocols in Fig. 2.1 and Fig. 3.2

Fig. 2.1	**Fig. 3.2**
<CPU, <cmd>, PWR, 1>	<GFX, <cmd><addr>, SSA, 1>
<PWR, <cmd>, Radio, 2>	<SSA, <cmd><addr>, CPU, 2>
<Radio, <cmd>, PWR, 3>	<GFX, <data>, SSA, 2>
<PWR, <cmd>, CPU, 4>	<CPU, <cmd>, SSA, 3>
	<CPU, <data>, SSA, 4, 0.5>
	<SSA, <cmd><addr><data>, D-Unit, 5>

Table 3.2: Textual representation of upstream write family protocol

Upstream Write
Initiators: GFX, Audio
Targets: PWR, Audio, GFX, USB, UART
<I, <cmd><addr>, SSA, 1>
<SSA, <cmd><addr>, CPU, 2>
<I, <data>, SSA, 2>
<CPU, <cmd>, SSA, 3>
<CPU, <data>, SSA, 4, 0.5>
<SSA, <cmd><addr><data>, T, 5>
end

"Wake Radio Request," but to convert this protocol to our textual format, we must know that a wake request is defined by the command field. In other systems, this message may take the use both command and data fields, where the block to wake is determined by the data field. We assume that these inner-workings of the protocol are available to the designer at this stage in the design process.

As mentioned on page 21, this representation also allows for protocol families to be defined to help enumerate all protocols for a given general task. An example of an upstream write family definition is shown in Table 3.2. Initiators and targets are defined and a separate protocol will be created for each initiator-target pair (excluding the cases when Initiator and Target are the same). The characters "I" and "T" identify the initiator and target, respectively.

3.2 Count Unique Occurrences of Each Message

Incorrect data can be sent on any field at at any time, so in order to ensure that all data bugs can be captured in our trace selection, we would need to

Table 3.3: Decomposition into single payload field messages

Original	**Decomposed**
<GFX, <cmd><addr>, SSA, 1>	<GFX, <cmd>, SSA, 1>
	<GFX, <addr>, SSA, 1>
<SSA, <cmd><addr>, CPU, 2>	<SSA, <cmd>, CPU, 2>
	<SSA, <addr>, CPU, 2>
<GFX, <data>, SSA, 2>	<GFX, <data>, SSA, 2>
<CPU, <cmd>, SSA, 3>	<CPU, <cmd>, SSA, 3>
<CPU, <data>, SSA, 4, 0.5>	<CPU, <data>, SSA, 4, 0.5>
<SSA, <cmd><addr><data>, USB, 5>	<SSA, <cmd>, USB, 5>
	<SSA, <addr>, USB, 5>
	<SSA, <data>, USB, 5>
end	end

observe all sent and received payload fields in our set of protocols. Because of limited trace buffer space, observing all payload fields is not possible in most designs and therefore an optimization may take place to observe the maximum number of messages. To formulate this optimization problem, we need to enumerate the occurrences of each message in all protocol descriptions as described above. First, we enumerate all protocols by taking each protocol family definition and creating the entire protocol family. Then, we decompose all messages defined in our input format into messages with only a single data field for easier analysis. An example of this decomposition is shown in Table 3.3. After doing this, we add up the conditionality of each message occurrence, such that each tuple in the form `<sender, payload field, receiver>` occurs only once, with a reward, which is the sum of the conditionality.

3.3 Find Subset of Messages for Each Channel

Each channel covers a subset of all the decomposed messages. To find this subset we enumerate all channels as a tuple in the form `<sender, payload field, 0>` for outgoing channels and `<0, payload field, receiver>` for incoming fields. Each channel also has a cost associated with it, which is the length of the data field.

The function $e(q_i, m_j)$ defines whether a given channel, q_i, covers a message, m_j. If either the sender or receiver of a message, along with the payload field, matches the channel, then the message can be observed. We can then say that a set of channels Q covers a message if at least one channel within it covers a message. These are formally defined in Eq. 3.1.

$$\begin{aligned} e(p_i, m_j) &= \begin{cases} 1, & \text{if} \quad d_i = d_j \text{ and } (r_i = r_j \text{ or } s_i = s_j) \\ 0, & \text{otherwise} \end{cases} \\ f(Q, m_j) &= \begin{cases} 1, & \text{if for at least one } q_i \in Q,\ e(q_i, m_j) = 1 \\ 0, & \text{otherwise} \end{cases} \end{aligned} \tag{3.1}$$

3.4 Linear Program Formulation

We define the frequency coverage (FC) of a set of channels to be observed, Q in Eq. 3.2.

$$FC = \frac{\sum\limits_{\forall m_j \in M} f(Q, m_j) \cdot r_j}{\sum\limits_{\forall m_i \in M} r_i} \tag{3.2}$$

In this definition, r_i is the reward of message i. Frequency coverage should be a value between 0 and 1, where 1 indicates all messages are covered and 0 indicated none are covered. Imposing a cost on each channel creates the budgeted maximum coverage problem, where the goal is to maximize frequency coverage under a certain cost constraint C, the width of the trace buffer. This is formulated into a linear programming problem as shown in Eq. 3.3.

$$\begin{aligned} \text{maximize :} \quad & \sum_{\forall m_i \in M} r_i y_i \text{ (maximize frequency coverage)} \\ \text{subject to:} \quad & \sum_{\forall q_i \in Q} c(q_i) x_i \le C \text{ (cost constraint)} \\ & \sum_{\forall q_i \in Q} x_i \le y_j \quad \forall m_j \in M \\ & 0 \le y_i \le 1 \\ & x_i \in \{0, 1\} \end{aligned} \tag{3.3}$$

Defined in this manner, we can see that this problem is the same as the budgeted maximum coverage problem [52], which can be solved using linear programming. The function $c(q_i)$ defines the cost for channel q_i. The y and x variables are simply indicator variables where $y_i = 1$ corresponds to message m_i being covered and $x_i = 1$ corresponds to channel q_i being selected. The second constraint equation sets all y variables (to 0 or 1) based on the selection of x variables. The solution for the problem is the selection of x variables.

3.5 Find High Reward Solutions

The linear program formulated above was solved using the GNU Linear Programming Kit (GLPK) [48]. The optimal solution was found and saved, along with the reward value. Then, the optimal solution was removed from the possible set of solutions by adding an additional constraint to the linear program. Another solution is found and also recorded. It is removed from the set of possible solutions by adding another constraint. This process is repeated until all solutions with reward values within 95% of the optimal solutions are found and recorded. The parsing to all solutions within 5% of the optimal reward value is intuitively based on the fact that as frequency coverage increases, interval scores (introduced in the next section) should decrease because as more messages are observed, we are likely to decrease the average interval between observed messages. In Chapter 4 we will show that it is very likely that the final solution is within this parsed set of solutions.

3.6 Message Interval Heuristic

For halting bugs, in addition to observing as many messages as possible, a debugger may also want to observe messages in sequence on a regular basis in order to observe when a possible divergence of the correct protocol has occurred. For example, if a protocol contains nine messages, each one directly after the other, observing only the first four messages would not be valuable to observe a divergence from the protocol in the second half of the execution. However, observing the first, third, fifth, seventh, and ninth message would allow us more flexibility to observe when the execution diverges with some small error latency. This is shown in Fig. 3.3. In addition to the spacing, observing the first and last message of each protocol is important so that a debugger knows when each protocol has started and ended, so extra consideration will be taken for these two messages.

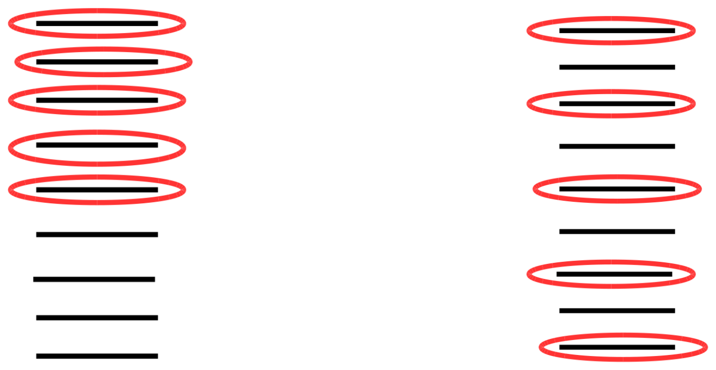
(a) Non-ideal spacing (b) Ideal spacing

Figure 3.3: Spacing example for a nine message sequence with four messages covered. Red circles indicate message is observable.

Assuming equal frequency coverage for two channel selections, the solution with a more ideal spacing should be chosen. To obtain a heuristic for comparing the interval spacing in different solutions, the iterative algorithm shown in Fig. 3.4 was used.

This iterative algorithm is run on each protocol after all covered messages are determined from the channel set, Q. The value returned from this function will be averaged over all protocols and called the interval score. The interval score does not give the exact average interval size missed over all protocols, but rather a weighted average. Determining the start and end of a given protocol is important, so the first and last sequence times are weighted twice as much as others. Also, sequence times that only contain conditional messages should be weighted lower, so the weights of all messages in a sequence time are added, up to a maximum of 1. As an example, the interval score for the channel selection in Fig. 3.3a would be 5. There is one large interval, and the penalty for each missed sequence time is 1, except for the last sequence, where the penalty is 2. The interval score for Fig. 3.3b would be 1, as there are 4 intervals, each one with a penalty of 1.

T is a set of the covered sequence times in a protocol
t_f is the final sequence time in algorithm

```
procedure AVERAGE-INTERVAL($T$, $t_f$)
    $int\_count = 0$
    $int\_num = 0$
    for $i = 1..t_f$ do
        if $i == 1$ && $i \notin T$ then                ▷ Missing initiating message(s)
            $int\_count++$
            $weight = 0$
            for all messages, $m_j$ with $t_j == 1$ do
                $weight\ += 2 * w_j$
            end for
            if $weight > 2$ then
                $weight = 2$
            end if
            $int\_num += weight$
        else if $t_i \notin T$ then                ▷ There is a missing sequence time
            $weight = 0$
            for all messages, $m_j$, with $t_j == i$ do
                $weight\ += w_j$
            end for
            if $weight > 1$ then
                $weight = 1$
            end if
            $int\_count = int\_count + w_i$
            $int\_num = int\_num + t_i - t_{i-1}$
        else if $i == t_f$ && $t_f \notin T$ then                ▷ Missing ending message(s)
            $int\_count++$
            $weight = 0$
            for all messages, $m_j$ with $t_j == t_f$ do
                $weight\ += 2 * w_j$
            end for
            if $weight > 2$ then
                $weight = 2$
            end if
            $int\_num\ += weight$
        end if
    end for
    return $int\_num/int\_count$
end procedure
```

Figure 3.4: Algorithm for interval score heuristic

3.7 Select Best Solution(s)

With metrics to determine frequency coverage and interval observation intervals, the iterative algorithm shown in Fig. 3.5 is used to produce either one unique solution or a set of equal solutions for the user to choose from. The goal is to find a solution with a balance of high frequency coverage and low interval score. The general trend, as will be shown later, is that higher frequency coverage should lead to lower interval scores, but this is not always the case, so we must attempt to balance both frequency coverage and interval score when looking for a final solution. For an ideal solution, the ratio $I/(1 + FC)$ is as small as possible, where I is the interval score and FC is the frequency coverage. In the case that this ratio is equal, the solution with a higher frequency coverage is chosen.

The output is either one unique solution or a set of equal solutions. Given an input of the entire set of protocols for a design, we will call the solution(s) from this algorithm the "global view". This will be the base view used by a debugger and should allow for the most general observability. We will define other views later.

3.8 Block-Specific Views

After localizing a bug to a smaller set of blocks using the global view, most debuggers would like to look at each of these possible root cause blocks separately to determine the exact root cause. We achieve this functionality by creating a separate view for each block in the design. Each block will have its own set of protocols defined, found from the original set of system protocols. Figure 3.6 shows a high-level view of how this done.

Each message in a protocol is added to two lists of messages: one list for the sender and one for the receiver. This is done for all messages in a protocol. Once the end of the protocol is reached, these lists have their sequence times reordered such that they represent the order in that set of messages only. For example, if a list for a specific block had messages with sequence times of 3, 3, 6, and 8, they would be reordered to 1, 1, 2, and 3, respectively. Each list is output to a group view file for each block. Once all protocols are iterated over, all view files will be complete and the original frequency

```
            $cov(S_i)$ is the frequency coverage of solution $S_i$
            $int(S_i)$ is the interval score of solution $S_i$
            $S_{opt}$ is the optimal frequency coverage solution.
1: procedure Solution Comparisons($T$, $t_f$)
2:     Use linear programming method to produce a set of solutions $S$ that
   have reward values $\geq 95\%$ of the optimal solution's reward value
3:     $soln_{cov} = cov(S_{opt})$
4:     $soln_{int} = int(S_{opt})$
5:     Push $S_{opt}$ onto $soln\_array$
6:     for $\forall S_i \in S$ do
7:         if $\frac{int(S_i)}{1+cov(S_i)} < \frac{sol\_int}{1+soln\_cov}$ then
8:             $soln\_cov = cov(S_i)$
9:             $soln\_int = int(S_i)$
10:            Clear $soln\_array$
11:            Push $S_i$ onto $soln\_array$
12:        else if $\frac{int(S_i)}{1+cov(S_i)} == \frac{sol\_int}{1+soln\_cov}$ && $cov(S_i) > soln\_cov$ then
13:            $soln\_cov = cov(S_i)$
14:            $soln\_int = int(S_i)$
15:            Clear $soln\_array$
16:            Push $S_i$ onto $soln\_array$
17:        else if $\frac{int(S_i)}{1+cov(S_i)} == \frac{sol\_int}{1+soln\_cov}$ && $cov(S_i) == soln\_cov$ then
18:            Push $S_i$ onto $soln\_array$
19:        end if
20:    end for
21:    return $soln\_array$
22: end procedure
```

Figure 3.5: Coverage and interval solution comparisons

```
1: $P$ is the entire set of system protocols
2: procedure Protocol-Grouping($P$)
3:     for $\forall p_i \in P$ do                          ▷ for all protocols
4:         for $\forall m_i \in p_i$ do          ▷ for all message in protocol
5:             Push $m_i$ onto msg_list[$s_i$]
6:             Push $m_i$ onto msg_list[$r_i$]
7:         end for
8:         Reorder time sequence in all message lists
9:         Output all message lists to correct block view file
10:    end for
11: end procedure
```

Figure 3.6: Message grouping algorithm

coverage-interval algorithm will be done on each block-specific view file to generate a trace signal selection view for that block. Intuitively, this method attempts to capture all communication with each block in the most efficient manner (most efficient use of trace signals) possible.

3.9 Control View

In addition to data and halting bugs, some functional units may have bugs that cause messages to be sent to the wrong destination or messages that may accidentally be sent to more destinations than are correct. We will call these types of bugs control bugs. While it may be possible to observe where messages have been sent using the above frequency coverage and message interval solution, this requires observing both the sender and all possible receivers. Because a message can be sent to a large set of destinations, it is not feasible to observe all receivers. Therefore, we attempt to create another view that will attempt to observe control bugs by tracing a combination of outgoing channels along with the destination (within the header of the message) for each message and incoming channels. Our method of selection for this set will a) determine what channels have the most "decision" messages sent across them and b) how we may observe the maximal number of decisions. An overview of the method for selecting our control view is shown in Fig. 3.7.

3.9.1 Decision Points

A decision point within a protocol is where a functional unit makes a decision where to send the next message. Up until the decision, every execution of this protocol family will be the same. We use our previous example of a simple power-on protocol to show a decision point in Fig. 3.8. It is at this point that the protocol will differ from other executions of the protocol within the power-on protocol family. For example, another power-on execution may send a wake message to a block other than the Radio block. The decision was made within this execution to send this message to the Radio block. In the upstream write protocol shown in Fig. 3.2, there are two decisions points: one where the SSA block sends the final data, and another when the CPU

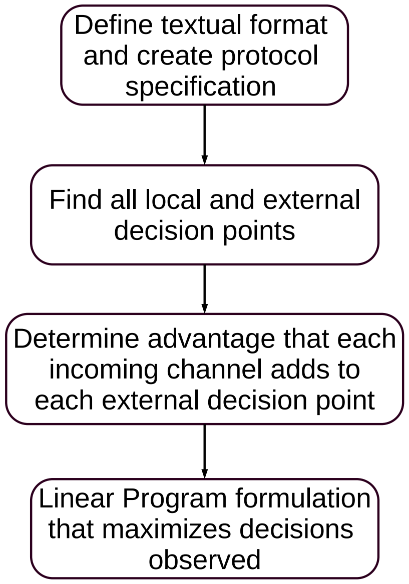
Define textual format and create protocol specification
Find all local and external decision points
Determine advantage that each incoming channel adds to each external decision point
Linear Program formulation that maximizes decisions observed

Figure 3.7: Step-by-step flow of control view selection method

sends its conditional hit data.

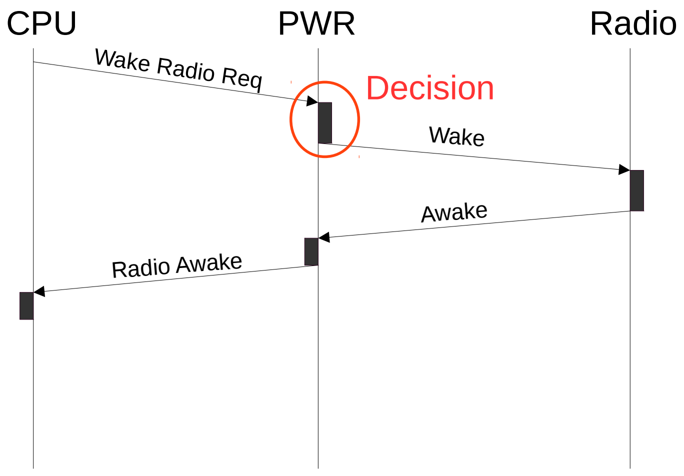

Figure 3.8: Protocol for power-on of radio block from CPU request with decision point shown

We can further place decision points into two different categories: external and local decisions. External decisions are those decisions that determine where to send a message based entirely on the previous information passed to it during the protocol. Local decisions are those decisions that determine where to send a message based on a local state at that functional unit. The two examples earlier in this chapter have external decisions; the PWR block sends its message (the wake message) based on the message sent to it by the CPU (the wake request) in the power-on protocol. In the upstream write protocol, the address that the SSA is sent from the initiator determines where the final data is sent to. An example of a local decision could be when a block selects one recipient of the next message randomly from a set of possible recipients. A case where a block selects from multiple sets of recipients based on the prior information it receives in the protocol, but randomly chooses one recipient from each, would classify as both a local and external decision.

To understand whether an external decision is correct in a trace, the debugger would need to know what information was sent to the block during the protocol, as this determines the decision. Therefore, observing an external decision point is only helpful if some or all of the incoming channels of that block are also observed. Local decisions do not depend on prior external information and therefore do not need incoming channels to also be observed.

3.9.2 Find Decision Points

The first step is the same as the previous method for the global and block view selection, so we move onto the second step. To select a set of channels for control bugs, we first must find all external and local decisions. To find all local decisions, we find all conditional messages (those with a conditionality < 1) at a specific sequence time, and determine the different recipients at that point. The messages with a conditionality less than 1 represent the probability of selecting one recipient out of a set of recipients. For example, at some point in a protocol, a functional unit may decide between two recipients randomly. Each of these messages would be defined in our format as having a conditionality of 0.5.

For external decisions, we enumerate all protocol families and then, within each protocol family, find all messages where the sender, data fields, and sequence time values match, but the recipient differs. This means that the sender can send to different recipients, but it is based on the protocol execution up to that point. All of these points in each family are added up to get the number of decisions on each outgoing channel.

3.9.3 Determine Advantage of Each Incoming Channel

Now, we need to determine the value that tracing each incoming channel adds to each external decision point. Observing an external decision point is only useful if we have information that was sent to the block that is making the external decision prior to that decision. We say that each incoming channel adds an *advantage* to each external decision. Now that each external decision point is known, each protocol execution where an external decision is made is analyzed. The overall number of incoming messages into the decision-making block before any decision is determined. Then, the number of decisions made overall at this decision point is determined. Each incoming message then gets an advantage equal to the number of decisions divided by the number of messages. As an example, take the following protocol family:

```
Targets: GFX, Audio, USB, PWR, UART
<CPU, <cmd><addr><data>, SSA, 1>
<SSA, <cmd><addr><data>, T, 2>
end
```

There are a total of 5 executions (one for each target) and therefore 5 decisions made on the command, address, and data fields of SSA. The number of messages is determined after the protocol is decomposed, so the total number of messages before the decision is 3*5=15. Therefore, each incoming message that can be traced adds an advantage of 5/15=0.3333 for this decision point. Observing all 15 messages would give us an advantage of 15*0.333=5 for this decision point, or, the number of decisions. Therefore, if we observe any one incoming channel (either command, address, or data), its advantage is 1.667 for this decision point. We see in this example how for each decision point, we need to observe all incoming messages to gain the full advantage of observing this decision. If only 10 of the incoming messages were observed (by tracing two of the incoming channels), we would not gain the full advantage of observing this decision point, so the value would only be 10*0.3333 = 3.333 instead of 5.

After the value of each incoming channel for each decision point is determined, we combine the total number of decisions for each outgoing channel and the advantage each incoming channel adds to that outgoing channel. For example, if the outgoing command channel of the CPU has 40 external decisions decided on it, the incoming command channel may add an advantage of 10, the incoming data channel may add an advantage of 10, and the incoming address channel may add an advantage of 20. The advantage of all incoming channels will add up to the number of external decisions on that outgoing channel.

3.9.4 Formulate Linear Programming Problem

Now that each outgoing channel has advantage values that are determined by what incoming channels are selected, we can formulate another linear programming problem to solve for our set of channels to trace as shown in Eq. 3.4. Note that we must add the tracing width of the recipient field in our trace for any external decision outgoing channel traced. We assume this to be 8 bits. Just as before, we will output all solutions within 5% of the optimal solution. In this case, it is to allow the designer to select from a set of roughly equal solutions. The designer should choose channels that may not be well represented in the other views so that all the views together cover as many bugs as possible. Future work may involve automating this by including other previously selected views in the selection of a new view.

$$
\begin{aligned}
&\text{maximize: } \sum_{i=1}^{n} d_i \cdot dl_i + \sum_{i=1,\ k=1}^{i=m,\ k=n} w_{i,k} \cdot a_{i,k} \text{ (maximize decisions)}\\
&\text{subject to: } \sum_{i=1}^{n} cd_i \cdot d_i + \sum_{i=1}^{m} cm_i \cdot x_i \leq C \text{ (cost constraint)}\\
&\qquad \forall (i \in (1..m),\ \ k \in (1..n))\ \ (x_i + d_k) \cdot 0.5 \geq a_{i,k} \text{ (sets indicator matrix } a)\\
&\qquad (a_{i,k}, x_i, d_k) \in \{0,1\}\\
&\text{where:}\\
&n \text{ is the number of outgoing decision channels}\\
&m \text{ is the number of incoming channels}\\
&w_{i,k} \text{ is the advantage of each incoming channel } i \text{ on outgoing decision channel } k\\
&dl_k \text{ is the local decisions on outgoing decision channel } k\\
&n \text{ is the number of outgoing decision channels}\\
&cd_k \text{ is the cost of outgoing decision point}\\
&C \text{ is the width of trace buffer}\\
&cm_i \text{ is the cost of incoming channel } i\\
&x_i \text{ is the indicator variable of incoming channel } i\\
&d_k \text{ is the indicator variable of outgoing decision point } k\\
&a_{i,k} \text{ is the indicator variable for incoming channel } i \text{ and outgoing decision channel } k
\end{aligned}
\tag{3.4}
$$

CHAPTER 4

IMPLEMENTATION, RESULTS, AND CONCLUSION

4.1 Implementation

The main test platform for our trace signal selection method is a SystemC-TLM [53] model that consists of a small SoC and implements a set of protocols that would typically be seen in modern SoC designs. A block diagram of the model is shown in Fig. 4.1.

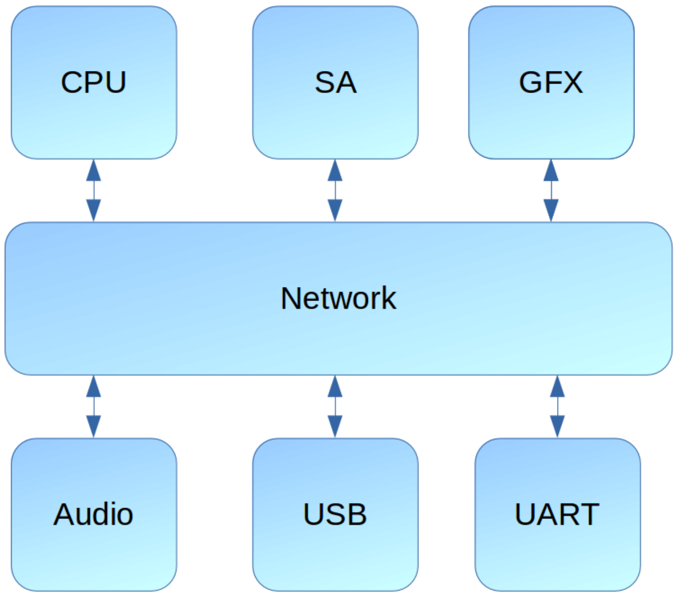

Figure 4.1: SystemC model

This system consists of 7 total blocks: A CPU, a graphics (GFX) block, an audio block, a serial UART controller, an MMC/USB controller, a power controller (PWR), and a system agent (SA). The CPU block also contains the main memory. Much of the communication in this system goes through the system agent block, which is responsible for maintaining system status and routing messages to correct places. All blocks have internal memory that is mapped to an address range that can be read by anyone and written to by certain blocks. The CPU is able to send reads and writes. These are called downstream reads and writes, as they are downstream from the CPU and do not require a check within the CPU cache. The GFX and Audio blocks are allowed to both read and write as well, but these are upstream reads and writes, and all data must be checked within the CPU cache. The PWR block is responsible for powering on all blocks and the USB and UART blocks can only initiate upstream reads, but cannot write. All blocks may issue a machine service interrupt (MSI) at any time. All of the protocols implemented in this system are shown in Appendix A. The CPU, GFX, and Audio blocks act as master/slave components that both initialize and respond to transactions. The USB, UART, and PWR blocks act primarily as slave blocks that respond to any transactions involving them, but cannot initiate transactions. The SA routes messages and keeps track of what components are powered on. If any components are powered off, it will power them on before a message is sent to them by sending a request to the PWR block. For the purposes of this work, a transaction and protocol are the same: a series of messages that complete some system-level function.

The system was implemented using the standard blocking interface included in the TLM standard. Extensions were added to the standard generic payload to allow for the needed communication fields. As transactions are often split in modern bus-based interconnect designs [43], and certainly in NoC interconnections by use of an ID or tag, the use of a blocking interface assumes that our messages can be reordered on a transaction basis before being presented to the debugger. A straightforward approach to this may be to record the IDs and some global time vector of each traced message and then reorder the traces off-chip or simply present the IDs to the debugger, but this introduces a storage overhead in the already constrained trace buffer. Other methods to order the messages on a transaction basis or to reduce the overhead that storing IDs would cause can be formulated and will be the focus of future work. One example is a dynamic tracing method that only traces messages as they arrive or leave an NI instead of at every clock cycle, which would be similar to already proposed trigger architectures [54, 33]. This could allow space for IDs to be stored.

The results section is outlined as follows: algorithm analysis experiments, comparison to gate-level selection experiments, and finally bug case studies with localization results.

4.2 Algorithm Analysis

To understand how our method provides solutions and how decisions made during protocol specification can affect the output, two experiments were performed.

4.2.1 Pruning Analysis

The first experiment done is a study of the relationship between frequency coverage and distance from the optimal solution for solutions generated by GLPK, shown in Fig. 4.2. The R^2 values of the linear fits in Fig. 4.2 are shown in Table 4.1. The optimal solution was found for each set of solutions using the frequency coverage to interval score ratio. The distance of each solution is defined as $\frac{I_{opt}}{1+FC_{opt}} - \frac{I}{1+FC}$, where I_{opt} and FC_{opt} are the interval score and frequency coverage of the optimal solution and I and FC are the interval score and frequency coverage of the current solution. This experiment's purpose is to demonstrate that our pruning of all solutions to those within 5% of the optimal solution does not remove, or very rarely removes, solutions that would be chosen at the end of our algorithm.

Table 4.1: R^2 values of linear fits in Fig. 4.2

Solution Set	**R^2 Value for linear fit line**
24 bit	0.6732
32 bit	0.2094
72 bit	0.00857
144 bit	0.1310

These results show that over a large range of frequency solutions, the optimal solution is likely to be near the top of the solution range. There is a trend towards a direct relationship between frequency and distance from the optimal solution for all but one of the trace buffer sizes, and in that case the fit is very poor. The stronger fits were seen for the sets that had a larger range of frequency values. The optimal solution in each solution

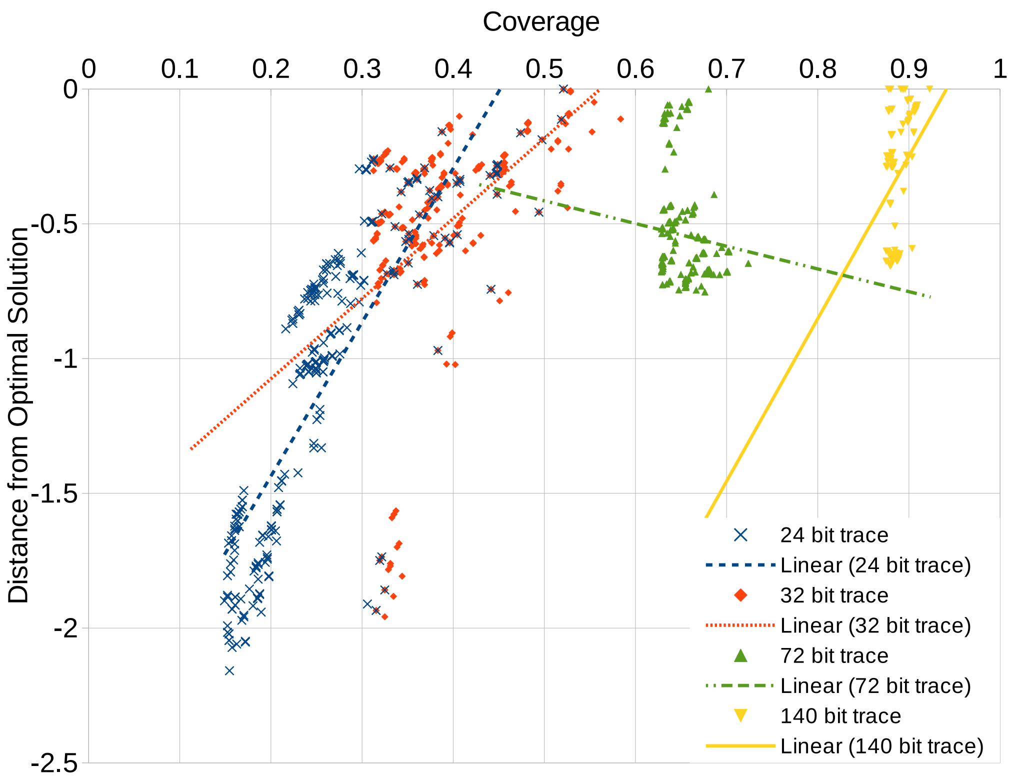

Figure 4.2: Relationship between frequency coverage and distance from optimal solution for the top 500 solutions at three different trace buffer sizes

set can be identified as the point(s) along the x-axis. A qualitative analysis suggests that these points also always seem to have a frequency coverage near the top of the range of all solutions in that set. The 144 bit trace selection has many points along the x-axis because there were many solutions with an interval score of 0. However, in this case, the solutions with highest frequency coverages are always selected. This shows that the pruning of all coverage solutions to the top 5% is unlikely to remove any solutions that would be selected after doing interval score calculations and comparisons. For more certain accuracy, a larger percentage of the top solutions could be used at the cost of run time.

4.2.2 Protocol Specification Sensitivity

Next, a study is done on the final frequency-interval score solutions for three different sets of protocols. The first protocol set, SoC, is the one used and implemented in our SystemC model. The next two are the same directory based cache coherence scheme [55], but differ in the use of conditional messages in their specification. The protocol specifications are listed in Appendix A. The cache coherence scheme specification is for a 6 CPU system with 3 home directories. A cache coherence scheme was chosen as another set of protocols to use because its traffic patterns are different from communication in an SoC system. Cache coherence involves a large number of broadcast and conditional messages to homogeneous functional units compared to the SoC-like communication that is typically more point-to-point and differs more based on the initiating functional unit.

The difference between the two cache coherence specifications is that while the conditional specification uses conditional messages to specify both the invalidations sent from the home directories to the caches during write misses and the data sent from the owner cache in read misses, the non-conditional specification does not use conditional messages at all. The focus of the specification without conditional messages would be to capture all messages with the same priority, while the conditional specification will likely trace signals into and from the home directories more often, as the messages they receive are weighted more highly.

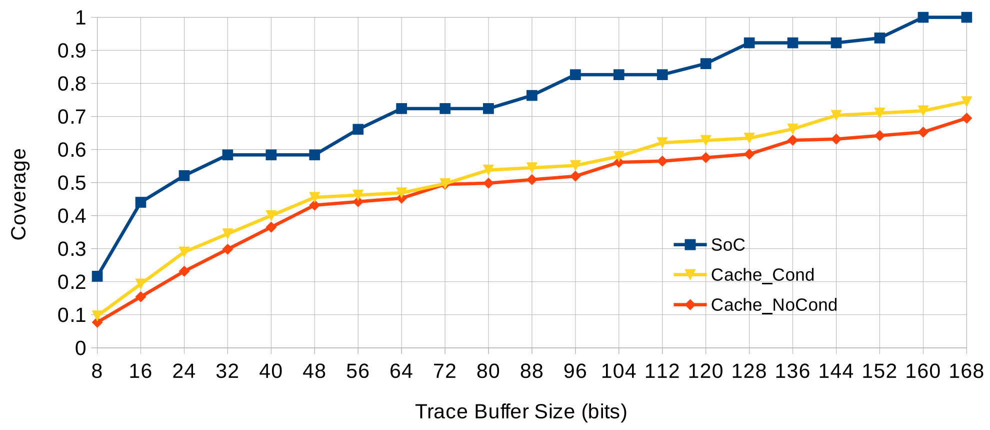

Figure 4.3: Frequency coverage for different protocol sets

Figures 4.3 and 4.4 show the results. We can see that for the SoC protocols, once the frequency coverage increases to a certain point, it increases further in steps that are about 32 bits apart. This tells us that once 32 bits of trace buffer are used, almost all low cost (8 bit command channels), high reward (many messages) channels have been selected and that adding more 8 bit

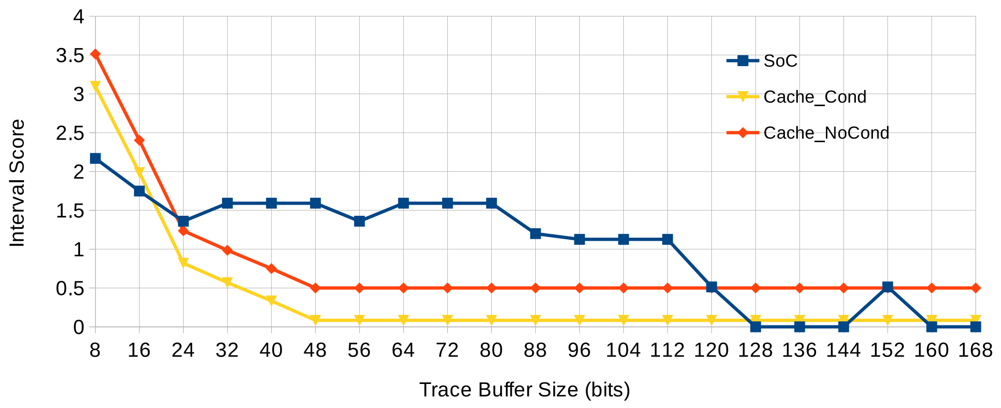

Figure 4.4: Interval score for different protocol sets

command channels is almost negligible to the overall frequency coverage. Larger increases in frequency coverage can only be gained by adding one of the 32 bit channels. The interval score trend is to decrease or stay constant unless there is a frequency coverage increase. Notice that all the increases in interval score correspond with an increase in frequency coverage. This shows that our final selection method is sacrificing some interval score for an increase in frequency coverage, just as is expected.

For the coherence protocols, we notice that there is a similar pattern in the frequency as the trace buffer size increases, but it is less pronounced. The coherence protocols have more broadcast messages, so as even the lesser used channels are added, they will show up in many of the executions of the protocol and increase the frequency coverage by some amount. The interval score in both coherence protocols also bottoms out at a value and never changes after that value. This is also due to the broadcast nature of the protocols. Some of the middle messages in the protocol are low value broadcast messages and, because they are towards the middle of the protocol, the interval penalty is not high. Therefore, the algorithm decides to omit these intervals entirely and adds messages from other intervals.

The graphs above could be valuable to a designer if deciding on a trace buffer size. As we can observe, there is no need to increase the buffer size past 160 bits for the SoC protocols for this view. Also, the difference between a buffer size of 128 and 136 is nonexistent for this view. This, combined with the results from other views, can guide a selection of trace buffer size.

One important result to report is the different solutions given for the conditional and non-conditional cache coherence protocols. For the conditional set, the selection, for a trace buffer size of 176 bits, is the following. Note that there were multiple solutions, but the only difference was the outgoing ADDR channel selection. For other solutions, a different home directory was chosen.

```
HOME3,cmd,0
HOME3,addr,0
HOME2,cmd,0
HOME1,cmd,0
0,addr,HOME3
0,addr,HOME2
0,addr,HOME1
0,cmd,HOME3
0,cmd,HOME2
0,cmd,HOME1
```

And the result for the non-conditional set for the same trace buffer size is as follows. Note that there were multiple solutions, but all were variations on this selection. For example, instead of selecting the channel (0,cmd,P1), another solution would select another processor instead of P1.

```
HOME3,cmd,0
HOME3,addr,0
HOME2,cmd,0
HOME2,addr,0
0,cmd,P1
HOME1,cmd,0
HOME1,addr,0
P6,cmd,0
P5,cmd,0
P4,cmd,0
P3,cmd,0
P2,cmd,0
P1,cmd,0
```

From these two selections, we see how the use, or lack, of conditional messages has guided this selection. For the conditional selection, only incoming and outgoing channels of the home directories were chosen. This is because all messages that involve a processor that is not the initiator are conditional and therefore have a lower weight. In the case where no conditional messages were used, home directory channels were clearly still targeted, but not as much as in the conditional case because the weight of all messages was equal. For other trace signal sizes, however, the selection was the same for both protocol sets. So the user can guide the selection for this set of protocols, but only to a certain extent, and the selection may vary little with varying details of the specification. The selection for the control view was the same in both cases.

4.3 Comparison to Gate-level Selection

A comparison is done between the popular gate-level, restoration-based trace signal selection method presented in [22] and the system-level selection presented here. In order to create a trace selection for the gate-level method, a USB controller block RTL design was used and a network interface was added that asserts high-level signals to implement the high-level behavior from the USB block in the SystemC TLM model. This was synthesized into a gate level netlist using Design Compiler [56] for use with the restoration-based method. To compare these two methods, we use a 72 bit trace buffer width and produce signal selection with both methods. We present the number of erroneous messages observed for certain bugs within the USB block. An erroneous message is a message within the correct specified execution of a protocol that is either not present, has incorrect data, or is sent to the wrong place. Observing an erroneous message means that either a message is on a traced channel but is not seen, incorrect data is directly observed, or a message is sent and is not observed to be received at a traced channel. The restoration-based selection only includes a single signal selection, so for the system-level trace, we will limit message count to only a single view with the highest number of observed erroneous messages. Results are shown in Table 4.2. Because the restoration based method does not include any notion of higher level functionality, it does not observe any of these messages.

We cannot detect the data bug in Table 4.2 because we lack the ability to see the data fields in our selected view. However, as will be shown later, if some other method is used to detect this type of bug, we can still effectively

Table 4.2: Error detection comparison between system-level and restoration-based methods. Note that neither method is able to detect data corruption in the USB block.

Bug Type	**Bug**	**Stimulus**	**# erroneous messages in our selection**	**# erroneous messages in restoration selection**
Data	USB corrupts data after downstream write	Downstream write-read pair to USB	0	0
Control	USB sends data to incorrect block on downstream read	Downstream write-read pair to USB	1	0
Halting	USB does not send PN_USB message after wake message from PWR block	One downstream read to USB block	1	0
Control	USB sends read request to incorrect block	One USB read	1	0

debug it using our selection.

For closer analysis of the inability of the restoration-based method to capture high-level functionality in the form of the message signals within the NI, Fig. 4.5 is presented. This figure shows the temporal depth between each message signal and the selected signals for the gate-level method for a 104 bit trace buffer width. For comparison, we present Fig. 4.6, which shows the heat map that corresponds to our direct selection of these system-level message signals. Temporal depth is the number of flip-flops between the two signals. For example, a signal that is within the combinational logic in the output stage of a flip-flop would have a temporal depth of 0 from that flip-flop. In general, a low temporal depth between two signals means that these two signals are highly related both spatially and functionally within a design. Note that in this implementation, the address channel was only 11 bits, so the top 21 bits of both address channels were removed from the graph.

The main observation about the two heat maps is that the gate-level selection method is unable to create any predictable relation to the high-level message signals. Dark blue (temporal depth of 0) portions do exist, but only

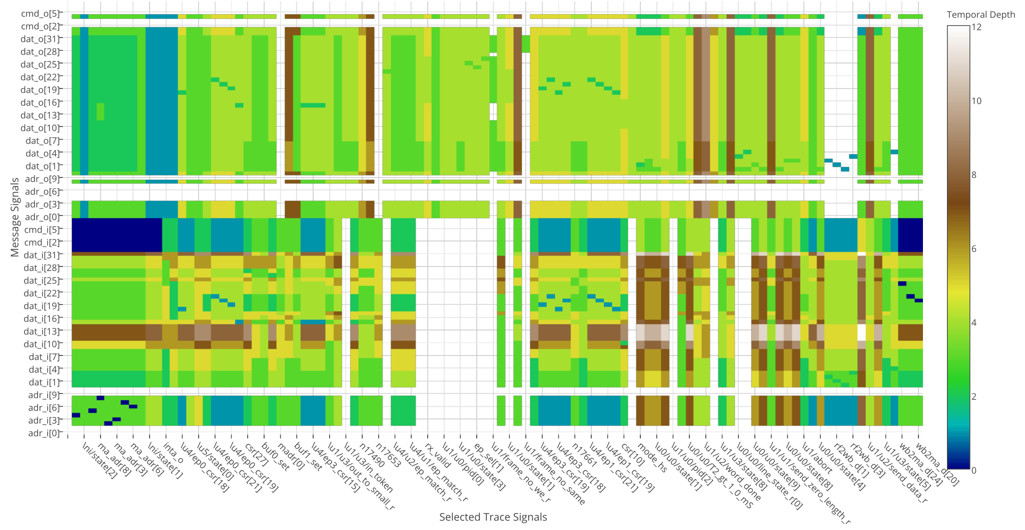

Figure 4.5: Temporal depth between gate-level based selection trace signals and system-level message signals

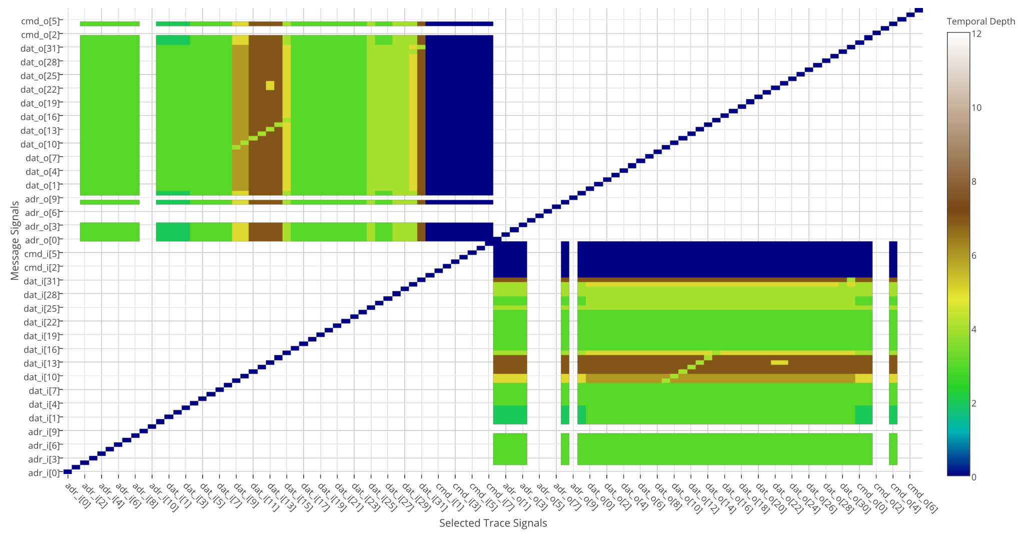

Figure 4.6: Temporal depth between our trace signal selection and system-level message signals

for a few select channels. Any depth beyond 0 would be unreliable in capturing the high-level message signal because of the large amount of logic between the two signals. Even a temporal depth of 0 indicates not that the high-level message signal can be captured in the selected signal, but rather that it is more likely to be captured. The direct selection predictably produces a more regular pattern between the selected signals and message signals and can be used to reliably capture this information.

4.4 Bug Case Studies and Localization

To understand how the traces selected aid in debug and the extent to which each bug can be localized, we now present a case study of different bugs that are debugged using signals selected with our trace signal selection method. Localization results are also presented. The input into the signal selection was the set of SoC protocols mentioned above. A trace width size of 72 bits was chosen. This is the number of bits needed to capture only one complete set of one incoming or outgoing channel (command, address, and data) and trace buffer widths in modern systems would be limited to around this relative size. Each incoming and outgoing channel was instrumented within the SystemC code with a function to look up when each incoming or outgoing message should be traced in the selected view, and if so, it was printed with the correct values to act as a trace.

4.4.1 Halting Bugs

We implement three different halting bugs. Each bug's root cause is a functional unit not responding in any manner to some stimulus message.

First, let us do a general case of the Audio not responding to any messages of any type. The input to get the trace was a series of downstream reads and writes to various functional units with initially no functional units powered on. The Audio block does not respond to any messages. The following global view trace was observed. Traces are in the format `<cmd, data, addr>`. Note that a value of `XX` in the trace denotes that while that field is being traced, this message did not have valid information on that field. For example, the third message shown below is a PN_Audio message, which does not contain any information on the address field. Also, a value of `[]` denotes that this field is not being traced.

```
Out CPU = <MEM_W, [], []>
In SA = <MEM_W, [], []>
Out SA = <PN_Audio, [], XX>
Out CPU = <MEM_R, [], []>
```

```
In SA = <MEM_R, [], []>
Out SA = <REQR, [], XX>
In CPU = <REQR, [], []>
Out CPU = <REPR, [], []>
In SA = <REPR, [], []>
Out SA = <PN_Audio, [], XX>
Out CPU = <MEM_W, [], []>
```

This trace shows that for both the write and read, no response was received after the SA sent the power-on message. This can be a problem with either the PWR block not forwarding the power-on message or the Audio block not responding once it is forwarded. We know that the Audio block view includes the outgoing command channel of the PWR block, so we use this view to attempt to confirm the message forwarding to the Audio block.

```
In SA = <MEM_W, [], []>
Out SA = <PN_Audio, [], []>
Out PWR = <PN_Audio, [], []>
In Audio = <PN_Audio, [], []>
In SA = <MEM_R, [], []>
Out SA = <REQR, [], []>
In SA = <REPR, [], []>
Out SA = <PN_Audio, [], []>
Out PWR = <PN_Audio, [], []>
In Audio = <PN_Audio, [], []>
```

Here we see that the Audio block is to fault for this error. We can localize this bug to the entire Audio block because we cannot confirm any correct behavior, so we must examine the entire design. This took 2 runs.

Now, let us specialize this error such that the Audio block only fails to respond to downstream writes. The following is the global trace of this error.

```
Out CPU = <MEM_W, [], []>
In SA = <MEM_W, [], []>
Out SA = <PN_Audio, [], XX>
Out PWR = <PN_Audio, [], []>
Out PWR = <PN_Audio, [], []>
In SA = <PN_Audio, [], []>
```

```
Out SA = <MEM_W, [], 0x200>
Out CPU = <MEM_R, [], []>
In SA = <MEM_R, [], []>
Out SA = <REQR, [], XX>
In CPU = <REQR, [], []>
Out CPU = <REPR, [], []>
In SA = <REPR, [], []>
Out SA = <MEM_R, [], 0x200>
In SA = <XX, [], []>
Out SA = <XX, [], 0x200>
In CPU = <XX, [], []>
Out SA = <RESR, [], XX>
In CPU = <RESR, [], []>
CPU read 0xaa at addr 0x200
```

Here, we see that the PWR block sends its response that the Audio block is powered on, so the SA block sends a MEM_W message. This write is programmed to write 0x55 to address 0x200, but we cannot confirm that this write is indeed the same one that is writing 0x55 or that it is reaching the Audio block. We do another run with the control view to attempt to see where this message is being sent to.

```
In SA = <MEM_W, [], []>
SA sent = <PN_Audio, XX, []> to PWR
In PWR = <PN_Audio, [], []>
In PWR = <PN_Audio, [], []>
In SA = <PN_Audio, [], []>
SA sent = <MEM_W, 0x55, []> to Audio
In SA = <MEM_R, [], []>
SA sent = <REQR, XX, []> to CPU
In SA = <REPR, [], []>
SA sent = <MEM_R, XX, []> to Audio
In SA = <XX, [], []>
SA sent = <XX, 0xaa, []> to CPU
SA sent = <RESR, XX, []> to CPU
CPU read 0xaa at addr 0x200
```

Here, we see that the message is sent to the Audio block with data of 0x55. This means the root cause is in the Audio block, either with how the write is being handled or some data corruption once there is a read from the same location. In either case, this problem has been localized to the handling of the MEM_W and/or MEM_R in the Audio block. This took 2 runs.

Now, let us explore another halting-type bug. This bug is on the conditional hit data that may be sent during an upstream read/write. It fails to be sent and would cause an upstream read/write to never complete. The input for this is a series of upstream writes from the GFX block to the USB. The bug can be confirmed by checking the values written. In this case, certain addresses, one of them being 0x30A, are not reading what they should have been written. We start out with a global view trace.

```
In SA = <MEM_W, [], []>
Out SA = <MEM_W, [], 0x309>
In CPU = <MEM_W, [], []>
Out CPU = <MISS_W, [], []>
In SA = <MISS_W, [], []>
Out SA = <MEM_W, [], 0x309>
In SA = <MEM_W, [], []>
Out SA = <MEM_W, [], 0x30a>
In CPU = <MEM_W, [], []>
Out CPU = <HIT_W, [], []>
In SA = <HIT_W, [], []>
In SA = <MEM_W, [], []>
Out SA = <MEM_W, [], 0x30b>
In CPU = <MEM_W, [], []>
Out CPU = <MISS_W, [], []>
In SA = <MISS_W, [], []>
Out SA = <MEM_W, [], 0x30b>
```

This shows that the write to 0x30A was a hit, but no data is ever sent from the CPU to the SA block so that it can be forwarded along. Even though the data field is not traced, the incoming command channel of the SA block is, so we would see some message on that incoming channel. In this case, we know the issue is with how the CPU is handling MEM_W commands with a hit. This localization only took one run.

4.4.2 Data Bugs

We now introduce 4 different data bugs. These bugs introduce data corruption either as a data field is forwarded or while it is being stored in memory at a block. In all cases, the bug is already known and does not need to be detected, only localized. This is different from the study done above. Our first bug is data corruption as a downstream read is being completed. The SA block will somehow corrupt the data before it is forwarded, causing the wrong value to be read at the CPU. The following is the trace from the portion that initiates a read from the Audio block after a write of 0x55 has completed just before.

```
Out CPU = <MEM_R, [], []>
In SA = <MEM_R, [], []>
Out SA = <REQR, [], XX>
In CPU = <REQR, [], []>
Out CPU = <REPR, [], []>
In SA = <REPR, [], []>
Out SA = <MEM_R, [], 0x200>
In SA = <XX, [], []>
Out SA = <XX, [], XX>
In CPU = <XX, [], []>
Out SA = <RESR, [], XX>
In CPU = <RESR, [], []>
CPU read 0xac at addr 0x200
```

The wrong value is read, but from the trace, there seems to be no error in either the read or write (the write is not shown for brevity). At this point, we are not even sure if the read is being sent to the correct block, so let us try again, but with the Audio block view.

```
In SA = <MEM_R, [], []>
Out SA = <REQR, [], []>
In SA = <REPR, [], []>
Out SA = <MEM_R, [], []>
In Audio = <MEM_R, [], []>
Out Audio = <XX, 0x55, []>
In SA = <XX, [], []>
Out SA = <XX, [], []>
Out SA = <RESR, [], []>
CPU read 0xac at addr 0x200
```

Again, no error is shown from this trace, but we do know that the Audio block is receiving the request. We do see that the Audio block sends the expected correct data, 0x55, to the SA block however. Therefore, the bug must be in either the CPU or SA block. We know that the control view captures all outgoing data field message from the SA, so we use this view to try to capture the data transfer between CPU and SA.

```
In SA = <MEM_R, [], []>
SA sent = <REQR, XX, []> to CPU
In SA = <REPR, [], []>
SA sent = <MEM_R, XX, []> to Audio
In SA = <XX, [], []>
SA sent = <XX, 0xac, []> to CPU
SA sent = <RESR, 0x0, []> to CPU
CPU read 0xac at addr 0x200
```

In this view we observe that the SA block is sending the incorrect data value of 0xAC to the CPU. From previous traces, we know that the Audio block sent the correct data, so we can localize this bug to the handling of a downstream MEM_R message in the SA block. This debug took 3 runs to localize. If the corruption occurred in the memory of Audio block instead of in the forwarding at the SA block, this bug would also be captured in the second trace. Note that while our control view includes the outgoing data channel of the SA block, the goal of the control view is to observe possible control bugs. The fact that our control view has included the data field is helpful for this data bug, but we should note that the control view likely will not help in finding data bugs. Because many messages in SoC protocols are concerned with routing commands to the correct blocks and less focused on sending the actual data, data is usually sent less often and therefore does not show up in many of our views. To combat this, a designer could either create a set of protocols of only the data messages and then get a selection on these, similar to how block-specific views are created, or manually add a data view. In the case above, we would not be able to localize this bug to the SA block without this control view channel being observed because we are not able to observe the data going out of the SA block or into the CPU.

Now, let us examine the case where the data is corrupted in the memory of the Audio block between a write and read. The input is the same as before. First, a section of the global view is shown.

```
Out CPU = <MEM_R, [], []>
In SSA = <MEM_R, [], []>
Out SSA = <REQR, [], XX>
In CPU = <REQR, [], []>
Out CPU = <REPR, [], []>
In SSA = <REPR, [], []>
Out SSA = <MEM_R, [], 0x200>
In SSA = <XX, [], []>
Out SSA = <XX, [], 0x200>
In CPU = <XX, [], []>
Out SSA = <RESR, [], XX>
In CPU = <RESR, [], []>
CPU read 0xac at addr 0x200
```

This looks the same as the previous case, so let us use the Audio block view again to get a closer look.

```
In SSA = <MEM_R, [], []>
Out SSA = <REQR, [], []>
In SSA = <REPR, [], []>
Out SSA = <MEM_R, [], []>
In Audio = <MEM_R, [], []>
Out Audio = <XX, 0xac, []>
In SSA = <XX, [], []>
Out SSA = <XX, [], []>
Out SSA = <RESR, [], []>
CPU read 0xac at addr 0x200
```

Here we observe that the data from the Audio block is incorrect this time. This bug can be in the handling of a write or read in the Audio block. It took us 2 runs to localize the issue.

Corruption can also occur in fields other than the data field. In this next bug, we introduce a bug in the command field when an upstream write occurs. Instead of sending a write to a block, the SA sends a read. The following trace is from a series of writes from the Audio block to the USB block. First, a trace is done with the global view.

```
In SA = <MEM_W, [], []>
Out SA = <MEM_W, [], 0x301>
In CPU = <MEM_W, [], []>
Out CPU = <MISS_W, [], []>
In SA = <MISS_W, [], []>
Out SA = <PN_USB, [], XX>
Out PWR = <PN_USB, [], []>
Out PWR = <PN_USB, [], []>
In SA = <PN_USB, [], []>
Out SA = <MEM_W, [], 0x301>
In SA = <MEM_W, [], []>
Out SA = <MEM_W, [], 0x302>
In CPU = <MEM_W, [], []>
Out CPU = <MISS_W, [], []>
In SA = <MISS_W, [], []>
Out SA = <MEM_R, [], 0x302>
In SA = <XX, [], []>
In SA = <MEM_W, [], []>
Out SA = <MEM_W, [], 0x303>
```

The first write powers on the USB block and we see no errors in the first write. However, beginning in the second write we observe that the SA block receives a MISS_W message from the CPU, but instead of sending a MEM_W to the block being written to, it sends a MEM_R message. The block that this message was sent to (USB) responds with the read data to the SA, but this data is ignored by the SA and the next write begins. We see that this bug can also be classified as a halting bug because it halts the upstream memory write protocol once the MEM_R message is sent incorrectly. However, the root cause is a data bug on the command field. This can be observed in this first run, and localized to the handling of a MISS_W message within the SA block.

Let us implement another incorrect command field bug. This time, the PWR block sends an incorrect power-on message when powering on the GFX block. The input is a series of downstream writes and reads to different blocks with all blocks initially off. The bug is detected by the lack of read response to the CPU after a read instruction was issued. A section of the global view trace is shown below.

```
Out CPU = <MEM_W, [], []>
In SSA = <MEM_W, [], []>
Out SSA = <PN_GFX, [], XX>
Out PWR = <PN_Audio, [], []>
Out CPU = <MEM_R, [], []>
In SSA = <MEM_R, [], []>
Out SSA = <REQR, [], XX>
In CPU = <REQR, [], []>
Out CPU = <REPR, [], []>
In SSA = <REPR, [], []>
Out SSA = <PN_GFX, [], XX>
Out PWR = <PN_Audio, [], []>
```

Here, we see that the SSA block attempts to send a PN_GFX message to the PWR block to forward, but the PWR block sends a PN_Audio message, ending the transaction.

4.4.3 Control Bugs

Now to test our trace selection's ability to observe control bugs, we introduce multiple control bugs. The first bug will be the SA block sending an upstream read to the wrong location. This bug could be found by end result checking of reads following writes. Our trace is of the write and the read to address location 0x300, the USB block.

```
In SSA = <MEM_W, [], []>
Out SSA = <MEM_W, [], 0x300>
In CPU = <MEM_W, [], []>
Out CPU = <MISS_W, [], []>
In SSA = <MISS_W, [], []>
Out SSA = <PN_USB, [], XX>
Out PWR = <PN_USB, [], []>
Out PWR = <PN_USB, [], []>
In SSA = <PN_USB, [], []>
Out SSA = <MEM_W, [], 0x300>
In SSA = <MEM_R, [], []>
Out SSA = <MEM_R, [], 0x300>
In CPU = <MEM_R, [], []>
```

```
Out CPU = <MISS_R, [], []>
In SSA = <MISS_R, [], []>
Out SSA = <MEM_R, [], 0x300>
Out SSA = <PN_UART, [], XX>
```

From these traces, we do not see any issues with the write and read from the Audio block to the USB block, however there is this PN_UART message once our memory read is sent out, which does not seem correct because once our read finishes, an upstream write should begin. We will concern ourselves with this after finding the cause of our incorrect data. We know that this read returns the wrong value from an end result check, so we use the USB block view to get a closer look.

```
In SSA = <MEM_W, [], []>
Out SSA = <MEM_W, [], []>
In SSA = <MISS_W, [], []>
Out SSA = <PN_USB, [], []>
In PWR = <PN_USB, 0x3, []>
Out PWR = <PN_USB, [], []>
In USB = <PN_USB, 0x3, []>
Out USB = <PN_USB, [], []>
In PWR = <PN_USB, 0x3, []>
Out PWR = <PN_USB, [], []>
In SSA = <PN_USB, [], []>
Out SSA = <MEM_W, [], []>
In USB = <MEM_W, 0x3, []>
In SSA = <MEM_R, [], []>
Out SSA = <MEM_R, [], []>
In SSA = <MISS_R, [], []>
Out SSA = <MEM_R, [], []>
Out SSA = <PN_UART, [], []>
```

Here we observe that the SA block sends a MEM_R message, but it is not received by the USB block. Also, the PN_UART message is still present. We now use the control view to see where that MEM_R message was sent.

```
In SSA = <MEM_W, [], []>
SSA sent = <MEM_W, 0x3, []> to CPU
In SSA = <MISS_W, [], []>
```

```
SSA sent = <PN_USB, 0x3, []> to PWR
In PWR = <PN_USB, [], []>
In PWR = <PN_USB, [], []>
In SSA = <PN_USB, [], []>
SSA sent = <MEM_W, 0x3, []> to USB
In SSA = <MEM_R, [], []>
SSA sent = <MEM_R, XX, []> to CPU
In SSA = <MISS_R, [], []>
SSA sent = <MEM_R, XX, []> to GFX
SSA sent = <PN_UART, XX, []> to UART
```

Here we see that the MEM_R was sent to the GFX unit, which is very likely the cause of the incorrect data. We have localized this bug to the handling of either MEM_R or MISS_R messages within the SA block. Also, we observe that the PN_UART message is sent to the UART block, which is an incorrect behavior for the SA block. All PN messages should be forwarded to the PWR block. Also, there is no reason the SA block should have sent this message. This is also localized to handling of MEM_R and/or MISS_R messages within the SA block. Clearly, there is some bug within the SA block causing this incorrect behavior. After examining the code for the SA block, we found the cause of this behavior, which was an incorrect address range check on the memory read. If a memory read for the address range that corresponds to the USB block was sent to the SSA, it would incorrectly send a PN_UART message along with the correct PN_USB if the respective blocks were currently powered off. In addition to the fact that this message should not be sent, it was also sent to the wrong place. This bug was not intended to be tested, but it demonstrates that the trace signal selection is applicable for finding bugs that may realistically occur while developing a system.

Table 4.3 shows the amount of the design that must be examined to find the root cause of each bug from the above case study. The localization is reported as the percentage of SystemC lines that must be examined to find the root cause. This table shows that while many bugs can be localized to less than 5% of the design, the bugs within the SA block usually cannot be localized to this extent because of the large amount of logic needed to handle each message type. In the SA block, the handling of a MEM_R message uses 120 lines of SystemC code. The handling of same message by the Audio block, for example, is around 14 lines. These show the differences within the behavior of the blocks. Therefore, even though the bug was localized to handling of a single message type within one block, the amount of the overall design that must be examined may be different. In either case, all bugs implemented here were localized to within 12.39% of the total design in the worst case, and, on average, were localized to less than 5% of the design.

Table 4.3: Overview of bug case studies

Bug	**% of Design**	**Run**	**Localization Detail**
No response from Audio block	4.597	1st	PWR or Audio block
	4.264	2nd	Audio Block
No response for downstream writes from Audio block	12.66	1st	SA or Audio Block
	1.599	2nd	Audio Block
Hit data not sent from CPU	1.200	1st	CPU block
SA corrupts downstream read data	100	1st	No erroneous message seen
	19.25	2nd	CPU or SA block
	7.795	3rd	SA block
Audio corrupts downstream read data	100	1st	No erroneous message seen
	1.599	2nd	Audio block
SA sends read instead of write for upstream write	6.396	1st	SA block
PWR sends wrong command	0.3331	1st	PWR block
SA forwards read request to wrong block	100	1st	No erroneous message
	12.39	2nd	SA block
	9.793	3rd	SA block
Erroneous PN_UART message sent	16.59	1st	SA block
	16.59	2nd	SA block
	12.39	3rd	SA block

4.5 Conclusion and Future Work

4.5.1 Conclusion

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

1. Provides accurate results with the given pruning used (within 5% of the optimal frequency coverage solution)
2. Provides the engineer specifying the protocols some degree of guidance in the final solution
3. Provides the first automated selection of trace signals that capture system-level behavior
4. Is effective at localizing bugs using a realistic trace buffer width relative to the overall design size.

Results in Section 4.2.1 indicate that a pruning to 5% of the optimal frequency coverage value will likely include the best solution once both the frequency and interval score are considered. This is because our experiments indicate an inverse relationship between frequency coverage and interval score. The accuracy of the results could be improved by increasing the amount of solutions pruned to, but it is detrimental to run time.

Our experiments in Section 4.2.2 show that our method, while automated, still allows some guidance from the user by the use of the conditionality values within the protocol specification and highlight how the frequency coverage and interval score values can be affected. Even though guidance is possible, our results indicate that the differences are small, which also indicates that our method is not overly reliant on the input, such that any reasonable interpretation of the protocol specification will produce similar results.

In Section 4.3, a comparison to a popular gate-level trace selection was performed. The results indicate what was intuitively speculated: gate-level methods do not have any information available about system-level functionality of the circuit they are analyzing and, therefore, do capture system-level behavior.

Section 4.4 proves that our trace signal selection is able to localize all implemented bugs to within 12.4% of the design in the worst case. Our results prove that this automated trace signal selection is able to provide a quick selection of trace signals that, as a result of the usage of a linear programming formulation, provide enough quality to localize our set of bugs to a small percentage of the design. Once a bug has been localized to this extent, either other post-silicon DfD structures or targeted pre-silicon tests can be used to find the root cause of the bug.

In addition to the already mentioned achievements of our method, it also provide other advantages. First, our method is virtually non-intrusive to the design, as we simply need to route and trace signals. Other debug structures and methods require augmentations to the working design to provide debugging services, while our method does not [34, 35, 14]. Also, our selection can occur very early in the IC design flow because we only require a protocol specification, allowing for plenty of design time to route and add the needed structures. Gate-level selection methods require a netlist, which means that they require a full RTL implementation that occurs much later in the design process. In addition to choosing a set of trace signals for a given trace buffer width, our method can also be used to find the minimum trace buffer width needed to observe all messages. In some cases, this width may be much less than the width needed to trace all channels within a design and may be an attractive solution to post-silicon debugging. Finally, our tool can also be helpful to designers as a way to understand message patterns in the specified protocols. This may lead a designer to change their design, which can be done, because all of this is done early in the design process.

4.5.2 Future Work

Future work on our specific solution to the problem of system-level trace signal selection could focus on the implementation in systems with split transactions, as mentioned in Section 4.1. For our purposes, we equate transactions to protocols. In a real system, a block may be receiving messages that are a part of many different protocols concurrently. Each block is able to handle this because the header of each message it receives would include an ID that is unique to that transaction/protocol. Therefore, raw, untouched traces of the incoming and outgoing channels would include interleaved messages from many protocol/transactions, making debug more difficult. One way to alleviate this is to simply record the IDs and then reorder each trace before it is presented to the debugger, but this uses valuable trace buffer space. Other methods may include real-time reordering of traces or compression techniques to reduce the overhead of storing IDs.

In addition to implementation-related future work, automation in the localization process may be possible. Once traces are obtained and offloaded, they may be able to be compared to the specified protocols to automatically detect erroneous behavior. It may be possible to both detect and localize bugs using this method. In addition to offloading traces prior to comparison, a real-time comparison module that has the ability to detect differences between the execution traces and correct behavior during execution is a possibility. An initial thought about a real-time comparison module is that the overall area overhead may be too large, so this would have to be implemented in an area-efficient manner.

APPENDIX A

PROTOCOLS

Shown in this appendix are the protocols used in our experiments. These are presented in both text and the flow diagrams provided in Figs. A.1 - A.7.

A.1 SoC Protocols

```
//Upstream Memory Write (Dirty snoop)
Initiators: GFX, Audio
Targets: PWR, Audio, GFX, USB, UART
<I, <cmd><addr><data>, SA, 1> //initial request
<SA, <cmd><addr>, CPU, 2> //forward to cpu
<CPU, <cmd>, SA, 3> //hit response
<CPU, <data>, SA, 4, .5>  //conditional data on hit
<SA, <cmd><addr><data>, T, 5> //commit write to target
end

//Upstream Memory write; target and CPU powered down
Initiators: GFX, Audio
Targets: PWR, Audio, GFX, USB, UART
<I, <cmd><addr><data>, SA, 1> //initial request
<SA, <cmd>, PWR, 2> //power up if CPU off
<PWR, <cmd>, CPU, 3> //PWR up CPU
<CPU, <cmd>, PWR, 4> //CPU is on
<PWR, <cmd>, SA, 5> //SA knows CPU is on
<SA, <cmd><addr>, CPU, 6> //forward to cpu
<CPU, <cmd>, SA, 7> //hit response
<CPU, <data>, SA, 8, .5>  //conditional data on hit
<SA, <cmd>, PWR, 9> //power up if Target off
<PWR, <cmd>, T, 10> //PWR up Target
<T, <cmd>, PWR, 11> //Target is on
```

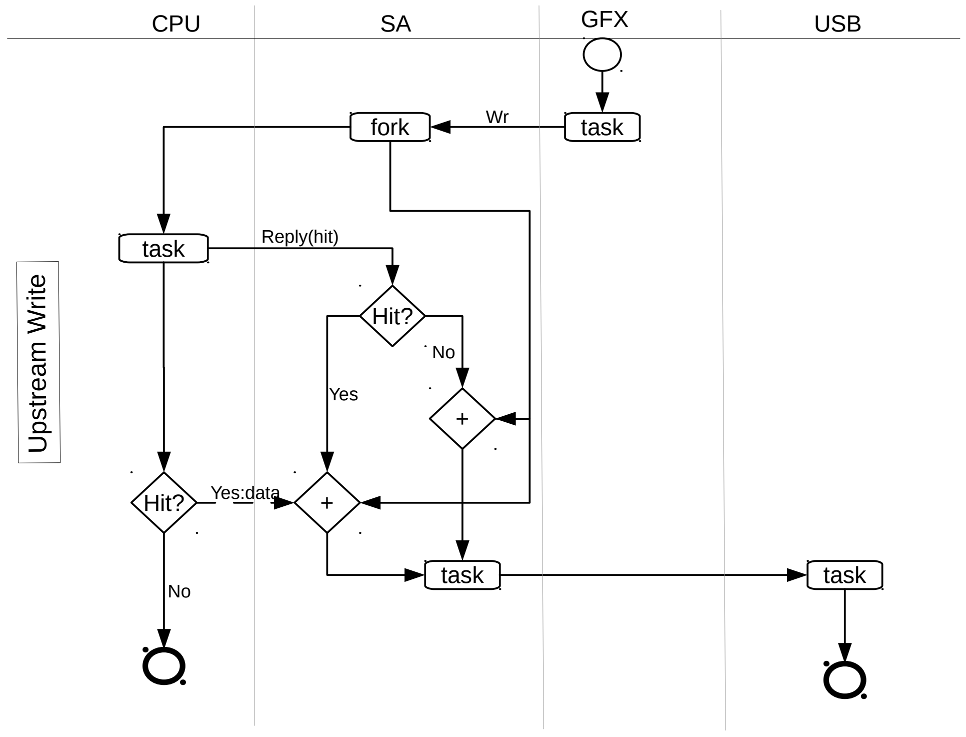

Figure A.1: Upstream write

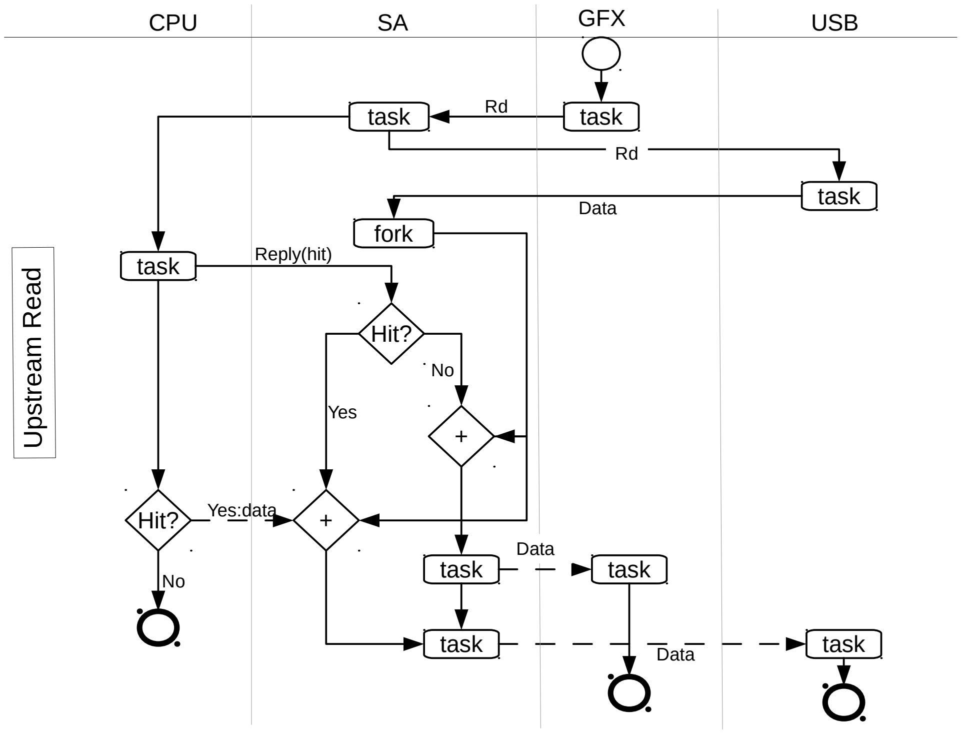

Figure A.2: Upstream read

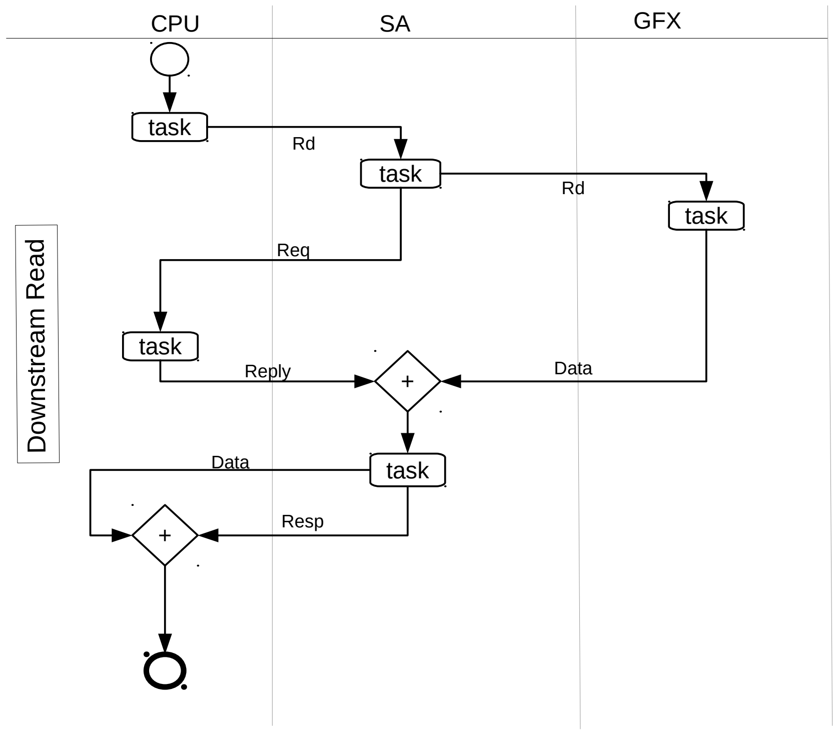

Figure A.3: Downstream read

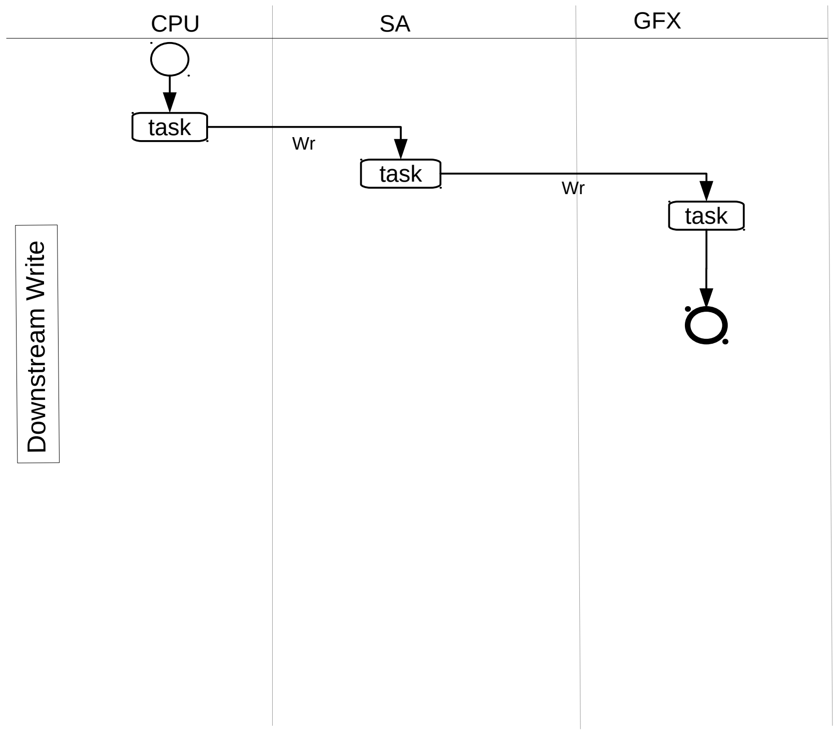

Figure A.4: Downstream write

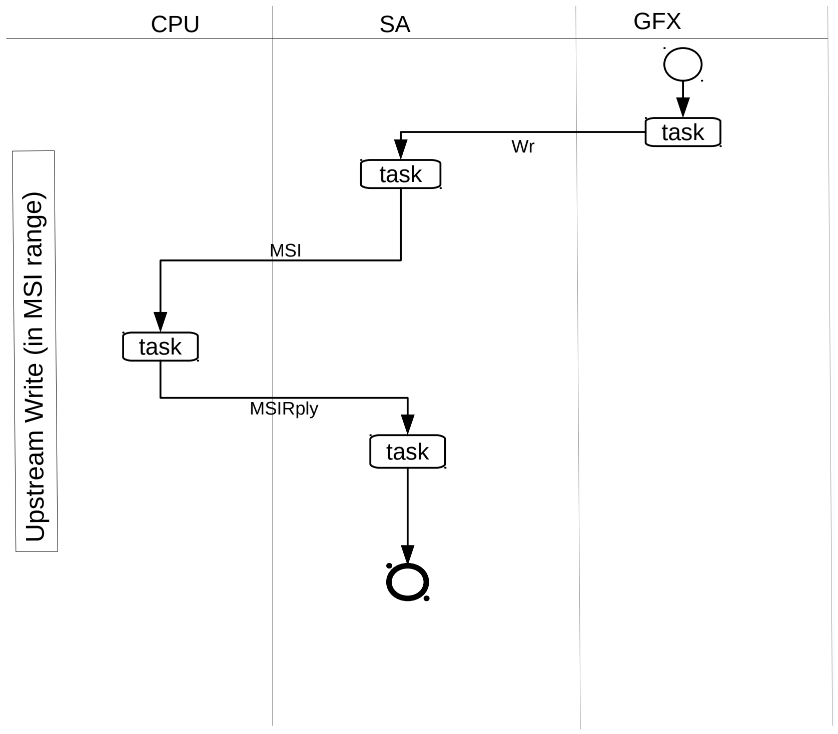

Figure A.5: Upstream write (in MSI range)

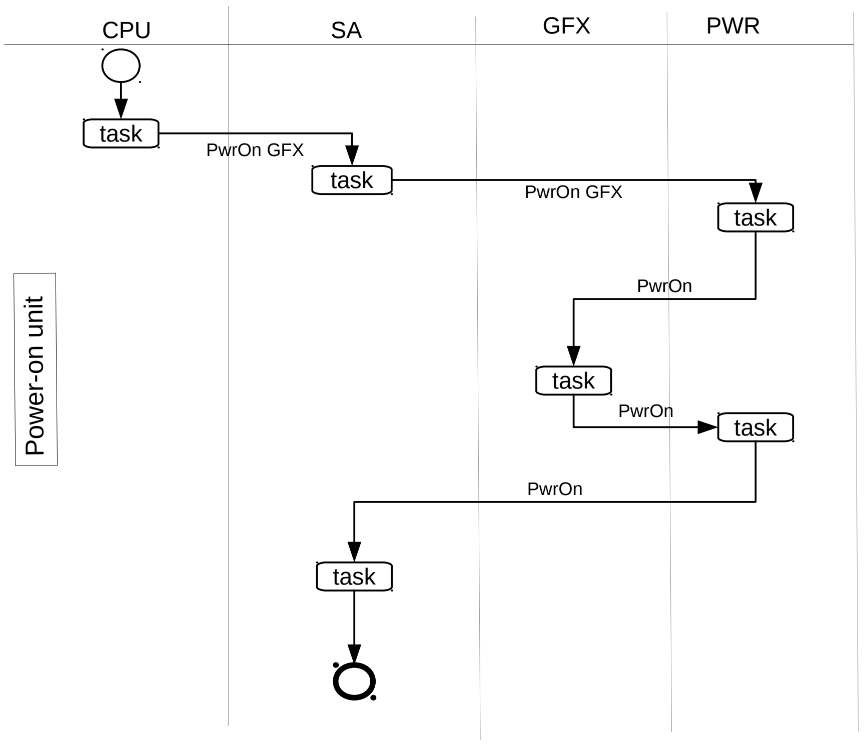

Figure A.6: Power-on unit

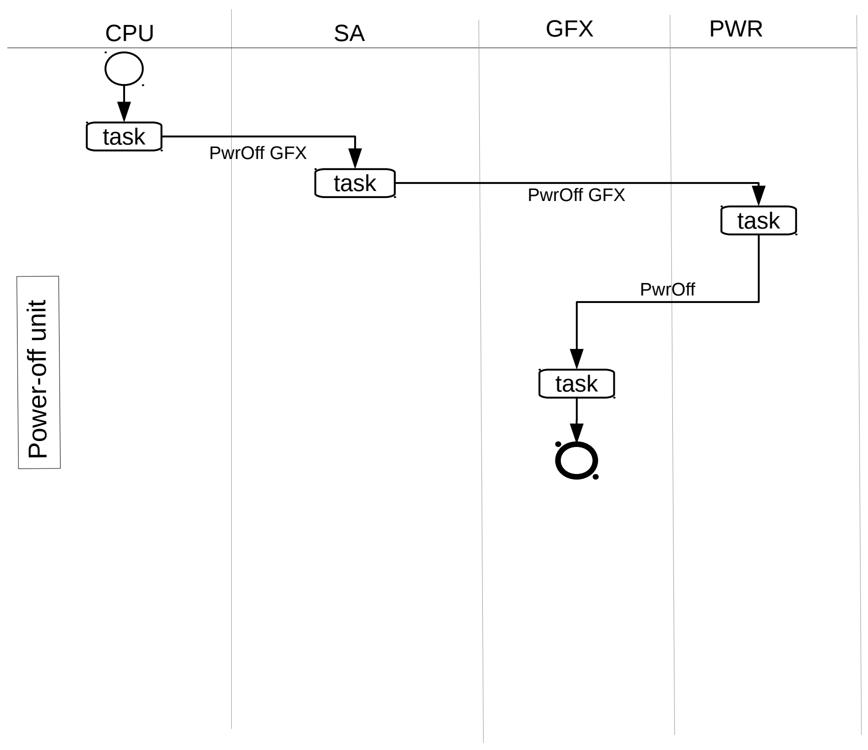

Figure A.7: Power-off unit

```
<PWR, <cmd>, SA, 12> //SA knows Target is on
<SA, <cmd><addr><data>, T, 13> //commit write to target
end

//Upstream Memory write; only target powered down
Initiators: GFX, Audio
Targets: PWR, Audio, GFX, USB, UART
<I, <cmd><addr><data>, SA, 1> //initial request
<SA, <cmd><addr>, CPU, 2> //forward to cpu
<CPU, <cmd>, SA, 3> //hit response
<CPU, <data>, SA, 4, .5>  //conditional data on hit
<SA, <cmd>, PWR, 5> //power up if Target off
<PWR, <cmd>, T, 6> //PWR up Target
<T, <cmd>, PWR, 7> //Target is on
<PWR, <cmd>, SA, 8> //SA knows Target is on
<SA, <cmd><addr><data>, T, 9> //commit write to target
end

//Upstream Memory write; only CPU powered down
Initiators: GFX, Audio
Targets: PWR, Audio, GFX, USB, UART
<I, <cmd><addr>, SA, 1> //initial request
<SA, <cmd>, PWR, 2> //power up if CPU off
<PWR, <cmd>, CPU, 3> //PWR up CPU
<CPU, <cmd>, PWR, 4> //CPU is on
<PWR, <cmd>, SA, 5> //SA knows CPU is on
<SA, <cmd><addr>, CPU, 6> //forward to cpu
<I, <data>, SA, 6> //initial data
<CPU, <cmd>, SA, 7> //hit response
<CPU, <data>, SA, 8, .5>  //conditional data on hit
<SA, <cmd><addr><data>, T, 9> //commit write to target
end

//Upstream Memory Read (Dirty snoop)
Initiators: GFX, PWR, Audio, USB, UART
Targets: PWR, Audio, GFX, USB, UART
<I, <cmd><addr>, SA, 1> //initial message
<SA, <cmd><addr>, T, 2> //forward to target
```

```
<SA, <cmd><addr>, CPU, 2> //forward to CPU
<T, <data>, SA, 3> //D-Unit response to SA
<CPU, <cmd>, SA, 3> //CPU hit response to SA
<CPU, <data>, SA, 4, .5>
<SA, <data>, I, 5>
<SA, <data>, T, 6, .5>
end

//Upstream Memory Read (Dirty snoop); Both CPU and target powered off
Initiators: GFX, PWR, Audio, USB, UART
Targets: PWR, Audio, GFX, USB, UART
<I, <cmd><addr>, SA, 1> //initial message
<SA, <cmd>, PWR, 2> //power up if CPU off
<PWR, <cmd>, CPU, 3> //PWR up CPU
<CPU, <cmd>, PWR, 4> //CPU is on
<PWR, <cmd>, SA, 5> //SA knows CPU is on
<SA, <cmd>, PWR, 6> //power up if Target off
<PWR, <cmd>, T, 7> //PWR up Target
<T, <cmd>, PWR, 8> //Target is on
<PWR, <cmd>, SA, 9> //SA knows Target is on
<SA, <cmd><addr>, T, 10> //forward to target
<SA, <cmd><addr>, CPU, 10> //forward to CPU
<T, <data>, SA, 11> //Target response to SA
<CPU, <cmd>, SA, 11> //CPU hit response to SA
<CPU, <data>, SA, 12, .5>
<SA, <data>, I, 13>
<SA, <data>, T, 14, .5>
end

//Upstream Memory Read (Dirty snoop);target powered off
Initiators: GFX, PWR, Audio, USB, UART
Targets: PWR, Audio, GFX, USB, UART
<I, <cmd><addr>, SA, 1> //initial message
<SA, <cmd><addr>, CPU, 2> //forward to CPU
<SA, <cmd>, PWR, 2> //power up if Target off
<CPU, <cmd>, SA, 3> //CPU hit response to SA
<PWR, <cmd>, T, 3> //PWR up Target
<CPU, <data>, SA, 4, .5> //conditional hit data from CPU
```

```
<T, <cmd>, PWR, 4> //Target is on
<PWR, <cmd>, SA, 5> //SA knows Target is on
<SA, <cmd><addr>, T, 6> //forward to target
<T, <data>, SA, 7> //Target response to SA
<SA, <data>, I, 8>
<SA, <data>, T, 9, .5>
end

//Upstream Memory Read (Dirty snoop); CPU powered off
Initiators: GFX, PWR, Audio, USB, UART
Targets: PWR, Audio, GFX, USB, UART
<I, <cmd><addr>, SA, 1> //initial message
<SA, <cmd><addr>, T, 2> //forward to target
<SA, <cmd>, PWR, 2> //power up if CPU off
<T, <data>, SA, 3> //Target response to SA
<PWR, <cmd>, CPU, 3> //PWR up CPU
<CPU, <cmd>, PWR, 4> //CPU is on
<PWR, <cmd>, SA, 5> //SA knows CPU is on
<SA, <cmd><addr>, CPU, 6> //forward to CPU
<CPU, <cmd>, SA, 7> //CPU hit response to SA
<CPU, <data>, SA, 8, .5>
<SA, <data>, I, 9>
<SA, <data>, T, 10, .5>
end

//MIMO Downstream Memory Read:
Targets: GFX, Audio, USB, PWR, UART
<CPU, <cmd><addr>, SA, 1>
<SA, <cmd><addr>, T, 2>
<SA, <cmd>, CPU, 2>
<CPU, <cmd>, SA, 3>
<T, <data>, SA, 3>
<SA, <data>, CPU, 4>
<SA, <cmd>, CPU, 4>
end

//MIMO Downstream Memory Read: target powered down
Targets: GFX, Audio, USB, PWR, UART
```

```
<CPU, <cmd><addr>, SA, 1>
<SA, <cmd>, PWR, 2> //power up if Target off
<PWR, <cmd>, T, 3> //PWR up Target
<T, <cmd>, PWR, 4> //Target is on
<SA, <cmd><addr>, T, 5>
<SA, <cmd>, CPU, 5>
<CPU, <cmd>, SA, 6>
<T, <data>, SA, 6>
<SA, <data>, CPU, 7>
<SA, <cmd>, CPU, 7>
end

//Upstream Interrupt:
Initiators: GFX, Audio
<I, <cmd><addr>, SA, 1>
<SA, <cmd><addr>, CPU, 2>
<CPU, <cmd>, SA, 3>
end

//Upstream Interrupt, CPU is powered-down:
Initiators: GFX, Audio, USB, UART
<I, <cmd><addr>, SA, 1>
<SA, <cmd>, PWR, 2>
<PWR, <cmd>, CPU, 3>
<CPU, <cmd>, PWR, 4>
<PWR, <cmd>, SA, 5>
<SA, <cmd><addr>, CPU, 6>
<CPU, <cmd>, SA, 7>
end

//Power on unit:
//Initiators: CPU, GFX, USB, UART, Audio
//Targets: CPU, GFX, USB, UART, Audio
//<I, <cmd><data>, SA, 1>
//<SA, <cmd>, PWR, 2>
//<PWR, <cmd>, T, 3>
//<T, <cmd>, PWR, 4>
//<PWR, <cmd>, SA, 5>
```

```
//<SA, <cmd><data>, I, 6>
//end

//Power off unit:
Initiators: CPU
Targets: CPU, GFX, USB, UART, Audio
<I, <cmd>, SA, 1>
<SA, <cmd>, PWR, 2>
<PWR, <cmd>, T, 3>
end

//MIMO Downstream Memory Write:
Targets: GFX, Audio, USB, PWR, UART
<CPU, <cmd><addr><data>, SA, 1>
<SA, <cmd><addr><data>, T, 2>
end

//MIMO Downstream Memory Write: target powered down
Targets: GFX, Audio, USB, PWR, UART
<CPU, <cmd><addr>, SA, 1>
<SA, <cmd>, PWR, 2> //power up if Target off
<PWR, <cmd>, T, 3> //PWR up Target
<T, <cmd>, PWR, 4> //Target is on
<SA, <cmd><addr><data>, T, 5>
end
```

A.2 FLASH Cache Coherence - Conditional

```
//Read Miss, not dirty
Initiators: P1, P2, P3, P4, P5, P6
Targets: HOME1, HOME2, HOME3
<I, <cmd><addr>, T, 1> //get GET message to home
<T, <data>, I, 2, .167> //send data back if not dirty
end

//Read Miss, dirty
Initiators: P1, P2, P3, P4, P5, P6
```

```
Targets: HOME1, HOME2, HOME3
<I, <cmd><addr>, T, 1> //get GET message to home
<T, <cmd><addr>, P1, 2, .167> //forward GET to owner
<T, <cmd><addr>, P2, 2, .167> //forward GET to owner
<T, <cmd><addr>, P3, 2, .167> //forward GET to owner
<T, <cmd><addr>, P4, 2, .167> //forward GET to owner
<T, <cmd><addr>, P5, 2, .167> //forward GET to owner
<T, <cmd><addr>, P6, 2, .167> //forward GET to owner
<P1, <data>, I, 3, .167> //owner sends data to initiator and ack to home
<P1, <cmd><addr>, T, 3, .167>
<P2, <data>, I, 3, .167>
<P2, <cmd><addr>, T, 3, .167>
<P3, <data>, I, 3, .167>
<P3, <cmd><addr>, T, 3, .167>
<P4, <data>, I, 3, .167>
<P4, <cmd><addr>, T, 3, .167>
<P5, <data>, I, 3, .167>
<P5, <cmd><addr>, T, 3, .167>
<P6, <data>, I, 3, .167>
<P6, <cmd><addr>, T, 3, .167>
end


//Write Miss, not dirty
Initiators: P1, P2, P3, P4, P5, P6
Targets: HOME1, HOME2, HOME3
<I, <cmd><addr>, T, 1> //get GETX message to home
<T, <cmd><addr>, P1, 2, .167> //send invals
<T, <cmd><addr>, P2, 2, .167>
<T, <cmd><addr>, P3, 2, .167>
<T, <cmd><addr>, P4, 2, .167>
<T, <cmd><addr>, P5, 2, .167>
<T, <cmd><addr>, P6, 2, .167>
<T, <cmd><addr>, I, 3> //send PUTX back if not dirty
<P1, <cmd>, T, 3, .167> //send INV ack back if not dirty and invalidated
<P2, <cmd>, T, 3, .167>
<P3, <cmd>, T, 3, .167>
<P4, <cmd>, T, 3, .167>
```

```
<P5, <cmd>, T, 3, .167>
<P6, <cmd>, T, 3, .167>
end


//Write Miss, dirty
Initiators: P1, P2, P3, P4, P5, P6
Targets: HOME1, HOME2, HOME3
<I, <cmd><addr>, T, 1> //get GETX message to home
<T, <cmd><addr>, P1, 2, .167> //send invals and forward GETX to owner
<T, <cmd><addr>, P2, 2, .167>
<T, <cmd><addr>, P3, 2, .167>
<T, <cmd><addr>, P4, 2, .167>
<T, <cmd><addr>, P5, 2, .167>
<T, <cmd><addr>, P6, 2, .167>
<P1, <cmd><addr>, T, 3, .167> //send FAck back to home if dirty owner
<P2, <cmd><addr>, T, 3, .167>
<P3, <cmd><addr>, T, 3, .167>
<P4, <cmd><addr>, T, 3, .167>
<P5, <cmd><addr>, T, 3, .167>
<P6, <cmd><addr>, T, 3, .167>
<P1, <cmd><addr>, I, 3, .167> //send putx back to requester if dirty owner
<P2, <cmd><addr>, I, 3, .167>
<P3, <cmd><addr>, I, 3, .167>
<P4, <cmd><addr>, I, 3, .167>
<P5, <cmd><addr>, I, 3, .167>
<P6, <cmd><addr>, I, 3, .167>
end
```

A.3 FLASH Cache Coherence - No Conditional

```
//Read Miss, not dirty
Initiators: P1, P2, P3, P4, P5, P6
Targets: HOME1, HOME2, HOME3
<I, <cmd><addr>, T, 1> //get GET message to home
<T, <data>, I, 2> //send data back if not dirty
end
```

```
//Read Miss, dirty
Initiators: P1, P2, P3, P4, P5, P6
Targets: HOME1, HOME2, HOME3
<I, <cmd><addr>, T, 1> //get GET message to home
<T, <cmd><addr>, P1, 2> //forward GET to owner
<T, <cmd><addr>, P2, 2> //forward GET to owner
<T, <cmd><addr>, P3, 2> //forward GET to owner
<T, <cmd><addr>, P4, 2> //forward GET to owner
<T, <cmd><addr>, P5, 2> //forward GET to owner
<T, <cmd><addr>, P6, 2> //forward GET to owner
<P1, <data>, I, 3> //owner sends data to initiator and ack to home
<P1, <cmd><addr>, T, 3>
<P2, <data>, I, 3>
<P2, <cmd><addr>, T, 3>
<P3, <data>, I, 3>
<P3, <cmd><addr>, T, 3>
<P4, <data>, I, 3>
<P4, <cmd><addr>, T, 3>
<P5, <data>, I, 3>
<P5, <cmd><addr>, T, 3>
<P6, <data>, I, 3>
<P6, <cmd><addr>, T, 3>
end


//Write Miss, not dirty
Initiators: P1, P2, P3, P4, P5, P6
Targets: HOME1, HOME2, HOME3
<I, <cmd><addr>, T, 1> //get GETX message to home
<T, <cmd><addr>, P1, 2> //send invals
<T, <cmd><addr>, P2, 2>
<T, <cmd><addr>, P3, 2>
<T, <cmd><addr>, P4, 2>
<T, <cmd><addr>, P5, 2>
<T, <cmd><addr>, P6, 2>
<T, <cmd><addr>, I, 3> //send PUTX back if not dirty
<P1, <cmd>, T, 3> //send INV ack back if not dirty and invalidated
```

```
<P2, <cmd>, T, 3>
<P3, <cmd>, T, 3>
<P4, <cmd>, T, 3>
<P5, <cmd>, T, 3>
<P6, <cmd>, T, 3>
end


//Write Miss, dirty
Initiators: P1, P2, P3, P4, P5, P6
Targets: HOME1, HOME2, HOME3
<I, <cmd><addr>, T, 1> //get GETX message to home
<T, <cmd><addr>, P1, 2> //send invals and forward GETX to owner
<T, <cmd><addr>, P2, 2>
<T, <cmd><addr>, P3, 2>
<T, <cmd><addr>, P4, 2>
<T, <cmd><addr>, P5, 2>
<T, <cmd><addr>, P6, 2>
<P1, <cmd><addr>, T, 3> //send FAck back to home if dirty owner
<P2, <cmd><addr>, T, 3>
<P3, <cmd><addr>, T, 3>
<P4, <cmd><addr>, T, 3>
<P5, <cmd><addr>, T, 3>
<P6, <cmd><addr>, T, 3>
<P1, <cmd><addr>, I, 3> //send putx back to requester if dirty owner
<P2, <cmd><addr>, I, 3>
<P3, <cmd><addr>, I, 3>
<P4, <cmd><addr>, I, 3>
<P5, <cmd><addr>, I, 3>
<P6, <cmd><addr>, I, 3>
end
```

REFERENCES

[1] *1666-2011 IEEE Standard for System C Language*, IEEE Std. [Online]. Available: http://standards.ieee.org/getieee/1666/download/1666-2011.pdf

[2] *1800-2012 IEEE Standard for System Verilog*, IEEE Std. [Online]. Available: http://standards.ieee.org/getieee/1800/download/1800-2012.pdf

[3] H. Foster, "Why the Design Productivity Gap Never Happened," in *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, Nov. 2013, pp. 581–584.

[4] S. Mitra, S. Seshia, and N. Nicolici, "Post-silicon Validation Opportunities, Challenges and Recent Advances," in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, June 2010, pp. 12–17.

[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*, ser. DAC '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837278, pp. 3–7.

[6] P. Wolkotte, J. Rutgers, P. Hölzenspies, M. Westmijze, R. Blumink, and G. Smit, "An Automated Design-flow for FPGA-based Sequential Simulation," in *ProRISC 2008, 19th Annual Workshop on Circuits, Systems and Signal Processing*, no. 2008/1. Veldhoven, the Netherlands: Technologiestichting STW, November 2008. [Online]. Available: http://doc.utwente.nl/65230/, pp. 126–132.

[7] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, R. Singhal, J. Brayton, S. Steibl, and H. Wang, "Intel Nehalem Processor Core Made FPGA Synthesizable," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723116, pp. 3–12.

[8] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A Reconfigurable Design-for-Debug Infrastructure for SoCs," in *Design Automation Conference, 2006 43rd ACM/IEEE*, 2006, pp. 7–12.

[9] B. Vermeulen and K. Goossens, *Debugging Systems-on-Chip.* Springer International Publishing, 2014.

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

[11] 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*, June 2008, pp. 373–378.

[12] 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*, ser. DAC '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837367, pp. 368–373.

[13] T. Hong, Y. Li, S.-B. Park, D. Mui, D. Lin, Z. Kaleq, N. Hakim, H. Naeimi, D. Gardner, and S. Mitra, "QED: Quick Error Detection Tests for Effective Post-silicon Validation," in *Test Conference (ITC), 2010 IEEE International*, Nov. 2010, pp. 1–10.

[14] 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*, Oct. 2006, pp. 1–10.

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

[16] N. Kitchen and A. Kuehlmann, "Stimulus Generation for Constrained Random Simulation," in *Proceedings of the 2007 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '07. Piscataway, NJ, USA: IEEE Press, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1326073.1326127 pp. 258–265.

[17] P. Mishra and N. Dutt, "Functional Coverage Driven Test Generation for Validation of Pipelined Processors," in *Design, Automation and Test in Europe, 2005. Proceedings*, vol. 2, March 2005, pp. 678–683.

[18] *CoreSight Debug and Trace*, ARM Ltd, March 2015. [Online]. Available: http://www.arm.com/products/system-ip/debug-trace/

[19] *Design Debugging Using the SignalTap II Embedded Logic Analyzer*, Altera Inc., March 2015. [Online]. Available: http://www.altera.com

[20] "EJTAG Trace Control Block Specification," MIPS Technologies Inc. [Online]. Available: http://www.mips.com.

[21] K. Han, J.-S. Yang, and J. 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*, Jan. 2013, pp. 302–307.

[22] 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*, Jan. 2011, pp. 352–357.

[23] 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, Aug. 2012.

[24] 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, April 2013.

[25] 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*, Nov. 2011, pp. 595–601.

[26] K. Basu, P. Mishra, P. Patra, A. Nahir, and A. Adir, "Dynamic Selection of Trace Signals for Post-Silicon Debug," in *Microprocessor Test and Verification (MTV), 2013 14th International Workshop on*, Dec. 2013, pp. 62–67.

[27] 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*, Jan. 2013, pp. 308–313.

[28] 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, May 2013.

[29] 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*, Dec. 2010, pp. 243–248.

[30] Y. Lee, T. Matsumoto, and M. Fujita, “On-chip Dynamic Signal Sequence Slicing for Efficient Post-silicon Debugging,” in *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, Jan. 2011, pp. 719–724.

[31] H. Shojaei and A. Davoodi, “Trace Signal Selection to Enhance Timing and Logic Visibility in Post-silicon Validation,” in *Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on*, Nov. 2010, pp. 168–172.

[32] 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*, May 2007, pp. 69–76.

[33] H. Ko, A. 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, Aug. 2011.

[34] 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*, Sept. 2014, pp. 127–134.

[35] X. Liu and Q. Xu, “Interconnection Fabric Design for Tracing Signals in Post-silicon Validation,” in *Design Automation Conference, 2009. DAC ’09. 46th ACM/IEEE*, July 2009, pp. 352–357.

[36] E. Singerman, Y. Abarbanel, and S. Baartmans, “Transaction Based Pre-to-post silicon Validation,” in *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, June 2011, pp. 564–568.

[37] 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*, May 2007, pp. 95–106.

[38] H. Vranken, T. Garcia, S. Mauw, and L. Feils, “IC Design Validation Using Message Sequence Charts,” in *Euromicro Conference, 2000. Proceedings of the 26th*, vol. 1, 2000, pp. 122–127.

[39] A. Bunker, G. Gopalakrishnan, and K. Slind, “Live Sequence Charts Applied to Hardware Requirements Specification and Verification: A VCI Bus Interface Model,” *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 4, pp. 341–350, Aug. 2005. [Online]. Available: http://dx.doi.org/10.1007/s10009-004-0145-x

[40] 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, proceedings of the Eighth International Workshop on Automated Verification of Critical Systems (AVoCS 2008). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066109003417

[41] M. Barnum, L. Lambrecht, and T. Ozguner, “Method and Apparatus for Guaranteeing Memory Bandwidth for Trace Data,” Sept. 2007, US Patent App. 11/347,415. [Online]. Available: https://www.google.com/patents/US20070220361

[42] B. Mihajlović and Z. Zilić, “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 Great Lakes Symposium on VLSI*, ser. GLSVLSI ’11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1973009.1973075, pp. 331–336.

[43] *AMBA Documentation*, ARM, March 2015. [Online]. Available: http://infocenter.arm.com/

[44] 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, July 2010.

[45] 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*, Jan. 2014, pp. 640–645.

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

[47] J. K. Strayer, *Linear Programming and Its Applications.* Springer, 1989.

[48] *GLPK (GNU Linear Programming Kit)*, March 2015. [Online]. Available: https://www.gnu.org/software/glpk/

[49] *AMPL*, AMPL Optimization Inc., March 2015. [Online]. Available: http://www.ampl.com

[50] *CPLEX*, IBM, March 2015. [Online]. Available: http://www-03.ibm.com/software/products/en/ibmilogcpleoptistud/

[51] *MATLAB*, The MathWorks, Inc., March 2015. [Online]. Available: https://www.mathworks.com/products/matlab/

[52] S. Khuller, A. Moss, and J. S. Naor, "The Budgeted Maximum Coverage Problem," *Information Processing Letters*, vol. 70, no. 1, pp. 39 – 45, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020019099000319

[53] *TLM-2.0. OSCI TLM-2.0 Language Reference Manual*, 2009. [Online]. Available: http://www.systemc.org

[54] S.-Y. Chen, M.-Y. Hsiao, W.-B. Jone, and T.-F. Chen, "A Configurable Bus-tracer for Rrror Reproduction in Post-silicon Validation," in *VLSI Design, Automation, and Test (VLSI-DAT), 2013 International Symposium on*, April 2013, pp. 1–4.

[55] A. Durand, "Modeling Cache Coherence Protocol - A Case Study with FLASH," in *Workshop on Abstract State Machines*, 1998, pp. 111–126.

[56] *Design Compiler*, Synopsys, March 2015. [Online]. Available: www.synopsys.com