A Communication-Centric Framework for Post-Silicon System-on-chip Integration Debug

Yuting Cao
University of South Florida

Follow this and additional works at: https://scholarcommons.usf.edu/etd

Part of the Computer Sciences Commons

Scholar Commons Citation

This Dissertation is brought to you for free and open access by the Graduate School at Scholar Commons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.
A Communication-Centric Framework for Post-Silicon System-on-chip Integration

Debug

by

Yuting Cao

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science and Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Hao Zheng, Ph.D.
Srinivas Katkoori, Ph.D.
Swaroop Ghosh, Ph.D.
Sanjukta Bhanja, Ph.D.
Sandip Ray, Ph.D.

Date of Approval:
November 8, 2019

Keywords: validation, debug infrastructure, trace analysis, specification mining

Copyright © 2019, Yuting Cao
Dedication

To God, my mother Haiqin Jia, and my husband Tiankai Su, whose love made all this possible.
Acknowledgments

Foremost, I would like to express my sincere gratitude to my adviser Prof. Hao Zheng for the continuous support of my PhD study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this dissertation. I could not have imagined having a better adviser and mentor for my PhD study.

Besides my adviser, I would also like to thank the rest of my dissertation committee: Dr. Ghosh Swaroop, Dr. Sandip Ray, Dr Srinivas Katkooi and Dr Sanjukta for their encouragement and insightful comments.

Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of research and the dissertation writing. This accomplishment would not have been possible without them. Thank you!

This research was partially supported by grants and gifts from the Intel Cooperation.
Table of Contents

List of Tables iv
List of Figures v
Abstract vii

Chapter 1: Introduction 1
  1.1 SoC Validation - Overview 1
  1.2 Post-Silicon Debug - Challenges and Techniques 4
  1.3 Communication-Centric SoC Debug 6
  1.4 Specification Mining 7
  1.5 Contributions 8
  1.6 Dissertation Organization 10

Chapter 2: Previous Work 12
  2.1 Communication-Centric System-Level Debug 12
  2.2 On-Chip Debug Infrastructure 14
  2.3 Trace Signal Selection 18

Chapter 3: Message Flow Guided Trace Analysis 20
  3.1 Message Flow Specifications 20
  3.2 Trace Analysis 26
    3.2.1 Flow Execution Scenarios 28
    3.2.2 Algorithm 29
    3.2.3 Illustration 32
  3.3 Dealing with Partial Observability 35
    3.3.1 Message Flow Guided Trace Abstraction 35
    3.3.2 Generalized Trace Interpretation 39
    3.3.3 Handling Missing Events 42
  3.4 Complexity and Accuracy 45
  3.5 Interactive Trace Analysis for More Efficient Debug 46

Chapter 4: On-Chip Communication Tracing Infrastructure 48
  4.1 Communication Event Acquisition 48
  4.2 Event Output 50
    4.2.1 Parallel Output 51
    4.2.2 Interleaved Output 51
List of Tables

Table 4.1 Operations of the event output unit for 3 monitors over 5 cycles. 54
Table 6.1 Runtime results from analyzing traces obtained in different approaches. 72
Table 6.2 Area overhead of the monitoring infrastructure. 73
Table 6.3 Results with different flows under observation. 73
Table 6.4 Impacts of different capacities of event queues. 74
Table 6.5 Comparisons of different event selection methods 74
Table 6.6 Runtime results of trace analysis with different trace signal selections. 77
Table 7.1 An example trace of four events 86
Table 7.2 Mined patterns using the original traces with $\theta = 0.2$ 92
Table 7.3 Mined patterns using the sliced traces with $\theta = 0.2$ 92
Table 7.4 Mined patterns after applying causality filtering 93
Table 7.5 Mined patterns after applying additional threshold 93
List of Figures

Figure 1.1 Structure of an embedded logic analyzer 5
Figure 1.2 The communication-centric post-silicon debug framework for SoC designs. 9
Figure 3.1 A graphical representation of a SoC firmware load protocol. 21
Figure 3.2 Protocol in Figure 3.1 represented in graphical live sequence chart. 22
Figure 3.3 LPN formalization of protocol in Figure 3.1 24
Figure 3.4 LPN formalization of a CPU write protocol. 44
Figure 4.1 Architecture of the Communication Tracing Module (CTM) for SoC integration debug. 49
Figure 4.2 An example of the AXI read event on a communication link, and the output of a monitor attached to that link. 50
Figure 4.3 Detailed architecture of the output unit for the CTM. 52
Figure 5.1 Possible interleavings between two flows 61
Figure 6.1 The RTL model structure 67
Figure 6.2 Peterson’s algorithm on two CPUs 70
Figure 6.3 The SoC prototype with monitor attached to each communication link. 71
Figure 7.1 Overview of the sequential pattern extraction approach architecture 84
Figure 7.2 Structure of a LSTM unit 87
Figure 7.3 The overall structure of the proposed LSTM model 88
Figure 7.4 Mined patterns of original traces and sliced traces with different thresholds $\theta$ 91
Figure A.1 Flow sequence chart of write operation when requested data is not included in Dcache. 110
Figure A.2  Flow sequence chart of write operation when XCache has the exclusive right of requested data.

Figure A.3  Flow sequence chart of write operation when requested data is shared by another component.

Figure A.4  Flow sequence chart of read operation when XCache has the exclusive right of requested data.

Figure A.5  Flow sequence chart of read operation when requested data is shared by another component.

Figure A.6  Flow sequence chart of read operation when requested data is not present in the Cache.

Figure B.1  Flow specification of a cache coherent write operation initiated from CPU1 to instruction cache.

Figure B.2  Flow specification of a cache coherent read operation initiated from CPU1 to instruction cache.

Figure B.3  Flow specification of a cache coherent read operation initiated from CPU1 to data cache.

Figure C.1  CPU write when cache has exclusive right of the requested data.

Figure C.2  CPU write when data only exist in the other CPU’s cache

Figure C.3  CPU write when requested data only reside in Memory

Figure C.4  Cache send write back request to Memory

Figure C.5  CPU read when cache has exclusive right of the requested data.

Figure C.6  CPU read when data only exist in the other CPU’s cache

Figure C.7  CPU read when requested data only reside in Memory

Figure D.1  Flow specification of a cache write back operation initiated from Cache1.

Figure D.2  Flow specification of a cache coherent write operation initiated from CPU1 to Cache.

Figure D.3  Flow specification of a cache coherent read operation initiated from CPU1 to Cache.
Abstract

System-on-Chip (SoC) plays an important role in people’s everyday life. These systems are often deployed in critical applications, bugs discovered after their deployment in field can be extremely expensive, resulting in catastrophic loss of company revenues, compromise of personal and national security, and even human life. Post-silicon debug is a critical component of the validation of modern microprocessors and SoC designs. A major challenge in post-silicon debug is the severely limited observability where only a small number of debug interface signals are available to observe a vast space of internal executions of SoC designs. This dissertation addresses the above issues with a post-silicon system-level communication-centric debug framework for SoC designs. This work considers post-silicon integration debug of SoC designs, which concerns debugging anomalies in executions of communication protocols among various IPs. It consists of three main concepts: a communication event selection method guided by system-level protocols, an on-chip communication monitoring infrastructure, and an off-chip trace analysis method specifically accounting for the system-level protocols. This framework enhances observability, and enables efficient and accurate reconstruction of the internal executions for SoC designs. This dissertation demonstrates the framework with experiments on a non-trivial multicore SoC prototype and further shows that the proposed framework allows for precise interpretation of the SoC behaviors. Moreover, it shows that with only little area overhead, this framework is able to generate rich debugging information.

Comprehensive and well defined specifications are the foundation of the above trace analysis framework and many other SoC design activities. However, such specifications are not
always in a desired form. Modern SoC specifications can be ambiguous, incomplete or even contain inconsistencies or errors that is common for system-on-chip (SoC) design validation. This dissertation addresses this problem by developing a message flow specification mining approach that automatically extracts sequential patterns from SoC transaction-level traces such that the mined patterns collectively characterize system-level specifications for SoC designs. This approach exploits long short-term memory (LSTM) networks trained with the collected SoC execution traces to capture sequential dependencies among various communication events of those traces. Then, a novel algorithm is developed to efficiently extract sequential patterns on system-level communications from the trained LSTM models. Several trace processing techniques are also proposed to enhance the mining accuracy. We test the proposed approach on simulation traces of a non-trivial multi-core SoC prototype. Initial results demonstrate that the trained neural network model has a high correct rate on extracting the implemented specifications of the SoC model.
Chapter 1: Introduction

This chapter briefly reviews the concepts of pre- and post-silicon validation and discusses the current challenges in post-silicon validation and possible solutions. It also reviews related works in post-silicon validation and motivates the proposed communication-centric SoC debug framework, and the specification mining approach in this dissertation.

1.1 SoC Validation - Overview

Computing devices play an important role in our everyday life. Its application expands from ear buds, smart watches to medical devices. Such devices are generally implemented through a System-on-Chip design paradigm. An System-on-Chip (SoC) design is typically constructed by composing a large number of pre-designed hardware or software blocks often referred to as intellectual properties or IPs that are coordinated through complex protocols to implement system-level behavior [1]. In general, SoC engineers start the SoC design with a set of specifications that describe the desired system behaviors and functionalities. Then they convert the specifications into a transaction level model. This model describes the system behavior at a very high abstraction level, thus allowing the designers to explore the architecture of design. After that, designers refine the transaction level model to a register transfer level (RTL) model. The RTL model captures the cycle accurate behaviors and the interconnections to input and outputs of the digital circuits on chip [2]. Once the RTL model is verified, it is then synthesized and mapped onto a gate-level netlist. And finally it is fabricated on a silicon chip.
To ensure the correctness of the design, validation is conducted on the system design in each step of the design process. Here we define validation as an activity to ensure that a product satisfies its specifications, works in target systems, and meets user expectations [3]. As the design process proceeds, more scenarios can be exercised to find bugs missed in the previous design phase. At the same time, the complexity of the design increases, making the controllability and configurability of the system more complex, and it becomes more difficult to identify and fix bugs, especially in the later stages of the design flow. Over the last decade, the number and heterogeneity of IPs integrated in an SoC design have continued to grow, and the trend is towards an even sharper growth gradient as we develop sophisticated systems targeting complex applications like automotives and Internet-of-Things. Unsurprisingly, this has led to an increasing count of design bugs [4, 5]. As these systems are often deployed in critical applications, bugs discovered after their deployment in field can be extremely expensive, potentially resulting in catastrophic loss of company revenues, compromise of personal and national security, and even human life [6]. Recent studies have shown that validation in modern IC development process takes up to 70% of design time and is still increasing [7]. Consequently, it requires bugs to be found as early as possible to reduce the cost and time of debugging.

Pre-silicon validation aims to verify the design before it is fabricated on a silicon chip. It is a very important research topic because the cost of debugging and refining the design is relatively low compared to changing the design after it is fabricated on the silicon. Pre-silicon validation techniques include software simulation, emulation using field-programmable gate arrays (FPGAs), and so on. Simulator is a software program that mimics the behavior of a design model running in a test environment. To achieve a satisfying verification coverage of the system, a large number of test vectors need to be generated and applied to the design under test. As the complexity of the design increases, the space of design behavior grows exponentially, and more test vectors are needed. The speed of simulation is multiple orders of
magnitude slower than the speed of the silicon implementation. This is due the reason that
the simulation is executed via software. Therefore, with the limited validation time allowed,
only a limited number of test vectors can be applied, and only a small portion of the design
can be simulated and tested during pre-silicon validation, leading to low coverage [8].

Emulation is another technique used very commonly in pre-silicon validation. It verifies
the system by implementing the design onto FPGAs and runs up to 3 orders of magnitude
faster than simulation. However, the speed of emulation is still much slower than the silicon
implementation. Due to the above limitations, it is impossible for pre-silicon verification to
validate the design with sufficient coverage. Thus, it cannot guarantee that the first silicon
is error-free.

Post-silicon validation starts when the first silicon prototype is fabricated, and continues
until the product is released to the market. It makes use of pre-production silicon SoCs to
ensure that the fabricated design works as desired under actual operating conditions with
real software. Since the silicon executes at target clock speed, post-silicon executions are
billions of times faster than simulation and several orders of magnitude faster than emulation.
This makes it possible to explore the design state space deeper and to identify errors missed
during pre-silicon validation. However, post-silicon validation is under aggressive timeline
requirement in order to meet the mass production timeline governed by market economics.
That is, the post-silicon debuggers need to perform high-quality validation within the limited
and fixed timeframe. Any bug escape, or missing of the timeline may cause potential product
recall or even cancellation that causes millions to billions of dollars in revenue.

Post-silicon validation ensures the correctness of a SoC chip in three disciplines: First, it
validates the functionality of the SoC prototype. Second, post-silicon validation checks the
compatibility of the SoC under different versions and types of operation system. Finally, it
validates the electrical properties of the SoC components and IO pins.
1.2 Post-Silicon Debug - Challenges and Techniques

Post-silicon debug is very labor-intensive and may take months to finish. It has become the most time-consuming part (on average 35%) of the circuit development process. This is because, as ITRS roadmap states, the time to locate the root cause of a problem grows exponentially with the advances in process technology that produces larger, denser, and more complex designs [9]. On the other hand, the physical limitation of the silicon implementation makes it difficult to obtain full observability and configurability of the target chip. Compared to the emulation or FPGA model used for pre-silicon validation, where hundreds or thousands of signals are available for debugging, in silicon one can only access about a hundred signals during an execution. A promising technique to gather internal signal information during post-silicon validation is by inserting a design-for-debug (DfD) component into the system design. Two of the most commonly used techniques are scan chains and trace buffers. This dissertation discusses the detailed definitions and debug technologies based on these two techniques in the following section.

The scan-based techniques reuse the internal scan chains that are placed in the Circuit Under Debug (CUD). Scan chain was originally designed to increase the controllability and observability of the system during manufacturing test. It uses the functional pins as scan pins to load multiple scan chains concurrently to reduce test time [10].

For post-silicon validation purpose, these scan chains are concatenated, where internal states are loaded and unloaded through a serial interface. When an internal state of the system is needed during the post-silicon debug process, debuggers can stop the system and enable scan chains to capture and offload the internal state elements (scan dump). After a scan dump is finished, the system can be resumed from where it is stopped. In the late 90s, majority of the debug were conducted with the aid of e-beam probing, needles, dedicated pins, and scan chains. As the IC became more complex with ever-shrinking size, authors
of [11] point out that the traditional scan chain alone is not enough anymore. They argue that certain pins are no longer free for scanning due to the multiple metal layer. To address the above issues, they present a general-purpose debug framework that allows flexibility to access scan chains through JTAG port.

Another widely used DfD techniques is the *trace-based techniques*. It is implemented by using embedded logic analyzers (ELA) with a trace buffer. Figure 1.1 shows an example of the ELA structure. There are four components of an ELA: a control unit, a trigger unit, a sample unit (trace buffer) and an offload unit. The control unit is in charge of all the other units inside an ELA. The trigger unit monitors a set of trigger signals to detect certain trigger events and activates the sample unit to start data acquisition. The sample unit contains a trace buffer to record data on selected signals, and the offload unit outputs the data through low-bandwidth device pins. The amount of data that can be acquired by a trace buffer is limited by two factors below:

- Trace buffer width limits the number of observable trace signals.
• Trace buffer *depth* limits the numbers of samples on the observable trace signals to be stored.

Despite the wide use of the above two techniques, they have severe limitations. During post-silicon validation, scan chains can be very useful when the system is deterministic, allowing CUD to be stopped and resumed from any state of interest. However, modern systems often include multiple clock domains for power efficiency. When CUD is stopped, it is very hard to obtain a coherent system state of all clock domains. Moreover, it is difficult to decide when to stop the CUD as often there is little knowledge about the cause of the bug. For these reasons, scan chains are not practical for complicated systems. One the other side, trace buffer based techniques offer temporal observability compared to scan chains, making trace analysis possible even when the location of the bug is not known. However, because of the limitation of the trace buffer *width*, the number of signals that can be observed is limited. This problem can be mitigated using trace information filtering [9] and compression techniques [12].

### 1.3 Communication-Centric SoC Debug

Post-silicon debug is a critical component of the design validation life-cycle for modern microprocessors and SoC designs. Unfortunately, it is also a highly complex component, performed under aggressive schedules and accounting for more than 35% of the overall design validation cost. Consequently, it is crucial to develop methods and techniques for streamlining and automating post-silicon validation activities.

With the increasing complexity of modern SoC designs nowadays, debugging individual IP blocks by themselves is not enough anymore. The complexity of the SOC increasingly resides in the interactions among the IP blocks. Errors in these interactions form some of the most subtle and hard-to-debug errors in validation. This is because, executions in modern
SoC designs entail significant interleavings of a large number of such protocols, and errors can occur because of a subtle race condition in a specific interleaving which is difficult to exercise or repeat [13]. Furthermore, while individual IPs are often reused across products, (resulting in their core functionality being hardened through multiple validation across products) their specific integration, — and consequently, the protocols involved in their communications —, is unique to each individual SoC. This results in unique bugs arising in the communication component of each SoC, which are hard to isolate, replay, triage, and root-cause. The situation is particularly exacerbated by the fact that due to partial observability only a small set of events in the participating protocols can be actually observed in each execution, making it harder to pinpoint the exact interleaving involved in the execution. Debug must be conducted at a higher abstraction level where the computation threads and communication threads interact. Therefore, communications between the IP blocks are the natural focus for system level debug [14].

This dissertation considers post-silicon integration debug of SoC designs, which concerns about debugging anomalies in executions of communication protocols among various IPs. A comprehensive approach is developed to reconstruct the protocol-level communication behavior from the observed silicon traces of SoC designs. It consists of an on-chip communication monitoring infrastructure, an off-chip trace analysis methods, and an observability selection method. Given a collection of system-level communication protocols and a trace of (partially observed) hardware signals, this approach infers, with a certain measure of confidence, the protocol instances (and their interleavings) being exercised by that trace.

1.4 Specification Mining

Another big challenge in SoC verification is that well defined specifications are not always available in a desired form. This is because writing formal specifications is time-consuming, and it requires the developers have a comprehensive knowledge on the SoC design. Moreover,
as the complexity of the SoC design gradually increases, the connection between the original specification (from design team) and the system behavior (from implementation team) becomes imprecise and disjoint. The lack of formal specifications may lead to potential misunderstandings of the design, and cause unintentional misbehavior to be implemented. It also hinders the debug process since effective debug requires well-formed comprehensive formal specifications. In order to overcome those challenges, an automatic specification mining approach is crucial for effective SoC validation.

In the literature, some existing methods [15, 16, 17, 18, 19, 20] allow extracting specifications from system execution traces. However, they are mainly designed for software, and cannot be directly applied to SoC designs where multiple protocols are typically executed concurrently, and only limited observability is allowed to observe the execution traces. To address those limitations of the existing methods, this dissertation develops a novel specification mining framework that is able to account for the concurrent nature of SoC designs, where execution traces are the results from multiple message flows executed in parallel, and that the availability of each trace event differs based on the specific architecture design. In this mining approach, the state-of-art LSTM neural networks, which are effective at capturing sequential dependencies, are trained with the SoC execution traces. Subsequently, sequential patterns are automatically extracted from the trained LSTM models, where message flow specifications can be formed from the mined sequential patterns. This flow specification collectively characterize system-level specifications for SoC designs, providing foundation for the proposed trace analysis approach in this dissertation and many other SoC design activities.

1.5 Contributions

In this dissertation, we present a post-silicon debug framework for SoC designs, as the key components is shown in Figure 1.2. The overall objective is to reconstruct system-level communication behavior from partially observed silicon traces. Accurate reconstruction
results can offer SoC debuggers an abstract and structured view of the internal execution of an SoC on the system-level protocols. The contributions rest on the developments of the three key constituent elements cooperating in a single framework. To the best of our knowledge, the proposed framework is the first comprehensive approach that elevates SoC debug to the system level guided by system-level protocols.

- The off-chip trace analysis method infers possible scenarios about internal executions of system-level protocols corresponding to an observed silicon trace that is possibly incomplete and lossy. It is possible that the inferred execution scenarios are incomplete on certain system-level protocols due to the incomplete and lossy nature of the observed traces. Therefore, this method also identifies those protocols that need to be focused for observation in the next debug run.

- The on-chip communication monitoring infrastructure monitors communication links of interest, detects, and outputs communication events transferred on those links to enhance observability and assist the efficient and accurate trace analysis. The output
of this infrastructure is sequences of communication events which are used by the trace analysis to derive the high level execution scenarios of system-level protocols.

- The communication event selection method chooses a subset of debug-critical events for more effective observation of a set of system-level protocols identified by debuggers or by the trace analysis method as described above. This method helps to reduce losses of the detected communication events caused by the limited bandwidth of the trace interface. When there are only limited communication events can be outputted, this method tries to select events are critical to the trace analysis. Such information is used to configure the monitoring infrastructure so that the critical events are outputted with higher priority.

In addition, this work propose a novel sequential pattern mining framework that automatically extracts sequential patterns from inherently concurrent SoC execution traces. These patterns may be recurrent in individual traces, and repeat themselves in multiple different traces. This approach exploits long short-term memory (LSTM) networks trained with the collected SoC execution traces to capture sequential dependencies among various communication events of those traces. Then, a novel algorithm is developed to efficiently extract sequential patterns on system-level communications from the trained LSTM models. Several trace processing techniques are also proposed to enhance the mining accuracy. We test the proposed approach on simulation traces of a non-trivial multi-core SoC prototype. Initial results demonstrate that the trained neural network model has a high correct rate on extracting the implemented specifications of the SoC model.

1.6 Dissertation Organization

This dissertation is organized as follows. We present a review of previous related work in Chapter 2. The proposed off-chip trace analysis approach is explained in Chapter 3. Chapter
4 presents the on-chip communication monitoring infrastructure, and Chapter 5 discusses the communication event selection method for observability enhancement. To demonstrate the efficiency of this framework, a series of experiments are conducted and their results are discussed in Chapter 6. The specification mining approach is presented in Chapter 7. Chapter 8 summarizes the dissertation, and points out some future directions. All the flow specifications used in the case studies are given in Appendix.A, B, C and D. Appendix.E shows the copyright permissions for works used in this dissertation.
Chapter 2: Previous Work

2.1 Communication-Centric System-Level Debug

The work presented in this dissertation is closely related to communication-centric and transaction-based debug. An early pioneering work is described in [14], which advocates the focus on observing activities on the interconnected network among IP blocks and mapping these activities to transactions for better correlation between computations and communications. Therefore, the communication transactions, as a result of software execution, provide an interface between computation and communication, and facilitate system-level debug. This work is extended in [21, 22, 23, 24]. However, this line of work is focused on the network-on-chip (NoC) architecture for interconnect using the run/stop debug control method. In another work [25], a machine-learning-based anomaly detection approach is proposed for post-silicon bug diagnosis. This work involves two steps: it first applies clustering techniques to all signals within a time step to locate the accurate occurrence of a bug, and then a second round of clustering-based anomaly detection identifies the responsible bug signals. While this method achieves bug localization with higher accuracy compared to other state-of-the-art solutions, there is no discussion regarding which signals are selected for observation. The approach is conducted with the assumption that the critical signals related the failure are observed, which is not likely due to the limited observability.

A similar transaction-based debug approach is presented in [26]. It proposes an automated extraction of state machines from high-level design models. From an observed failure trace, it performs backtracking on the transaction-level state machine to derive a set of transaction traces that lead to the observed failure state. In the subsequent step, bounded
model checking with constraints on the internal variables is used to refine the set of trans-
action traces to remove the infeasible traces. This approach requires user inputs to identify
impossible transaction sequences; however, the user may not find the states causing the
failure if the transaction traces leading to the observed failure state are long. Moreover,
backtracking from the observed failure state requires pre-image computation, which can be
computationally expensive. Another transaction-based online debug approach is proposed
in [27] to address these issues. This approach utilizes a transaction debug pattern specifi-
cation language [28] to define properties that transactions should meet. These transaction
properties are checked at runtime by programming debug units in the on-chip debug infra-
structure, and the system can be stopped shortly after a violation is detected for any one of
these properties. In this sense, it can be viewed as the hardware assertion approach in [29]
elevated to the transaction level.

In [30], a coherent workflow is described where the result from the pre-silicon validation
stage can be carried over to the post-silicon stage to improve efficiency and productivity of
post-silicon debug. This workflow is centered on a repository of system events and simple
transactions defined by architects and IP designers. It spans a wide spectrum of post-
silicon validation, including DFx instrumentation, test generation, coverage, and debug.
The DFx instruments are automatically inserted into the design RTL code driven by the
defined transactions. This instrumentation is optimized for making observable a large set
of events and transactions. Test generation is also optimized to generate only the necessary
but sufficient tests to allow all defined transactions to be exercised. Moreover, coverage for
post-silicon validation is defined at the abstract level of events and transactions rather than
the raw signals, and thus can be evaluated more efficiently. In [31], a model at an even
higher-level of abstraction, flows, is proposed. Flows are used to specify more sophisticated
cross-IP transactions such as power management, security, etc., and to facilitate reuse of the
efforts of the architectural analysis to check HW/SW implementations.
2.2 On-Chip Debug Infrastructure

Despite the large amount of work done with on-chip monitoring instrumentation, only a few of them focus on transaction monitors. Furthermore, among the existing works for transaction monitoring instrumentation, none is applicable for protocol-guided trace analysis. Lamport [32] proposes an interesting idea to synchronize a system of logical clocks to obtain a global event order where the partial ordering “happening before” is extended to a somewhat arbitrary global ordering. This work is further extended by Gharehbaghi and Fujita [33] where they apply Lamport’s algorithm into a network-on-chip for post-silicon validation. To improve the accuracy of extracted order, the local partial ordering is enriched by considering information from the neighboring tiles. The work in [34] talks about another interesting on-chip monitoring infrastructure targeting at run-stop debugging method. While this monitor allows observing communication activities at three different granularities (e.g., individual data words [elements], whole requests/responses, a whole transaction), it is primarily designed to control and stop the SoC when certain conditions are triggered; thus, it is not appropriate for communication-centric debugging. Ciordas et al. [35] proposed the first monitoring service to provide run-time observability of NoC behavior and supporting system-level debugging. However, they offer no explanation on how the detected events are outputted for off-chip analysis. Several similar works[36, 37] have been conducted to aid NoC verification.

Gharehbaghi and Fujita [38, 39, 40] introduce an on-chip instrumentation that allows transaction level message abstraction using formal specifications of the bus communication protocols. This methodology is highly automated as the on-chip monitor circuit can be automatically generated using the formal specification of the on-chip buses’ communication protocols, and its overhead (memory-dominated) is relatively small. They propose to consider only the start and end of transactions to build the transaction sequence from the
signal events; as a result, the area overhead is very low since the instrumentation includes the full communication protocols. The authors also propose an innovative encoding technique that reduces the address bits to two bits, representing three different states: SAME, SEQ, OTHER. These states indicate relationships between two consecutive transactions. When the state is SAME for an event, it shares the same slave address as the previous event; SEQ means the current address is one word different from the previous event; and OTHER represents any other address relationships. Despite its low area overhead, this method suffers from the inability to detect implementation errors that are not observed. Moreover, it lacks the ability to check the overall system communication protocols as it only focuses on communication interface protocols.

Work in [41] proposes another checking solution for microprocessor cores on acceleration platforms where it compares the outcome of each instruction with an architectural golden model. An innovative idea proposed is that they decouple the transaction tracing from transaction checking. This work employs bug models and detection mechanisms that are specific to microprocessors, hence their extension to general hardware and many-core systems is not obvious.

The above issue is addressed in [42] where an on-chip instrumentation BiPeD is inserted for each communication link in the circuit under debug. The BiPed learns the communication interface’s protocol during pre-silicon stage and reconfigures its detection hardware to check the learned protocols during the post-silicon validation. Once a bug is detected, the recent history of observed activity is transferred off-chip for analysis. With the recent history of the protocol-level activity related to detected bug, BiPed provides rich set of debugging information including bug location, time of occurrence, related critical signals, etc. The methodology is tested on an industrial-size OpenSPARC T2 design, and is able to provide intuitive debugging information regarding an observed failure. While BiPed is effective for detecting and locating many hardware bugs, the quality of the learned protocols depends
heavily on pre-silicon, where its completeness and correctness cannot be guaranteed. Moreover, the circular buffer implemented for each communication interface introduces large area overhead.

Another promising transaction monitor is proposed in [43, 44, 45]. This series of work focuses on transaction monitoring on NoC. While the NoC implements specific communication traffic protocols that are completely different from traditional bus-centered SoC communication strategies, they share similar concepts of collecting, compressing, and offloading transaction events. The authors present a generic NoC run-time monitoring service for NoC-based SoCs that provides non-intrusive data capturing and translation, configuration features on-the-run regarding events and attributes, and on-the-fly setup to control the monitoring architecture, with affordable overhead of 15-20% area cost. However, transaction events collected on each network interface is sent to a monitoring service access point (MSA) through the NoC system, introducing additional monitoring traffic besides user traffic. As a result, this causes potential hidden costs of redesigning the NoC to accommodate the monitoring traffic.

Authors of [45] provide a standard template for monitor instrumentation design to measure key system parameters in real-time. Even though necessary logics are implemented to allow certain reconfigurability, such as eliminating unnecessary components; among the six included components in this template, only two of them, protocol specific front end (data filtering) and control and status registers (configuring), are relevant for SoC on-chip monitoring instrumentation. Moreover, the overhead reported for this template is nontrivial, the configuring component takes more 17000 NAND2 equivalent gates, and the PSFE consumes more than 8000 equivalent gates. What’s more, the author failed to discuss how the monitored transactions should be stored or loaded off-chip.

Another work in [46] presents a transaction-based debug infrastructure that through monitoring, filtering, and a debug network (Debug Unit) can conduct efficient on-chip as-
sertion checking for online debug and online system recovery. It allows for interactive debug in which an external debug platform programs the finite state machine for debug units, and in the meantime, it filters observed activities according to the considered assertions at each round of debugging. The Debug Unit (DU) includes Local Debug Unit (LDU) for every two debug monitors, and they are connected through a tree-based network to a top Central Debug Unit (CDU). Each DU includes a set of relative assertions to be checked at run-time. When an assertion fails, CDU sends a recovery package containing the error type and additional information to the erroneous master for effective recovery process. Note that because the assertion is conducted on each individual DU, two first in first out storing units are implemented for each of its input to store necessary information. As a result, this work suffers from potential memory overhead due to the numbers of FIFOs needed for each DU. The authors also failed to discuss possible solutions to potential message-lost situations.

Intel developed the Trace Hub that provides a set of functional blocks to perform full system debugging [47]. It includes a set of sophisticated hardware blocks that support on-chip activity monitoring, collecting, storing, and offloading. ARM also developed a similar debug infrastructure CoreSight [48] that addresses the requirement for a multi-processor debug and trace solution with high bandwidth for entire systems beyond the processor. It includes a library of modular components and interconnects, a set of architecture discovery and identification methods to allow for flexible system design, easy inclusion of differentiated debug/trace functions, and standard implementation of the ARM Debug Interface for debug tools to work with. This framework includes a tracing macrocell that is designed for non-intrusive real-time software tracing. Those two debug infrastructures are well known, and can be found in many industry designs.
2.3 Trace Signal Selection

Previous work on trace signal selection such as [49, 50, 51, 52] is typically applied to gate-level design models, and the quality of the results is evaluated by the commonly used State Restoration Ratio (SRR) metric. SSR calculates the ratios of the overall internal signal states that can be referred from a set of observed signal. Sai et al. [53] point out that it is difficult to scale the traditional SRR-based methods to large and complex SoC designs. They are not suitable for evaluating trace signal quality due to the following reasons:

- SRR treat all signals equally, while some signals are more important than others.
- SRR favors big arrays, which may not be useful for debugging.
- Signals selected at the gate level are often irrelevant to system-level functionalities.

To address the above problems, they propose a new metric assertion coverage and conduct the signal selection on the level of system-level communication protocols [53]. The assertion coverage measures of a set of selection signals is defined as the the portions of the predefined assertions can be extracted. In addition, that paper introduces a new signal selection algorithm inspired from Google’s PageRank algorithm [54] which focuses more on the connectivity of a system. This algorithm starts by creating a direct graph, and for each instance, passes its weight to its outgoing edges equally. After a number of rounds, each instance retains a stable weight representing its connectivity. The algorithm selects the top signals with the highest connectivity. This algorithm can be applied to both netlist and RTL model. It is the only existing research that raised the abstraction level of signal selection algorithm from gate level to RTL level. While [53] shows significant improvement compared to traditional gate level signal selection algorithms, its effect is mitigated by the resource limitation of the on-chip debug infrastructure. The proposed selecting scheme in [53] is guided by a metric that treats every communication protocol event equally, and it
failed to consider special events that are critical for a comprehensive interpretation of the system behavior. Furthermore, the algorithm in [53] requires the interleaving graph of all flow instances to generate a set of selection. Such a requirement is impractical as itself is one of the critical information needed to be extracted during post silicon validation.

Authors of [55] also argue that the SRR is not suitable for practical use. Instead, they propose to conduct signal selection at the system level. That work uses functional coverage to measure the effect of a set of signals for the system debug. The system level specifications in this work is represented by an ordered sequence of flow messages. For a set of selected signals to observe, the functional coverage is defined as the portions of flow messages covered among the overall messages integrated for all specifications. In addition, this work implements an automatic signal selection algorithm that finds the set of signals with the optimal frequency coverage. The proposed algorithm goes through the following steps: First, it takes a family of protocols and decomposes each messages with multiple data fields into multiple copies that contain only one data field. It then groups the messages into a subset based on the communication channel each message belongs to. After that, the algorithm applies a linear program to the protocol family and find sets of flow messages within the 5% maximum frequency coverage. For high-reward solutions, it calculates their heuristic intervals individually. The heuristic interval indicates the quality of the spacing of a set of selected signals and it measures the average number of messages in each gap. The selection with the smaller interval value is always preferred. Finally, it selects the optimal solution considering both frequency coverage and interval values.
Chapter 3: Message Flow Guided Trace Analysis

In this chapter, we describe a trace analysis method in which the observed signal traces are interpreted with respect to system-level protocol specifications. In general, the trace analysis can offer debuggers an abstract and structured view of internal executions of the SoC in terms of communication transactions among the IP blocks of the SoC.

3.1 Message Flow Specifications

In engineering field, there are two approaches to representing the system protocols: informal and formal representations. Informal representation is human friendly and uses common graphical notation for better understandability and easier communication with the client. The formal representation, on the other hand, is designed to be machine friendly. It is usually built on strong mathematical notations and proofs for more automated verification purpose.

System development usually needs to create protocol in both formal and informal formats. At the beginning of the product development cycle, system designers create the specification in graphical (informal) form, providing good understandability while still in a standard graphical manner. After the design is finalized, specifications in formal format are developed for verification purpose. Usually they require manual translation from an infor--
mal description to a formal description, which consumes large amount of time and efforts as modern system involves massive amount of complicated specifications. [58] introduces a tool that translates live sequence chart into colored petri-nets that can be used to speed up the translation process.

An SoC design involves integration of numbers of IPs that communicate through complex protocols. Such system level protocols are typically specified in architecture documents as message flow diagrams. In this dissertation, we use the words “protocol” and “flow” interchangeably.

<table>
<thead>
<tr>
<th>Driver</th>
<th>Device</th>
<th>CE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Reset Device</td>
<td>3. Copy fw from SM to IM</td>
<td>( F_{tm} )</td>
</tr>
<tr>
<td>2. Copy fw to SM</td>
<td>Auth_rec</td>
<td></td>
</tr>
<tr>
<td>( \text{Load}_f )</td>
<td>( \text{Auth}_{\text{res}} )</td>
<td></td>
</tr>
<tr>
<td>Report(sts)</td>
<td>( \text{Lock IM; then verify signature of fw in IM} )</td>
<td></td>
</tr>
<tr>
<td>( \text{sts} = \text{PASS} )</td>
<td>( \text{Ack} )</td>
<td></td>
</tr>
<tr>
<td>5. Check sts</td>
<td>( \text{Unlock IM} )</td>
<td></td>
</tr>
<tr>
<td>6. Copy fw from IM to LM</td>
<td>( \text{Unlock IM} )</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1: A graphical representation of a SoC firmware load protocol.

Figure 3.1 shows a protocol example that authenticates and loads a firmware during system boot for firmware upgrade in Business Process Model and Notation (BPMN). BPMN is a standard for business process modeling that provides a graphical notation for specifying business processes in a Business Process Diagram (BPD), based on a flowcharting technique very similar to activity diagrams from Unified Modeling Language (UML) [2].
To start this protocol, the Driver resets Device and copies the needed firmware to a place in System Memory (SM) and notices the Device to load it. With the location of firmware provided from Driver, Device can retrieve firmware to Isolated Memory (IM) and send the message Auth\_req to Crypto-engine (CE), providing the location of the copied firmware, and asking for authentication. After verifying signature of firmware in IM, CE replies with PASS/FAIL status \((sts)\). Upon receiving the PASS \(sts\) such that \(sts = PASS\), the Device sends report message to Driver and acknowledgement message to CE and then jumps to the firmware from Local Memory (LM).

The BPMN format used here is a very detailed format, and it is mostly used in business field. A more commonly used graphical format in computer engineering field is sequence diagrams. It represents a life cycle of an IP and the interactions between the IPs implemented in a SoC design. Commonly used sequence diagrams include UML sequence diagrams, message sequence diagrams and live sequence charts [58].

![Live sequence chart](image-url)  
Figure 3.2: Protocol in Figure 3.1 represented in graphical live sequence chart.
Figure 3.2 shows the live sequence chart representation of the protocol in Figure 3.1. In this graph, we can clearly see that the relative time and contents of communications among components are omitted. Unlike the BPMN format in Figure 3.1, sequence diagrams is more abstract as the internal activities of components are not shown.

This dissertation focuses on algorithmic analysis of system behavior. Therefore, only a formal representation with rigorous semantics, methods and tools for analysis is needed. Such representation selected by this dissertation is the Labeled Petri-Nets (LPN).

A LPN is a formalization of state transition system behavior and it is capable of describing sequencing, concurrency, and choices. Compared with sequence diagrams, LPNs are more machine friendly, and can be analyzed using mathematical techniques and tools. Formally, an LPN representation for a flow is a tuple $(P, T, s_0, E, L)$ where:

- $P$ stands for a finite set of places,
- $T$ stands for a finite set of transitions,
- $s_0 \subseteq P$ stands for the initial marking.
- $E$ stands for a finite set of events.
- $L : T \rightarrow E$ stands for a labeling function that maps each transition $t \in T$ to an event $e \in E$.

As for each transition $t \in T$, its preset, denoted as $\bullet t \subseteq P$, is the set of places connected to $t$; And its postset, denoted as $t \bullet \subseteq P$, is the set of places that $t$ is connected to. A marking of a LPN is a set of places marked with tokens. There are two special states associated with each flow: $s_0 \subseteq P$ is the set of initially marked states, also referred to as the initial state; $s_\perp \subseteq P$ is the set of end states not going to any transitions. Each flow is associated with one start and several end events. An event $e \in E$ is a start event if $e = L(t)$ and $\bullet t \subseteq s_0$. An event $e \in E$ is an end event if $e = L(t)$ and $t \bullet \subseteq s_\perp$. In Figure 3.3, $s_0 = \{p_1\}$, and
Its start event is \( t_1 \), and its end event is the one labeled for transitions \( t_4 \) and \( t_5 \). The occurrence of a start event indicates the beginning of a flow execution, while the occurrence of an end event indicates the completion of a flow execution.

The communication protocol shown in Figure 3.1 is represented by the LPN shown in Figure 3.3. This format, compared with sequence diagrams, is even more abstract. It removes all the structure information from a system, and represents only communication activities among the components.

In this and the following figures for LPNs, the labeled circles denote places, and the labeled boxes denote transitions. Each transition is labeled with its name and the associated event. Each event is a tuple \((src, dest, cmd, \{data\_field\})\) where \( cmd \) is a command sent from a source component \( src \) to a destination component \( dest \), and \( data\_field \) represents the payload information carried by the event. It can be \( addr \) that carries the memory address at the target block where \( cmd \) is applied, or other information such as \( sequence\_ID \) and \( tag \) to identify an event. The information defining an event can be classified as \( static \) including
\texttt{src, dest} and \texttt{cmd}; as well as \textit{dynamic} including various \texttt{data fields} which are generated during SoC execution.

The operational semantics of a LPN is defined by transition executions. A transition can be executed after it is \textit{enabled}. A transition \( t \in T \) is enabled in a state \( s \) if every place in its preset is included in \( s \), i.e. \( \bullet t \subseteq s \). The set of enabled transactions in state \( s_i \) is denoted as \( \text{enabled}(s_i) \). Execution of \( t \subseteq \text{enable}(s) \) results in a new state \( s' \) such that

\[
s' = (s - \bullet t) \cup t \bullet.
\]

Let \( s' = t(s) \) denote the new state \( s' \) after \( t \) is executed in \( s \). When \( t \) is executed, the labeled \( e \) is emitted. Therefore, a sequence of transaction execution

\[
t_0 \ t_1 \ t_2 \ldots \ t_i \ldots
\]

results in a sequence of events

\[
e_0 \ e_1 \ e_2 \ldots e_i \ldots, \text{such that } \forall i \geq 0, e_i = L(s_i)
\]

As a result, information exchanges among components in a design can be modeled by sequences of LPN transition executions. Execution of a flow completes if its \( s_\perp \) is reached.

An \textit{execution} of a flow \( F \) is a sequence of events \( (e_0, e_1, \ldots, e_n) \) such that there is a sequence of transition firings \( (t_0, t_1, \ldots, t_n) \) of \( F \) where the following conditions hold.

- \( \forall 0 \leq i \leq n, \ e_i = L(t_i), \)

- \( \forall 0 < i \leq n, \ \bullet t_i \subseteq t_{i-1} \bullet, \)

- \( s_0 \subseteq \bullet t_0, \text{ and } t_n \bullet \subseteq s_\perp. \)
The definition above indicates that an execution of $F$ is the results of a sequence of transition firings from the initial state to the end state where transition $t_i$ is causally dependent on $t_{i-1}$. For example, the flow in Figure 3.3 has two executions: $\{t_1, t_2, t_5, t_4\}$, and $\{t_1, t_2, t_3, t_5\}$. Note that the transition numbers are used to represent events in the example above for simplicity.

During an execution of a SoC design, instances of flows $\tilde{F}$ that it implements are executed. Suppose that the set of flow instances executed is $\{F_{i,j} \mid F_i \in \tilde{F}\}$. The SoC execution yields a trace $\rho$

$$\rho = (E_0, E_1, \ldots, E_n)$$

where $E_i = \{e_{i,0}, \ldots, e_{i,k}\}$ is a set of events executed at time $i$, and $e_{i,*} \in \bigcup E_{i,j}$ for every $e_{i,*} \in E_i$.

Intuitively, an SoC execution trace is the result of multiple flow instances executed in an parallel nature. It is a sequence of sets of events where orderings of events in the same set of a trace are indistinguishable. Given a trace $\rho$ and two events $e_i$ and $e_j$, we define $e_i < e_j$ if $e_i \in E_i$, $e_j \in E_j$, and $i < j$. This notation is naturally extended to sequences of events.

### 3.2 Trace Analysis

In a typical validation setting, the SoC is executed in a test environment until it is terminated by the test environment or the system crashes due to a failure. During the execution, a trace on a small number of observable signals is streamed off the chip for debugging. The off-chip analysis includes two broad phases:

- Trace abstraction that translates signal traces into flow traces
- Trace interpretation that maps flow traces into flow execution scenarios

Trace abstraction maps a signal trace into higher-level architectural constructs, e.g., messages, operations, etc. A message such as Authorization request may be implemented
in hardware through a Boolean or temporal combination of specific hardware signals in the NoC fabric between Device and CE, e.g., as a sequence containing a header, a specific value of a sequence of data words, etc. We refer to such architectural constructs as protocol events or flow events. Note that due to limited observability, it may not be possible to map events on a given set of (observed) hardware signals uniquely to a flow event. Finally, a signal trace may be a result from several instances of the same protocol executing concurrently, e.g., a firmware authentication protocol may be invoked when another instance of the protocol has not completed.

Trace interpretation entails mapping a sequence of flow events created during trace abstraction to message flows in order to identify the set of flow instances (and their interleavings) responsible for creating the observed behavior. The trace interpretation takes a finite trace of flow events resulting from the trace abstraction and a set of system flows in LPNs $\vec{F}$, and it generates a set of possible system flow execution scenarios, which is defined below. A flow execution scenario indicates that at a certain point of SoC execution, what types of flows and the number of instances of a particular flow are activated as well as their corresponding current states.

The observed traces may help to identify problems in the message flows, e.g., an interleaving of some protocol executions may lead to an unexpected message being sent or cause the system to crash. More commonly, one finds a bug in the implementation of the flows, i.e., a trace inconsistent with any possible interleaving of the flow executions. Identifying these problems involves significant human expertise, and can often take days to weeks of efforts. The trace analysis method and algorithm presented in this chapter intends to address that hurdle.
3.2.1 Flow Execution Scenarios

A flow execution scenario represents a system state during system execution abstracted on system flow specifications. Let the set of system flows in LPN donate $\bar{F}$. A flow execution scenario can be viewed as a *state of system execution abstracted wrt system flows*, and it is defined as

$$\{(F_{i,j}, s_{i,j}, start_{i,j}, end_{i,j}) \mid F_i \in F\}$$

where $F_{i,j}$ is the $j$th instance of flow $F_i$, $s_{i,j}$ is used by the trace analysis to keep track of the current state of $F_{i,j}$ when an observed trace is interpreted. $start_{i,j}$ and $end_{i,j}$ are two indices representing relative time when $F_{i,j}$ is initiated and completed. The ordering relations can be derived by comparing their $start$ and $end$ indices. For example, for two flow instances in an execution scenario: $(F_{u,v}, s_{u,v}, start_{u,v}, end_{u,v})$ and $(F_{x,y}, s_{x,y}, start_{x,y}, end_{x,y})$, $F_{u,v}$ is initiated before $F_{x,y}$ if $start_{u,v} < start_{x,y}$, or $F_{x,y}$ is initiated after $F_{u,v}$ is completed if $end_{u,v} < start_{x,y}$.

Based on different levels of information captured, this dissertation classifies flow execution scenarios as follows:

- **Type-1** execution scenarios capture the number of instances of each flow specification initiated from a silicon trace, and their relative orderings of initiations.

- **Type-2** execution scenarios, on top of what is captured by **Type-1** scenarios, capture completion of each flow instance. This additional information can be used to identify potential problems if there is any flow instance that is not completed. Furthermore, **Type-2** execution scenarios capture the relative orderings among all flow instances as described above.

- **Type-3** execution scenarios, on top of what is captured by **Type-2** scenarios, capture information on execution paths followed by individual flow instances. This information
can provide a means for debuggers to have a detailed examination on how each flow
instance is executed. These different execution scenarios can be used to provide dif-

erent views of system execution, from coarse-grained to more detailed ones, at different
stages of debug.

3.2.2 Algorithm

For pedagogical reasons, here we assume full observability of all hardware signals is
involved in the flow events. In the next section we will extend the approach to consider
partial observability. Let

$$\text{accept}(F_{i,j}, s_{i,j}, e) = \begin{cases} s'_{i,j} & \text{if } \exists t, t \in \text{enabled}(s_{i,j}) \land (L(t) = e) \land (s'_{i,j} = t(s_{i,j})) \\ \emptyset & \text{otherwise} \end{cases}$$

be a function to decide if event $e$ can be admitted by flow instance $F_{i,j}$ in state $s_{i,j}$ or not. The function returns the corresponding new state if event $e$ can be admitted, otherwise it returns $\emptyset$. This function is used in the trace analysis algorithm later in this chapter.

Given an observed trace $\rho$, the goal of trace interpretation is to construct a set of can-
didate flow execution scenarios whose execution can create the sequence of events in $\rho$. In
other words, $\rho$ is the result of executing the flow instances in those execution scenarios by
following the corresponding LPN operational semantics starting from their initial states.

If every event in $\rho$ is successfully mapped to some flow instance, we can say that $\rho$
is compliant with the given protocol specifications. When this happens, the algorithm
returns a set of flow execution scenarios. On the other hand, inconsistent events may also
be encountered. An event $e_h$ is inconsistent if for each flow execution scenario $\text{scen}$, the
following two conditions hold:

- For each $(F_{i,j}, s_{i,j}) \in \text{scen}$, $\text{accept}(F_{i,j}, s_{i,j}, e_h) = \emptyset$, and
• For each $F_i \in \bar{F}$, $\text{accept}(F_i, \text{init}_i, e_h) = \emptyset$.

The inconsistent event $e_h$ is the one produced by SoC execution but cannot be mapped to any flow instances no matter how the trace prior to event $e_h$ is interpreted. Inconsistent events may indicate possible causes of observed system failures. When the analysis algorithm finds an inconsistent event, it returns the set of partially derived execution scenarios along with the discovered inconsistent event $e_h$.

Considering an observed flow trace $\rho$ and the set $\bar{F}$ of system protocol specifications, Algorithm 1 describes a basic procedure for computing a set of compliant flow execution scenarios. The algorithm operates by keeping track (in variable $\text{Scen}$) of a set of candidate flow execution scenarios compliant with each prefix of $\rho$. At each iteration, for each event $e_h$ in the observed trace, the algorithm updates $\text{Scen}$ by either updating the state of a member of $\text{scene}$ or initiating a new flow instance for each $\text{scene} \in \text{Scen}$ with respect to $e_h$ in every possible way. If $e_h$ cannot be accepted by any existing or new flow instances in $\text{Scen}$, this indicates that trace $\rho$ is inconsistent with $\text{Scen}$. If event $e_h$ is inconsistent with all existing execution scenarios, then the algorithm reports that the trace is inconsistent with $\text{Scen}$.

Given a trace of flow events $\rho = e_1e_2\ldots e_n$, the trace interpretation algorithm starts with an empty set of flow execution scenario $\text{Scen} = \emptyset$. Then, for each $e_h \in \mathcal{E}_l$ where $1 \leq l \leq n$ start with $l = 1$, and for each $\text{scene} \in \text{Scen}$, the following two steps are performed.

• Step 1 For each $(F_{i,j}, s_{i,j}) \in \text{scene}$, if $\text{accept}(F_{i,j}, s_{i,j}, e_h) = s'_{i,j}$, create a new scenario $\text{scene}' = (\text{scene} - (F_{i,j}, s_{i,j})) \cup \{(F_{i,j}, s'_{i,j})\}$, which is added into $\text{Scen}'$.

• Step 2 For each $F_i \in \bar{F}$, create a new instance $F_{i,j+1}$. If $\text{accept}(F_{i,j+1}, \text{init}_{i,j+1}, e_h) = s'_{i,j+1}$, create a new scenario $\text{scene}' = \text{scene} \cup \{(F_{i,j+1}, s'_{i,j+1})\}$, which is added into $\text{Scen}'$.

After $e_h$ is processed, $\text{Scen} = \text{Scen}'$, and the two steps above repeat for the next event $e_{h+1}$. 

30
Algorithm 1: Check-Compliance($\bar{F}, \rho$)

1. Create an empty scenario $scen$
2. $Scen = \{scen\}$
3. foreach $e_h \in \mathcal{E}_l$, $1 \leq l \leq n$ do
   4. $found \leftarrow \text{true}$
   5. $Scen' = \emptyset$
   6. foreach $scen \in Scen$ do
      7. foreach $(F_{i,j}, s_{i,j}) \in scen$ do
         8. $s'_{i,j} \leftarrow \text{accept}(F_{i,j}, s_{i,j}, e_h)$
         9. if $s'_{i,j} \neq \emptyset$ then
            10. Let $scen'$ be a copy of $scen$
            11. $scen' \leftarrow scen' - (F_{i,j}, s_{i,j}) \cup (F_{i,j}, s'_{i,j})$
            12. $Scen' \leftarrow scen' \cup Scen'$
            13. $found \leftarrow \text{false}$
      14. end
   15. end
   16. foreach $F_i \in \bar{F}$ do
      17. create a new instance $F_{i,j+1}$
      18. $s'_{i,j+1} \leftarrow \text{accept}(F_{i,j+1}, init_{j+1}, e_h)$
      19. if $s'_{i,j+1} \neq \emptyset$ then
         20. Let $scen'$ be a copy of $scen$
         21. $scen' \leftarrow scen' \cup (F_{i,j+1}, s'_{i,j+1})$
         22. $Scen' \leftarrow scen' \cup Scen'$
         23. $found \leftarrow \text{false}$
      24. end
   25. end
   26. if $found == \text{true}$ then
      27. return $\{Scen, e_h\}$
   28. end
29. $Scen = Scen'$
30. end
31. return $\{Scen, \epsilon\}$
Based on the discussion above, the trace interpretation algorithm generates two possible results:

- \( \{ \text{Scen}, \epsilon \} \) when \( \rho \) is compliant with the flow specification \( \bar{F} \).

  \text{Scen} is a set of flow execution scenarios, each of which is derived from the observed trace, and \( \epsilon \) is an empty event indicating non-existence of inconsistent events.

- \( \{ \text{Scen}, \epsilon_h \} \) when inconsistent event occurs.

  \text{Scen} is a set of partially derived scenarios and \( \epsilon_h \) is the corresponding inconsistent event. This result provides valuable information for debuggers to root cause system failures.

3.2.3 Illustration

To illustrate the basic idea of the trace analysis algorithm, consider the message flow shown in Figure 3.3. Let \( F_1 \) denote such flow. Suppose that the following sequence of flow events is abstracted from an observed flow trace:

\[
t_1 \ t_2 \ t_1 \ t_2 \ t_3 \ t_4 \ t_5 \ t_5 \ t_4 \ldots
\]  

(3.1)

This trace is interpreted from the first event to the last in order to derive all possible flow execution scenarios. Here transition names in the LPN are used to represent the flow events in the trace. At the beginning, event \( t_1 \) is processed first. According to the flow specification \( F_1 \), we know that one instance of such flow \( F_1, F_{1,1} \), is activated by the SUD as \( \text{accept}(F_{1,1}, \text{init}_1, t_1) = p_2 \) where \( \{p_1\} \) is the initial state of \( F_1 \). The flow execution scenario after interpreting the first event \( t_1 \) is \( \{(F_{1,1}, \{p_2\})\} \).

Next, the second \( t_2 \) is interpreted. This event is accepted by \( F_{1,1} \) as \( \text{accept}(F_{1,1}, p_2, t_2) = p_3 \). Next event \( t_1 \) activates another instance of flow \( F_1, F_{1,2} \). And event \( t_2 \) after that can be
accepted by $F_{1,2}$, resulting in the following flow execution scenario:

$$\{(F_{1,1}, \{p_3\}), (F_{1,2}, \{p_3\})\}.$$ 

For the fifth event $t_3$, it can be accepted by both $F_{1,1}$ and $F_{1,2}$. Therefore, two execution scenarios can be derived as showed below.

$$\{(F_{1,1}, \{p_4, p_5\}), (F_{1,2}, \{p_3\})\}$$

$$\{(F_{1,1}, \{p_3\}), (F_{1,2}, \{p_4, p_5\})\}.$$ 

After handing the following event $t_3$, the above two execution scenarios are reduced to the one as shown below.

$$\{(F_{1,1}, \{p_4, p_5\}), (F_{1,2}, \{p_4, p_5\})\}.$$ 

After processing the next event $t_4$, the two execution scenarios below can be derived:

$$\{(F_{1,1}, \{p_6, p_5\}), (F_{1,2}, \{p_4, p_5\})\}$$

$$\{(F_{1,1}, \{p_4, p_5\}), (F_{1,2}, \{p_6, p_5\})\}.$$ 

Next, processing the following event $t_5$ leads to execution scenarios derived from those shown above:

$$\{(F_{1,1}, \{p_6, p_7\}), (F_{1,2}, \{p_4, p_5\})\}$$

$$\{(F_{1,1}, \{p_4, p_7\}), (F_{1,2}, \{p_6, p_5\})\}$$

$$\{(F_{1,1}, \{p_6, p_5\}), (F_{1,2}, \{p_4, p_7\})\}$$

$$\{(F_{1,1}, \{p_4, p_5\}), (F_{1,2}, \{p_6, p_7\})\}.$$ 

Similarly, next event $t_5$ reduces the execution scenarios above to the following ones:

$$\{(F_{1,1}, \{p_6, p_7\}), (F_{1,2}, \{p_4, p_7\})\}$$

$$\{(F_{1,1}, \{p_4, p_7\}), (F_{1,2}, \{p_6, p_7\})\}.$$ 

(3.2)
Eventually, after handling the last event $t_4$ the execution scenario below is derived.

$$\{(F_{1,1}, \{p_6, p_7\}), (F_{1,2}, \{p_6, p_7\})\}$$

In this example all flow events are successful mapped and every flow scenario reached its end state. The result shows that two instances of the firmware loading flow are activated during the system run and finished correctly. While no error happens during the analysis process, debuggers can use this result to check if the numbers of flow instances are correct compared to the expected data extracted from verified simulation. This process involves checking types of flow specifications activated and numbers of instances of each flow. Moreover, depending on the correlation between the flows, together with extracted orders of flow instances, debugger can judge if the system functions correctly.

Now suppose that system generate a sequence of events same as the previous one in (3.1) except that the last event is $t_3$ instead of $t_4$. The new trace is showed below:

$$t_1 \ t_2 \ t_1 \ t_2 \ t_3 \ t_4 \ t_5 \ t_5 \ t_3 \ldots$$

The same execution scenario as in (3.2) are derived after the first nine elements are handled:

$$\{(F_{1,1}, \{p_6, p_7\}), (F_{1,2}, \{p_4, p_7\})\}$$

However, neither of these two existing scenarios can accept $t_3$. Furthermore, because no new flow instances can be created such that $t_3$ can be accepted in the initial states. Therefore $t_3$ is regarded as an inconsistent event.

When an inconsistent event is found, debuggers can make use of the current partially derived scenarios and the inconsistent flow event to identify possible causes and the potential
faulty components in the system. Based on this information, debuggers can select a new set of observable signals in order to better understand the activities around the suspicious components in a new SoC execution. The new observed traces can help debuggers better understand the problem, and may eventually lead to locating the root cause of the problem.

3.3 Dealing with Partial Observability

In SoC where the given system flow specification is implemented, a flow event is typically implemented as an event or a sequence of events on a set of hardware signals. Therefore, a mapping function that can translate a sequence of set of signal events to flow events is needed. However, an SoC usually contains tens of millions of signals, and it is impossible to observe every signals on the chip interface. As a result, the previous basic algorithm must account for the case where signal traces are obtained under partial observability.

In this section, the trace analysis algorithm is extended to deal with signal traces of partial observability. Hereafter, the term flow traces is used to refer to traces of flow events, and signal traces refers to traces of signal events observed from system execution.

3.3.1 Message Flow Guided Trace Abstraction

A signal event is defined as a state on or an assignment to a set of signals. In general, a signal trace of partial observability is a sequence of signal events such that the values of non-observable signals are unknown. In this case, all possible values of those signals need to be considered for every signal event during trace analysis. Thus we can say that one partially observed signal trace can be mapped to a set of fully observed signal traces.

Consider the following example for mapping individual signal events to flow events. Suppose there are three flow events: \(e_1\), \(e_2\), and \(e_3\), which are implemented in hardware by the signal events shown in the list below. We use Boolean expressions to represent signal events
for the discussion.

\[
e_1 : \quad abc \\
\]
\[
e_2 : \quad \overline{abc} \\
\]
\[
e_3 : \quad \overline{abc} \\
\]

In addition, assuming that only signals \( b \) and \( c \) are observable, we obtain the following trace:

\[
\rho = \{bc\} \{bc\} \{\overline{bc}\}
\]

Since \( a \) is not observable, both possible assignments to \( a \) need to be considered when these signal events are mapped to flow events.

The first and second signal events \( bc \), can be mapped to possible signal events with both values of \( a \) assigned: \( abc, \overline{abc} \). The first signal event \( abc \) can be mapped to \( e_1 \). While \( \overline{abc} \) can be mapped to \( e_2 \). Therefore, signal event \( bc \) with \( a \)'s value unknown can be mapped to \( \{e_1, e_2\} \). Similarly, the third signal event \( \overline{bc} \) can be mapped to \( a\overline{bc} \) and \( \overline{a\overline{bc}} \), respectively. In this case \( a\overline{bc} \) is mapped to \( e_3 \). On the other hand, \( \overline{a\overline{bc}} \) cannot be mapped to any flow event, therefore, this interpretation of signal \( a \) is invalid, and is ignored. Based on the above discussion, this signal trace \( \rho \) is abstracted to four possible flow traces:

\[
\{e_1, e_2\} \times \{e_1, e_2\} \times \{e_3\}.
\]

Next, we consider the more general case where a flow event is implemented by a sequence of signal events to model a transaction that takes a number of signal events to accomplish. For example, a flow event that represents a message sent from component \( A \) to component \( B \) following a handshake protocol consists of the following two steps:

(1) Component \( A \) sets the valid bit to 1 together with the command,

(2) Component \( B \) sets the acknowledgement signal to 1.
Algorithm 2: \( \text{Map}(\rho, h, \text{Flow\_Map}) \)

1. \( \text{Result} = \emptyset \)
2. \( \text{pref} = \epsilon \)
3. \( \text{foreach } i \in 0\ldots\min(\text{Max} - 1, |\rho| - 1 - h) \text{ do} \)
4. \( \quad \text{pref} = \rho[h, h + i] \)
5. \( \quad \text{foreach } (e, \sigma) \in \text{Flow\_Map} \text{ do} \)
6. \( \quad \quad \text{if } |\text{pref}| == |\sigma| \text{ then} \)
7. \( \quad \quad \quad \text{if } \sigma \Rightarrow \text{pref} \text{ then} \)
8. \( \quad \quad \quad \quad \text{Result} = \text{Result} \cup (e, h + i + 1) \)
9. \( \quad \quad \text{end} \)
10. \( \quad \text{end} \)
11. \( \text{end} \)
12. \( \text{return } \text{Result} \)

Mapping a sequence of signal events to a flow event is more precisely described in function \( \text{Map}(\rho, h, \text{Flow\_Map}) \), and its pseudocode is shown in Algorithm 2. This function takes the following inputs: signal trace \( \rho \), index of next signal event \( h \) in \( \rho \), and the mapping table \( \text{Flow\_Map} \) between flow events and signal events. It returns a set of pairs \((e, h')\) where \( h' \) is the position of the next signal event to be considered and \( e \) is a flow event mapped from the segment of \( \rho \) starting from signal event at index \( h \) to the signal event at \( h' - 1 \). Index \( i \) used in line 3 indicates the distance of last event of the prefix \( \text{pref} \) relative to the starting event at index \( h \), and \( \text{Max} \) is the length of the longest sequence of signal events that implement a flow event as defined in \( \text{Flow\_Map} \).

Once the mapping function is called, all segments of \( \rho \) of increasing length from 1 to \( \text{Max} \) or \(|\rho| - h \) (when the last signal event of \( \rho \) is reached) from the signal event at index \( h \) (expressed as \( \text{pref} = \rho[h, h + i] \)) are considered. All possible \( \text{pref}s \) are compared with \( \sigma \) in each instance \((e, \sigma)\) in \( \text{Flow\_Map} \) where \( e \) is a flow event and \( \sigma \) is the corresponding sequence of fully observed signal events. Due to the limitation of post-silicon validation, the
signal events in $pref$ are under partial observability. And for each partially observed signal event, a set of fully observed signal events can be obtained by considering all possible values of the unobservable signals. Therefore, a single $pref$ can represent a set of sequences of fully observed signal events. To compare the $pref$ with $\sigma$, we use $\sigma \Rightarrow pref$ to represent the successful mapping that $\sigma$ is included in the set of sequences of fully observed signal events represented by $pref$. More specifically, $\sigma \Rightarrow pref$ is defined below.

$$\forall i \in [0\ldots|\sigma|], \ \sigma[i] \Rightarrow pref[i]$$

To illustrate this algorithm, we assume that two flow events are implemented by two sequences of signal events as defined in the $Flow\_Map$ below.

$$Flow\_Map$$

$e_4: \{abc\} \{\bar{abc}\}$

$e_5: \{abc\} \{abc\} \{abc\} \{\bar{abc}\}$

Again, assume that $a$ is not observable, and suppose that an observed trace $\rho$ on signals $b$ and $c$ is obtained as shown in (3.3).

$$\rho = \{bc\} \{bc\} \{bc\} \{bc\} \quad (3.3)$$

The given mapping relationship $Flow\_Map$ between the flow events and the signal events shows the length of a signal trace for a flow event is either 2 or 4, hence the value of $Max$ is 4 in this example. The function takes $\rho, Flow\_Map$ and $h$ with value set to 0 as input.

Start with the first sequence of signal events $pref$ with $\rho[h, h + 0] = bc$, it cannot be matched to $e_4$ or $e_5$. Next, consider sequence $\rho[h, h + 1] = bc bc$, by looking up the mapping table $Flow\_Map$, this sequence can be mapped to $e_4$ where its corresponding signal events $\sigma = abc \bar{abc}$ is included in the set of sequences of fully observed signal events represented
by $\rho[h, h + 1]$. As a result, $(e_4, 2)$ is added to the Result. In the next step, sequence $pref = bc bc bc$ cannot be mapped to any flow event. Finally, a new $pref = bc bc bc bc$ is generated, and is mapped to $e_5$, therefore $(e_5, 4)$ is added to the Result. Subsequently, the function terminates and returns the set of pairs:

$$\{(e_4, 2), (e_5, 4)\}$$

As the pair $(e_5, 4)$ reaches the end of the signal trace $\rho$, there is no more signal event to be considered. For $(e_4, 2)$, the mapping function is applied to the same $\rho$ and $Flow\_Map$ with index $h$ changed to 2, and it returns the set of pairs $\{(e_4, 4)\}$. After combining the previous results, two flow traces are derived from $\rho$ as shown below.

$$\{e_4 e_4, e_5\}$$

3.3.2 Generalized Trace Interpretation

As shown in the previous section, more than one flow trace can be derived from a signal trace observed under partial observability. Each of the derived flow trace is very long and thus requires a lot of effort to interpret. Moreover, the number of flow traces can grow exponentially as the system complexity increases, and the number of observed signals remains limited. For the above reasons, the analysis time for the large number of flow traces can be impractical. To address this issue, our proposed work combines trace abstraction with trace interpretation into a new generalized algorithm as it is presented next in Algorithm 4.

This generalized algorithm takes two inputs: $F$ that contains a set of message flow specifications and a signal trace $\rho$. Instead of abstracting a set of flow traces from the signal trace $\rho$ and apply Algorithm 1 on each of the flow trace, this generalized algorithm tries to apply the analysis algorithm each time a new flow event is abstracted from the signal
Algorithm 3: Flow\_Analysis($\tilde{F}, \text{Scen}, e$)

1. $\text{Scen}' = \emptyset$
2. \textbf{foreach} scen $\in \text{Scen}$ \textbf{do}
    3. \quad \textbf{foreach} $(F_{i,j}, s_{i,j}) \in$ scen \textbf{do}
        4. \quad \quad $s'_{i,j} \leftarrow \text{accept}(F_{i,j}, s_{i,j}, e)$
        5. \quad \quad \textbf{if} $s'_{i,j} \neq \emptyset$ \textbf{then}
            6. \quad \quad \quad Let scen' be a copy of scen
            7. \quad \quad \quad scen' $\leftarrow$ scen' $- (F_{i,j}, s_{i,j}) \cup (F_{i,j}, s'_{i,j})$
            8. \quad \quad \quad Scen' $\leftarrow$ scen' $\cup$ Scen'
        9. \quad \quad \textbf{end}
    10. \textbf{end}
11. \textbf{foreach} $F_i \in \tilde{F}$ \textbf{do}
    12. \quad create a new instance $F_{i,j+1}$
    13. \quad $s'_{i,j+1} \leftarrow \text{accept}(F_{i,j+1}, \text{init}_{i,j+1}, e)$
    14. \quad \textbf{if} $s'_{i,j+1} \neq \emptyset$ \textbf{then}
            15. \quad \quad Let scen' be a copy of scen
            16. \quad \quad scen' $\leftarrow$ scen' $\cup (F_{i,j+1}, s'_{i,j+1})$
            17. \quad \quad Scen' $\leftarrow$ scen' $\cup$ Scen'
        18. \quad \textbf{end}
19. \textbf{end}
20. \textbf{return} Scen'

Trace $\rho$. This method can reduce the analysis time significantly by terminating the analysis whenever an inconsistent event is detected.

In the algorithm, variable $\text{Scens}$ holds a set of pairs $(\text{Scen}, h)$ where $\text{Scen}$ is a set of flow execution scenarios extracted from a segment of signal trace $\rho$ starting from the index 0 to index $h - 1$. The algorithm goes through each pair of $(\text{Scen}, h)$, and apply the mapping function from previous section to $\rho, h$ and $\text{Flow\_Map}$ to produce a set of pairs of signal events $e$ and $h'$ as its corresponding location of the next signal event to be considered. In
Algorithm 4: Generalized-Check-Compliance($\tilde{F}$, $\rho$)

1. $Scens = \{(\emptyset, 0)\}$
2. $Scens\_final = \emptyset$
3. while $Scens \neq \emptyset$ do
   4.     get $(Scen, h) \in Scens$
   5.     $flag = false$
   6.     $K \leftarrow Map(\rho, h, Flow\_Map)$
   7.     foreach $(e, h') \in K$ do
   8.         $Scens' \leftarrow Flow\_Analysis(\tilde{F}, Scen, e)$
         /* if $Scens = \emptyset$, $e$ is inconsistent with $Scen$ */
   9.         if $Scens' \neq \emptyset$ then
          10.             $flag = true$
          11.             if $h' = |\rho|$ then
          12.                 $Scens\_final = Scens\_final \cup (Scens', h')$
          13.             else
          14.                 $Scens = Scens \cup (Scens', h')$
          15.             end
          16.         end
   17.     end
   18.     if $flag == false$ then
   19.             $Scens\_d = Scens\_d \cup (Scen, h)$
   20.         end
   21.     $Scens = Scens - (Scen, h)$
22. end
23. if $Scen\_final = \emptyset$ then
24.     return $Scens\_d$
25. else
26.     return $Scen\_final$
27. end

the next step, function $Flow\_Analysis(\tilde{F}, Scen, e)$ (as shown in Algorithm 3) is applied to each $e$ and $Scen$, together with the flow specification set $\tilde{F}$ to produce an updated scenario
set $Scen'$. The updated $Scen'$ together with $h'$ as location of the next signal event to be considered is add to the $Scens$. For situations when $h' = |\rho|$, meaning that it has reached the end of the signal trace $\rho$, $(Scen', h')$ is add to $Scen_{final}$ instead of $Scens$ such that $Scen_{final}$ holds the set of pairs $(Scen, |\rho|)$ where $Scen$ is a set of flow execution scenarios that are extracted from the signal trace $\rho$. For debugging purpose, another instance $Scens_d$ is created to hold a set of pairs $(Scen, h)$ when one of the following two conditions holds:

1. When the returned set $K$ from function $Map(\rho, i, Flow\_Map)$ is empty, meaning that no flow events can be mapped from current signal events, or

2. When the returned set $K$ is not empty, but the the value of $flag$ is false. This means that non of the flow event in $K$ is consistent with the current flow execution scenario $Scen$.

When all pairs of $(Scen, h)$ in $Scens$ are considered, this algorithm terminates. Depends on the size of $Scen_{final}$, the following variables can be returned:

1. When $Scen_{final}$ is not empty, it means that there exists at least one flow execution scenario that is extracted from signal trace $\rho$, thus $Scen_{final}$ is returned.

2. When $Scen_{final}$ is empty, indicating that all flow events are inconsistent with its scenario set, thus $Scen_d$ is returned for debugging purpose.

3.3.3 Handling Missing Events

Another situation we consider in this section is when certain events are miss due to the trace collecting tool, and we presents a further revised algorithm for handling input traces with possible missing events.

We consider the general case that we do not know if or when there are any missing events that are discarded by the event collecting unit. Suppose an intermediate execution scenario
\{(F_{i,j}, \text{start}_{i,j}, \text{end}_{i,j}, s_{i,j}) \mid F_i \in \mathbf{F}\} \) is derived, and event \( t_i \) is the next in the trace to be analyzed. If \( t_i \) can be admitted by this execution scenario, the algorithm proceeds as normal. Otherwise, consider the following two possible situations.

- The algorithm checks if \( t_i \) is executable from \( s_{i,j} \) in \( F_i \). Event \( t_i \) is executable from state \( s_{i,j} \) in a flow specification \( F_i \) if there is a path in \( F_i \) from \( s_{i,j} \) ending at \( t_i \) such that executing all events on that path proceeding \( t_i \) leads to a state where \( t_i \) can also be executed. If \( t_i \) is executable from \( s_{i,j} \), the algorithm executes \( t_i \) from \( s_{i,j} \) leading to a state that is the same as the state by executing all events on the path from \( s_{i,j} \) to \( t_i \) including \( t_i \). Then, the execution scenario is updated accordingly.

- If an event \( t_i \) cannot be admitted to any flow instance, it is inconsistent. An inconsistent event may indicate an error in the system, or it may be due to missing events caused by the event output unit. Those dropped events may include the start events that could initiate the flows where \( t_i \) is specified, therefore causing flow instances that can admit \( t_i \) to be unobservable. In the presence of inconsistent events, the algorithm outputs those flows for more focused observation in the next debug run.

In summary, the revised trace analysis algorithm generates two pieces of information, possible flow execution scenarios, and a set of candidate flows for observation in the next run.

To illustrate the idea, consider the flow \( F_1 \) in Figure 3.4, and a trace shown below.

1 \( \langle \text{CPU}_0 : \text{Cache}_0 : \text{wr req} : 100 \rangle \)

2 \( \langle \text{CPU}_0 : \text{Cache}_0 : \text{wr req} : 160 \rangle \)

3 \( \langle \text{Cache}_1 : \text{Cache}_0 : \text{snp wr resp} : 100 \rangle \)

4 \( \langle \text{Cache}_0 : \text{Bus} : \text{wb resp} : 200 \rangle \)

\ldots
Figure 3.4: LPN formalization of a CPU write protocol.

The following flow execution scenario is derived after the first two events are processed.

\[
\{(F_{1,0}, \{p_2\}, 1, - , 100), \ (F_{1,1}, \{p_2\}, 2, - , 160)\} \tag{3.4}
\]

Event #3, \((\text{Cache}_1 : \text{Cache}_0 : \text{snp\_wr\_resp} : 100)\), cannot be admitted into either of the flow instances directly. Note that we do not know if there are missing events before event #3. The algorithm first checks if that event is executable from the current state of either flow instance. From Figure 3.4, it can be seen that event #3 is executable from \(\{p_2\}\) via event \((\text{Cache}_0 : \text{Cache}_1 : \text{snp\_wr\_req})\). Therefore, event #3 can be admitted to the flow instance
with address 100 indirectly. Processing event #3 in the above execution scenario results in the execution scenario as below.

\[ \{(F_{1,0}, \{p_4\}, 1, -, 100), \ (F_{1,1}, \{p_2\}, 2, -, 160)\} \]  

(3.5)

3.4 Complexity and Accuracy

Due to the limited observability, reconstructing system level executions from an observed silicon trace is an imprecise process. The large number of execution scenarios typically derived during the analysis would take large amounts of runtime and memory to process and to store, thus making it less efficient. This is referred to as the complexity problem of the trace analysis. After the analysis is done, a large number of derived execution scenarios make it difficult to understand the analysis results, thus being less helpful for debugging. Obviously, a single flow execution scenario derived at the end of the trace analysis provides much more precise information for debug than ten candidate flow execution scenarios. This is referred to as the accuracy problem of the trace analysis.

The contributing factors to the complexity and accuracy problems are explained below

- A signal event mapped to a set of flow events

  Due to the limited observability, a signal event of an observed silicon trace is often interpreted as a number of different flow events, which typically leads to derivation of a number of different execution scenarios. This situation is exacerbated by the fact that silicon traces are often very long, which could lead to excessively large numbers of possible execution scenarios derived during or at the end of the analysis.

- A flow event mapped to different temporal flow instances

  Temporal flow instances refer to the flow instances activated by the same component, e.g. read/write flows activated by CPU_0. If several temporal instances of some flows
are activated by a component, mapping flow events to those flow instances can be ambiguous. For example, suppose that an execution scenario includes two instances of the flow as shown in Figure 2 activated by CPU_0, one in state \{p2\}, and the other one in state \{p8\}. An instance of flow event (\texttt{Cache0 : CPU0 : wr.resp}) can be mapped to either flow instance leading to two new execution scenarios from the current one.

- A flow event mapped to flow instances activated by different components

This situation can happen when flow instances that share some common events are activated by different components. For example, suppose an execution scenario has two instances of the flow as shown in Figure 2, one activated by CPU_0 and the other one by CPU_1, and both are in state \{p6\}. A flow event (\texttt{Mem : Bus : rd.resp}) can be mapped to either one of these two instances, leading to two new execution scenarios derived from the current one.

The above issues can be mitigated by two potential approaches:

1. A better selection of signals to enhance post-silicon trace observability, which will be discussed in chapter 5.

2. Use of debugger’s insights during the analysis, which is briefly discussed in the next section.

3.5 Interactive Trace Analysis for More Efficient Debug

Post-silicon validation is performed by debuggers with deep knowledge about the system’s architecture and micro-architecture, and the test environment. Two key insights are (1) the maximal number of instances of a flow activated in the test environment, and (2) the mutual relationship between two flows. For example, the test environment may not allow multiple instances of firmware authentication to operate concurrently, or a flow involving audio and
Web browsing to initiate until the flows participating in boot are completed. Our framework permits incorporating such insights as constraints in trace analysis; flow execution scenarios that violate these constraints are ignored. These insights can lead to two advantages. First, they help to reduce the potentially large number of partial scenarios generated during the trace interpretation step, thus making the analysis more efficient. Second, they allow the debugger to quickly filter out uninteresting combinations of flows and focus on interesting interleavings.

If the precise knowledge of the system (micro-)architecture is hard to come by, this approach can be considered flexible as it allows a debugger to analyze the observed traces in a trial-and-error manner. For instance, the debugger might initially make a very restricted assumption on how the SoC executes a flow specification, and these assumptions can potentially lead to an empty set of flow execution scenarios. Depending on which of these assumptions triggered during the trace interpretation step, the debugger can study these assumptions more carefully, and relax some or all of them for the next run of analysis. This iteration can be repeated as many times as necessary until some results deemed meaningful are produced.

Alternatively, if all derived execution scenarios seem to be plausible, the implication that a debugger may draw from this result is that the failure may be independent of the flows being observed. Therefore, the testing environment can be adjusted in order for a different part or different behavior of the SoC to be observed.
Chapter 4: On-Chip Communication Tracing Infrastructure

The limited observability problem leads to two consequences. The large number of execution scenarios typically derived during the trace analysis would take long runtime and large amount of memory to process and to store, thus making it less efficient. This is referred to as the complexity problem of the trace analysis. After the analysis is done, a large number of derived execution scenarios make it difficult to understand the analysis results, thus being less helpful for debugging. Obviously, a single flow execution scenario derived at the end of the trace analysis provides much more precise information for debug than ten candidate flow execution scenarios. This is referred to as the accuracy problem of the trace analysis. This chapter proposes an on-chip monitoring infrastructure, communication tracing module (CTM) to address the problems above. The CTM consists of monitors attached to communication links to detect communication events, and an event output component to transport detected events off-chip for the trace analysis. Figure 4.1 shows the overview of the CTM architecture.

4.1 Communication Event Acquisition

A signal event denotes an assignment to a set of design signals. A flow event is an abstract construct used in flow specifications, and is typically implemented by a sequence of signal events. A monitor attached to a communication link reads signal events occurring

\[1\] This chapter was published in ISVLSI [59]: Cao, Yuting, Hernan Palombo, Sandip Ray, and Hao Zheng. "Enhancing Observability for Post-Silicon Debug with On-chip Communication Monitors." In 2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp. 602-607. IEEE, 2018. Permissions are included in Appendix B.
on that link, and generates an encoding for a flow event when a specific sequence of signal events have occurred. Figure 4.2 show the waveform where a master reads a slave through the AXI read protocol [60]. The blue lines (top 3 lines) are design signals as inputs to the attached monitor, while the red lines (bottom 2 lines) are the outputs of the monitor. From this example, it can be seen that one important function of a monitor is to compress a sequence of signal events across a potentially large number of cycles into a single cycle event, which is beneficial to reduce the bandwidth demand on the trace interface. Once an event is detected, a monitor can selectively encode information that is useful for the trace analysis including operation commands, addresses, etc. The basic idea of the above monitor can be naturally extended to different protocols such as the AXI write request and response.
Figure 4.2: An example of the AXI read event on a communication link, and the output of a monitor attached to that link.

It is possible that during debug only a subset of flows are observed either because they are necessary for understanding a particular use case or due to the limited observability of the trace interface. This requires to limit flow events to observe. Therefore, the monitors in our framework is designed such that they can be configured to output events that meet certain features. In general, each monitor has three configuration registers, \texttt{CCR}, \texttt{ACR\_BASE}, and \texttt{ACR\_OFFSET}, which are used to select events with particular commands or ranges of addresses. The width of \texttt{CCR} equals the number of unique commands that can be transferred on a link. An event is outputted by a monitor only when the corresponding bit to that event in \texttt{CCR} is set. Similarly, an event is outputted only when its address falls in the range as specified by \texttt{ACR\_BASE} and \texttt{ACR\_OFFSET}. This configurability can be extended to filter events if there is more diverse micro-architectural information available to encode events.

Furthermore, the monitors can also detect low level protocol errors timely and right on the spot. For example, the monitor shown in Figure 4.2 can output an unique \texttt{ERROR} event if \texttt{M\_Read\_Val} is set without \texttt{S\_Read\_Ready} set in the previous cycle.

4.2 Event Output

The detected events can be stored in the on-chip trace buffer, and offloaded from the chip at the end of system execution. However, the on-chip trace buffers can only store limited number of detected events due to the restriction on their capacities. As explained in the
previous section, when and where an error can happen are not know a priori, therefore, these limited events stored in the trace buffer may offer only limited debugability. This section describes an event output design that can output events via trace port on-the-fly, thus enabling system internal execution over an much extended period to be observed for off-chip analysis.

4.2.1 Parallel Output

The first approach is parallel where multiple links are traced simultaneously. Since the number of available trace signals are fixed, there is a trade-off between the number of links that can be traced simultaneously and the amount of information encoded for detected events on each link. More information encoded for events demands more trace signals, thus reducing the number of links that can be traced simultaneously. For example, suppose that a total of 100 trace signals are available, and there are 20 communication links to observe. If each event generated by the monitors for those links is encoded with 30 bits on average, then only 3 links can be traced simultaneously. On the other hand, if we wish to observe more communication links simultaneously, the number of bits for encoding events must be reduced, thus limiting the amount of information represented by events.

4.2.2 Interleaved Output

Since detected events by monitors are distributed over time relatively sparsely as illustrated in the last sub-section, an alternative approach is to interleave events detected on different links, and transport them off-chip serially. In this approach, monitors are connected to an event output unit like the one used in ARM CoreSight [61], which is shown in Figure 4.3. The events from monitors are routed through this output unit, merged into a sequence, and eventually output through the trace port. The biggest advantage of this
Figure 4.3: Detailed architecture of the output unit for the CTM.

approach is the very high observability in terms of the larger number of communication links to be traced and the higher amount of detailed information that can be encoded for events.

On the other hand, an issue with the interleaved approach is that the rate of events detected by monitors can exceed the peak bandwidth of the trace port from time to time. If that happens, some detected events cannot be transported off-chip. It can be viewed as another form of limited observability. Therefore, it would be desirable to reduce the number of events that have to be discarded.
The above issue can be addressed by using on-chip tracing event queues as shown in Figure 4.3. Those FIFOs can buffer detected events temporarily if they cannot be outputted right away. One FIFO is connected to the output of each monitor. On every cycle, the outputs of all monitors carrying detected events are stored into the corresponding FIFOs. At the same time, the event validity information of all monitors is collected into \texttt{Tr.Val}, and stored into a special FIFO \texttt{Tr.Valfifo}. This information is used to control how to output buffered events. The width of \texttt{Tr.Val} is equal to the number of monitors. \texttt{Tr.Val[i]=1} indicates that event output from monitor $M_i$ is valid. Otherwise, no valid output is from $M_i$. All the event FIFOs are connected to a $N$-to-1 selector where one event FIFO is routed to the trace port.

The control logic generates values for \texttt{sel} to control the selector based on the information stored in \texttt{Tr.Valfifo}. In the initial state, it asserts \texttt{Read_Tr.Val} to read the head of \texttt{Tr.Valfifo} into \texttt{Tr.Status}. If it contains some bits of 1, the control unit first determines the smallest index \texttt{i} such that \texttt{Tr.Status[i]=1}. This can be done by a priority encoder. Next, the event FIFO for monitor $M_i$ is connected to the trace port, and the event at its head is outputted. Then, \texttt{Tr.Status[i]} is reset to 0, and the control logic repeats the above step if there is a larger index \texttt{i} such that \texttt{Tr.Status[i]=1}. Otherwise, it returns to the initial state and read the next data from \texttt{Tr.Valfifo}. When \texttt{Tr.Status} is 0, \texttt{sel} is set to a special value $X$ to disable the selector. The control flow diagram for the control unit is at the bottom right corner in Figure 4.3.

We illustrate the operations of the event output unit with a simple example. Suppose that there are three monitors $M_0$, $M_1$ and $M_2$ connected to the event output unit. Their validity information is collected as a 3 bit \texttt{Tr.Val} and stored at the tail of \texttt{Tr.Status fifo} on each cycle. Table 4.1 shows how the output unit outputs the events from these monitors off-chip. In the columns under $M_0$, $M_1$ and $M_2$, a ✓ means its generated event is valid, while ✓ indicates it is not valid.
Table 4.1: Operations of the event output unit for 3 monitors over 5 cycles.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>$M_2$</th>
<th>$M_1$</th>
<th>$M_0$</th>
<th>Tr_Val</th>
<th>Read_Tr_Val</th>
<th>Tr_Status</th>
<th>Sel</th>
<th>Selector Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>101</td>
<td>1</td>
<td>000</td>
<td>X</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>not stored</td>
<td>010</td>
<td>0</td>
<td>101</td>
<td>0</td>
<td>$M_0$</td>
</tr>
<tr>
<td>3</td>
<td>not stored</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>100</td>
<td>2</td>
<td>0</td>
<td>$M_2$</td>
</tr>
<tr>
<td>4</td>
<td>not stored</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>010</td>
<td>1</td>
<td>0</td>
<td>$M_1$</td>
</tr>
<tr>
<td>5</td>
<td>not stored</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>000</td>
<td>X</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

In cycle 1, outputs from $M_0$ and $M_2$ are valid. They and $\text{Tr}\_\text{Val}=101$ are stored into their corresponding FIFOs, respectively. There is nothing to output, therefore $\text{Sel}$ is set to $X$. $\text{Read}\_\text{Tr}\_\text{Val}$ is set to 1 to read $\text{Tr}\_\text{Val}$ fifo. In cycle 2, similarly, outputs of all monitors and their validity are stored in the FIFOs. $\text{Tr}\_\text{Status}$ is used to compute $\text{Sel}$, which is 0 in this case. Therefore, the selector is directed to output the event from $M_0$ captured in cycle 1, as indicated in column of Selector Output. $\text{Tr}\_\text{Status}[0]$ is reset to 0 before the next cycle. Signal $\text{Read}\_\text{Tr}\_\text{Val}$ is reset to 0 when $\text{Tr}\_\text{Status}$ contains more than one bit of 1. In cycle 3, the event from $M_2$ captured in cycle 1 is outputted by the selector, and $\text{Read}\_\text{Tr}\_\text{Val}$ is asserted to read next validity data from $\text{Tr}\_\text{Val}$ fifo to prepare for the next cycle.

While the interleaved approach reduces the number of trace signals needed to output detected events, the event output unit can potentially introduce large area overhead. The area overhead are mainly due to the use of FIFOs to buffer events. Larger FIFOs can reduce the chance of events to be discarded, but increases area overhead. The optimal FIFO sizes are typically determined by the design of the interconnect network and the rates at which various blocks initiate system flows. In this work, when overflow happens to any FIFOs, new incoming events are simply discarded.
4.2.3 Priority Output

As indicated in the previous section, the frequency of event dropping is roughly affected by the gap between the rate of events generated by all monitors and the bandwidth of the trace port. The rate of event generation is determined by the design microarchitecture, test programs used during debug, and the number of flows and flow events selected for observation. One simple technique to reduce the frequency of event dropping is large buffers for events waiting for output, but it leads to a large area overhead. Due to the limitation on the buffer capacities, it is inevitable that some detected events will be dropped.

To address that problem, this section presents an alternative technique with a goal of reducing the impact of event droppings on the accuracy and efficiency of the trace analysis. As some events may be more important to the trace analysis than the other, it is preferable to drop events that are less important. During an execution run, the event output unit can be configured to enter the *priority output mode* under certain conditions where events with higher priorities are outputted first.

Section 5.2 discusses how to rank flow events, and their associated communication links. Such ranking information is used to configure the event output unit where events on the lower ranked links are dropped first when event buffers overflow. More specifically, a higher ranked link is associated with a threshold. During an execution run, events from lower ranked links can be outputted only when the numbers of events stored in the event queues for higher ranked links are lower than their thresholds. This technique can reduce droppings of more important events with possibly more frequent droppings of less important ones.

The priority output mode is controlled on-the-fly during system execution by an output configuration register (OCR), which has the number of bits equal to the number of monitors. This register is used as a mask for the output from *Event Validity Queue*. In the normal mode, all bits in OCR are set to 1. Suppose that the threshold for link *i* is crossed at some point, the output unit enters the priority output mode by setting the *jth* bit of OCR to 0 for
each link \( j \) that is ranked lower than link \( i \). This would effectively block events from the \( j \)th monitor from being output.

4.3 Event Encoding

Events can be encoded with information transferred over communication links at different levels of detail. In general, more bits are required to encode information at higher levels of detail. In the last section, two alternative event output approaches are discussed. Different representations of events are used for different approaches.

In the parallel output approach, multiple communication links are traced simultaneously, events representations are customized with respect to the specific protocols of different links. The representation below shows all the fields used in all events.

\[
\langle \text{Val, Cmd, Tag, Sid, Addr} \rangle
\]

The meanings of the message fields are defined below.

- **Val** indicates the validity of a detected event.
- **Cmd** carries operations to be performed by the target block. For the AXI protocol, there are separate links to support read/write request and response operations, this field is not needed.
- **Tag** is used to identify the original sources of events from different blocks that go to the same destination. For example, in Figure 6.1, Tag is needed for event \texttt{wr_req} from Bus to Memory in response to \texttt{wr_req} from either CPU.
- **Sid** is a unique number representing sequencing information associated with events initiated by a component that supports out-of-order execution.
• **Addr** carries the memory address at the target block where **Cmd** is applied. If the observability limitation does not allow full address information to be encoded, it can be abstracted with two bits to represent three states: *same as previous one*, *sequential*, and *others*, as described in [38].

Note that not all fields are used to represent events of all links. The sizes of events on different links may be different. Additionally, monitors can be configured to include only some selected fields to meet debug needs while satisfying observability constraints.

In the interleaved output approach, the trace port is shared among all links. As a result, a standard format as shown below is used for all different events.

\[(\text{Val}, \text{MasterID}, \text{SlaveID}, \text{Cmd}, \text{Tag}, \text{Sid}, \text{Addr}, \text{Step})\]  \hspace{1cm} (4.1)

where

- **MasterID** and **SlaveID** encode IDs of the sender and receiver of an event. The number of bits required for these two fields are determined by the number of masters and slaves in an SoC design.

- **Cmd** has a fixed number of bits for all events. Its width is determined by the largest number of events that any link can transfer.

- **Val**, **Tag**, **Sid**, and **Addr** remain the same.

- **Step**, which is only 1-bit, indicates the ordering of an event relative to its immediate predecessor. If this field is asserted, it indicates that the current event being outputted is detected after its immediate predecessor. Otherwise, the current event and its immediate predecessor are detected at the same time.

In the event output unit as shown in Figure 4.3, when an event is pulled out of its FIFO, it is converted to the above standard format before it enters the selector. Similarly, the
fields and their sizes in the above standard format can be reduced to meet the observability constraint.
Chapter 5: Communication-Centric Observability Selection

As indicated in the Section 4.2, limited bandwidth of trace ports may cause some communication events not to be observed. The frequency of event dropping is roughly affected by the gap between the rate of events generated by all monitors and the bandwidth of trace ports. The factors affecting the rate of communication event generation include the design micro-architecture, test programs used during debug, and the number of flows and flow events selected for observation. One simple technique to reduce the frequency of event dropping is large buffers for events waiting for output, but it leads to a large area overhead.

An important observation exploited in this dissertation is that some events are more important for understanding flow executions than others. If the less important events are not observed, then the whole trace port bandwidth can be dedicated to observing the more important events. Furthermore, not observing certain events may eliminate the need of observing certain communication links altogether. In that case, the corresponding monitors can be disabled, and the capacities of the associated event queues can be re-allocated to the queues for the links under observation. By increasing the queue capacities for the links under observation, the event droppings can be effectively reduced.

This section presents a number of communication event selection methods aiming to increase coverage metrics that are relevant to various debug objectives. These methods are optimized for an on-chip communication event tracing infrastructure that can be typically found in modern SoC designs. When such infrastructure is not available, we also provide

---

This chapter was published in ISQED [62]: Cao, Yuting, Hao Zheng, and Sandip Ray. "A Communication-Centric Observability Selection for Post-Silicon System-on-Chip Integration Debug." In 20th International Symposium on Quality Electronic Design (ISQED), pp. 278-283. IEEE, 2019. Permission is included in Appendix B.
schemes to select specific bits for the set of selected flow events. The main contributions of this part of work include the following.

- New coverage metrics are proposed for evaluating relevance and comprehensiveness of information captured on observed traces with respect to system-level communication protocols. The traditional metric, state restoration ratio (SRR), is not applicable for SoC integration debug.

- Communication event selection methods are driven by the proposed coverage metrics optimized for the real-time tracing infrastructure so that observed traces only capture the most relevant communication events under limited observability.

- Communication bit selection methods guided by the previously selected event selection.

5.1 Selecting Flows to Observe

A critical activity performed in post-silicon stage is to validate application usage scenarios. In this activity, individual target usage scenarios, e.g., for a smartphone, playing videos or surfing the Web, while receiving a phone call, are exercised, while possible failures, e.g., hangs, crashes, deadlocks, overflows, etc., are monitored. Usage scenario validation forms a key part of SoC integration validation. Each usage scenario usually involves interleaved execution of several flows among IPs in the SoC design, e.g., a usage scenario that entails receiving a phone call in a smartphone when the phone is asleep involves flows among the antenna, power management unit, CPU, etc. Therefore, only the flow events of the involved flows in a usage scenarios are typically observed. As explained in the previous section, observing a restricted subset of events can reduce incidents of events being dropped.
5.2 Flow Execution Coverage Metrics

In this section, we consider the problem of characterizing the importance and relevance of flow events by defining two coverage metrics for different debug purposes.

The first metric is flow instance coverage (FIC), which is defined below.

\[
FIC = \frac{I}{N}
\]  \hspace{1cm} (5.1)

where \( I \) is the number of flow instances observed, and \( N \) is the total number of flow instances executed. FIC defines a fraction of the number of flow instances actually observed versus the total number of flow instances executed. We say that a flow instance is observed in a trace if any event of that flow instance is observed in the trace. Note that the parameter \( N \) is not essential when this metric is applied to evaluate different observabilities as all of them are evaluated assuming the same \( N \).

The purpose of FIC is to offer a metric to evaluate different observabilities to support a coarse-grained global view of system execution. In this coarse-grained global view, we are interested in all flow instances executed in the entire course of a debug run, instead of detailed execution of individual flow instances. It may provide valuable information about anomaly behavior in system execution, \textit{e.g.}, an unusually high number of wakeup calls to a CPU from the power management unit. In this case, we want to select an observability that maximize FIC. Obviously, one observability is better than another one if its FIC derived from an observed trace is higher.
As explained in the introduction, many errors are due to intricate interleaved execution of flows, e.g., a firmware execution flow executed before a firmware authenticate flow completes, signaling a security breach. Therefore, the coarse-grained global view obtained by only observing flow instances is inadequate. Observed traces must capture sufficient information to allow interleaved execution of different flow instances to be extracted. The interleaving relations between two flow instances $F_{i,j}$ and $F_{x,y}$ are shown in Figure 5.1. In this figure, the length of arrows shows the duration of a flow instance execution, while the arrows at both ends indicate the time when a flow instance is initiated and when it completes. Figure 5.1 shows three possible interleavings: (1) $F_{i,j}$ starts before $F_{x,y}$ starts, and it completes after $F_{x,y}$ completes; (2) the initiation and completion of $F_{i,j}$ occur earlier than the initiation and completion of $F_{x,y}$; (3) $F_{i,j}$ completes before $F_{x,y}$ starts. An observability needs to be selected in order to support such interleavings to be captured on the observed traces. To evaluate different observabilities, we define the complete execution coverage (CEC) metric as below.

$$CEC = \frac{C}{N}$$

where $C$ is the number of complete flow instances extracted from an observed trace, and similarly $N$ is the total number of flow instances executed. A complete flow instance is observed if both its start and end events are found in the observed trace.

5.3 Coverage Driven Event Selection

This section presents observability selection methods driven by coverage metrics described in the previous section. The inputs to a selection method are a set of flows to observe and a coverage metric, and the output is a subset of flow events for observability that maximize the coverage metric.
To select an observability targeting coverage metric FIC, it is necessary to select a subset of flow events that cover all flows under observation such that an event in an observed trace can uniquely identify a flow instance. In practice, most SoC designs include architectural support for tagging, which allows uniquely identifying different flow instances from observing properly tagged events. Because of the unique correspondence between flow instances and observed events, all flow events are FIC-equivalent. Therefore, we aim to select a subset of events that maximizes the FIC.

Given a set of flows to observe, there can be many different selections of events of those flows. In order for the observed traces on the selected events to have high FIC, the losses of the selected events must be low. Recall the hypothesis presented in the previous section indicating that the losses of events can be reduced if the number of links under observation is reduced. By this hypothesis, for two sets of selected events, if the number of links to observe for one set of selected events is smaller than that of another set, then the former is preferred.

Next, we consider observability selection targeting the coverage metric CEC. In order for observed traces to have high CEC, the start and end events of all flow instances should be observable. Therefore, the start and end events of all flows to observe must be selected.

To facilitate more effective debug, it is necessary to know additional information beyond the initiation and completion of each flow instance. More specifically, it would be useful to know which path in the flow is followed when an instance of that flow is executed. Consider the flow in Figure 3.4 for an example. It has three possible execution paths. Observing only the start and end events (labelings of $t_1$ and $t_{10}$) is not sufficient to tell which execution path is actually followed. Given an observed trace shown below,

$$(\text{CPU}_x: \text{Cache}_x: \text{wr}_\text{req}), (\text{Cache}_x: \text{CPU}_x: \text{wr}_\text{resp}), \ldots$$

we are not able to confirm whether it is a result of executing that flow following the rightmost path ($p_1, t_1, p_2, t_{10}, p_9$) or one of the other two paths with all the intermediate events not observed. To obtain the information on paths following by a flow execution, some unique
event from each path needs to be selected. Consider the same flow example. Either one of \( \{ t_2, t_3 \} \) needs to be selected in order to identify the leftmost or middle path. Moreover, one event from \( \{ t_4, t_5, t_6, t_7 \} \) needs to be selected to identify the middle path. As the above illustration shows, there are different ways to select additional events to observe for detailed executions of a set of flows. Similarly, these different selections are evaluated based on the number of links that need to be observed for the selected events.

**5.4 Bit Selection**

This section provides basic guidelines for bit level selection when the on-chip monitoring infrastructure is absent. The bit level selection takes as inputs the event selection produced in the previous step, together with implementation details of the CUD, and generates a set of candidate trace signals that implement the selected events. The ultimate goal of the bit level selection is to produce a reduced set candidate trace signals optimized for the trace analysis approach. Since the bit level selection depends on implementation specifics, this section can only discuss some general guidelines and tradeoffs. Note that flow specifications are typically independent of memory address and data information. Therefore, the address and data bits included in event implementations can be generally ignored.

Signals that implement the \texttt{Cmd} field of flow events are selected based on their respective distinguishing power. Given a set of flow events \( E \) and a set of signals \( W \) that implement \( E \), the distinguishing power of \( W_i \subseteq W \) is defined by the amount of events in \( E \) that can be partitioned based on value of \( W_i \). Notice that \( W_i \) defines a partition of \( E \). A finer partition means higher distinguishing power. For example, suppose two flow events on link
(CPU_0, Cache_0) are implemented by eight signals \( b_7...b_0 \) with the following encodings.

\[
\begin{align*}
  (CPU_0 : \text{Cache}_0 : \text{wr}_\text{req}) & \quad 0100 \ 0000 \quad (5.3) \\
  (CPU_0 : \text{Cache}_0 : \text{rd}_\text{req}) & \quad 1000 \ 0000 \quad (5.4)
\end{align*}
\]

Under these encodings, signals \( b_5...b_0 \) have zero distinguishing power. \( b_7 \) and \( b_6 \) have the equal power, therefore selecting either one would be fine. Selecting signals with high distinguishing power helps to address issue 1 as discussed in Section 3.4.

RTL models may contain additional implementation information that can help to address issue 2 and 3. For example, memory operations may be executed out-of-order. In this case, CPUs usually assign unique sequence IDs to flow instances to maintain data and control dependency in the original programs. If sequence IDs are available, selecting signals implementing them can help address issue 2.

If the on-chip interconnect needs to handle events from different components in a system, the events are usually assigned with tags to identify their originating components. Selecting tags can affect how events are selected. Refer to Figure 3.4 for the following discussion.

- If unique events such as \( t_4 \) or \( t_7 \) are selected, observing tags is not needed.
- Shared events \( t_5 \) or \( t_6 \) are selected along with tags.

For option 2, tags can help to map events to the flow instances with the same tags during the trace analysis, thus addressing issue 3. Even though additional signals for tags are selected, the total number of events may be smaller if the shared events are used in many different flows, therefore resulting in reduced signals for observation overall.

The following discussion illustrates yet another example of how implementation information can allow different events to be selected. Refer to Figure 3.4. That flow contains two branching places, \( p_2 \) and \( p_4 \). When a flow instance reaches \( p_2 \), which branch to take next
depends on whether the cache operation is hit or miss. Similarly, which branch to take at $p_4$
depends on whether the cache snoop operation is hit or miss. If these two status signals are
available and included for observation, there is no need to select branch events. Observing
start/end events plus those status signals are sufficient to identify branches followed by a
flow instance during system execution.
Chapter 6: Experiments

This chapter demonstrates the usage of our proposed framework on two different models. A more detailed RTL model of a similar SoC is constructed to test the efficiency and effectiveness of our framework with different perspectives.

6.1 A Multicore SoC Model

In this section, we construct a cycle-accurate RTL model for a similar SoC to allow more detailed trace signal selection. For this model, the trace information is collected at the bit level, then an extra translation step is required.

6.1.1 Model Implementation

Due to the novelty of our work, we could not find an existing model with a well-documented flow specification. As a result, we have to build this model from scratch. This multi-core SoC prototype implements some common industrial message flows including

![RTL model structure](Figure 6.1: The RTL model structure)
cache coherence and power management. It is a cycle- and pin-accurate RTL model written in VHDL. Even though this model is simple compared to SoC designs used in commercial applications, it is much more sophisticated than the gate-level benchmark suites typically considered as targets for post-silicon analysis [63, 49, 41].

Since the proposed trace analysis approach is communication-centric, the focus of this model is the implementation of message flows. This model consists of two CPU models, each with its own 4KB Data Cache. The Data Caches are connected to a 256KB Memory through a bus model. The CPUs are treated as a test environment where software programs are simulated in VHDL to trigger various protocols. Therefore, there is no instruction cache as no instructions are involved when the CPUs are simulated. The peripheral blocks, GFX, PMU, Audio, etc, are also described as abstract models that generate events to initiate flows or to respond to incoming requests. More details of some message flows implemented in our model can be found in Appendices. They include downstream read/write protocols for each CPU, upstream read/write for the peripheral blocks, and system power management protocols which are abstracted from real industrial protocols.

These message flows are supported by inter-block communication protocols based on the ARM AXI4-lite [60]. A total of 16 flows are implemented for this prototype. A flow event is generated from a source and consumed by a destination by messages transmitted over that link. In our model, each message is organized as follows.

\[
\langle \text{Val}(1), \text{Cmd}(8), \text{Tag}(8), \text{Sid}(8), \text{Addr}(32), \text{Data}(32) \rangle
\]

The numbers following the individual fields indicate their respective widths. Note that not all fields are used on all links. This model has over four thousand single bit signals.

To collect communication activities of the system, values of some selected signals or observation are collected and outputted to a trace file of an internal format on each rising
clock edge. There are 32 communication links implemented in this model, each of which is implemented by several signals to ensure the correct communication behaviors of the SoC.

6.1.2 Test SetUp

Two separate SoC test settings were established to test different features proposed in this dissertation.

• **Test 1**

The prototype is simulated in a random test environment where CPUs, GFX, and other peripheral blocks are programmed to randomly select a flow to initiate in each clock cycle. The contents of \( \text{Cmd}, \text{Addr}, \text{and Data} \) in each activated flow are set randomly. Additionally, CPUs can activate power management protocols non-deterministically. Each of these blocks activates a total of 100 flow instances during the entire simulation.

• **Test 2**

We hard code a simple software that performs Peterson’s Algorithm with two threads, one for each CPU. The pseudocode is shown in Figure 6.2. This algorithm contains four shared variables: \( \text{flag0}, \text{flag1}, \text{turn}, \) and \( \text{shared} \). A CPU that wants to enter the critical section has to wait until the \( \text{flag} \) of the other CPU or \( \text{turn} \) get the desired values. Whenever a CPU enters the critical section, it increments the variable \( \text{shared} \) by one. By running this algorithm \( N \) times, the final value of \( \text{shared} \) should be \( 2N \), as both CPUs increment variable \( \text{shared} \) \( N \) times. During this test, one bug was found and fixed. Details of this bug case are discussed in the next section.

6.2 Experiments with the Communication Tracing Module

A total of 32 monitors are inserted into the cycle- and pin-accurate RTL model, as its structure shown in Figure 6.3. Notice that there can be multiple links between a pair
Figure 6.2: Peterson’s algorithm on two CPUs

of components. Table 6.1 shows the general effects of the CTM when different offloading approaches are selected, row 2 shows the numbers of bit used for each approach. In the table, row 3 and 4 show the peak count of flow execution scenarios encountered during the reconstruction process which is used to measure the complexity and the final count of flow execution scenarios derived at the end of the reconstruction process that is used to measure the accuracy, respectively. Row 5 shows the maximal number of flow instances activated by various components identified by the trace analysis. The last two rows show the runtime and memory usages by the trace analysis for different offloading approaches. The results from analyzing the simulation traces of those five runs are shown in column 2 – 6, respectively, in Table 6.1.

By comparing results in columns 2–4 in the table, it can be seen that using the monitoring infrastructure allows the same analysis result about system internal execution to be derived
as what can be derived with the full observability. More importantly, that is achieved by requiring *significantly reduced number of trace signals*. The trace analysis with the monitoring infrastructure achieves the same complexity and accuracy as those achieved with the full observability.

The last two columns show the trace analysis results without using the monitoring infrastructure. We assume that 36 signals are available for tracing as in the third run. Due to this restriction, we can select only a small number of links where the signal events can be observed accurately. The results from this run are shown in Column 5. Even though only one flow execution scenario is derived at the end, the limited number of signal events selected for observation allow much less number of related flow instances to be derived than what can be derived when the monitoring infrastructure is used. When we try to observe more links, we are forced to allocate fewer trace signals for each event on those links. This causes
Table 6.1: Runtime results from analyzing traces obtained in different approaches.

<table>
<thead>
<tr>
<th></th>
<th>Full</th>
<th>Parallel</th>
<th>Interleaved</th>
<th>SS1</th>
<th>SS2</th>
</tr>
</thead>
<tbody>
<tr>
<td># Bits</td>
<td>870</td>
<td>720</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td># scen(peak)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>282k</td>
</tr>
<tr>
<td># scen(final)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>#flows</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>Time</td>
<td>1.391</td>
<td>1.237</td>
<td>1.218</td>
<td>0.714</td>
<td>600</td>
</tr>
<tr>
<td>Mem</td>
<td>1.068</td>
<td>1.017</td>
<td>1.028</td>
<td>0.608</td>
<td>5GB+</td>
</tr>
</tbody>
</table>

ambiguity to the interpretations of the observed signal events. As a result, an excessively large number of potential flow execution scenarios are derived as shown in Column 6. After 10 minutes, the trace analysis has to be terminated due to memory usage explosion. These results show that under the limited observability, the complexity and accuracy of the trace analysis would suffer significantly if the monitoring infrastructure is not used.

We measure the hardware area overhead of the monitoring infrastructure by synthesizing the SoC model to the Xilinx Zynq FPGA xc7z020ckg484-1 using Vivado 2017.2. The synthesis results are shown in Table 6.2. The area overhead is measured by the FPGA resources used including LUTs, FFs, block RAMs (BRAMs), etc.

The rows for Parallel and Interleaved show the additional resources required to implement monitors with or without event output unit to the resources used on the previous row. For example, the parallel approach uses additional 1283 cells to implement all 32 monitors, and the interleaved approach requires extra 283 cells to implement the event output unit. From the table, other than the big jump in BRAM usage, demand on logic resource is small to implement the monitoring infrastructure. The BRAMs are used to implement the FIFOs in Figure 4.1. Since the size of the BRAMs is fixed, each FIFO only uses a small capacity of a BRAM. In practice, SoCs often have embedded trace buffers, which can be used for those FIFOs.
6.3 Experiments with Observability Selections

6.3.1 Flow Event Selection

Table 6.3 shows the experimental results when different flows are selected for observation. The capacity of all the queues is set to 8. Under the columns FIC and CEC, the numbers $A/B$ represent the ratios as defined in Section 5.2. The equivalent fractional numbers for $A/B$ are enclosed in parentheses. The second row shows the results when all 500 flow instances are observed. The third row shows the results from observing only flows initiated by CPU0 or CPU1. The last row shows the results from observing only the flows initiated by CPU0. From the table, it can be seen that as the number of flows under observation decreases, the number of events that need to be observed for FIC and CEC decreases. This leads to reduced losses of events, which is reflected in the higher FIC and CEC values from row two to row four. Particularly, in the last row, all executed instances of the flows under observation can be precisely inferred.

In the second experiment, the impacts of different queue capacities are evaluated. All flows and all flow events are observed. Event dropping happens only when a queue becomes full. In these experiments, the capacities of event queues are increased gradually. This increase in queue capacity can simulate the situation where the number of links to observe

Table 6.2: Area overhead of the monitoring infrastructure.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Cells</th>
<th>LUTs</th>
<th>FFs</th>
<th>Muxs</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>59154</td>
<td>24395</td>
<td>25962</td>
<td>3125</td>
<td>1</td>
</tr>
<tr>
<td>Parallel</td>
<td>+1283</td>
<td>+8</td>
<td>+1251</td>
<td>+15</td>
<td>+0</td>
</tr>
<tr>
<td>Interleaved</td>
<td>+232</td>
<td>+92</td>
<td>+126</td>
<td>-6</td>
<td>+32</td>
</tr>
</tbody>
</table>

Table 6.3: Results with different flows under observation.

<table>
<thead>
<tr>
<th>Scope</th>
<th>FIC</th>
<th>CEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>470/500 (0.94)</td>
<td>101/500 (0.202)</td>
</tr>
<tr>
<td>CPU</td>
<td>192/200 (0.96)</td>
<td>103/200 (0.515)</td>
</tr>
<tr>
<td>CPU0</td>
<td>100/100 (1)</td>
<td>100/100 (1)</td>
</tr>
</tbody>
</table>
is reduced and the queues of the non-observable links are re-allocated to the queues of links under observation. The results are shown in Table 6.4. It is noticeable that an increase in the size of queues leads to a significant improvement in FIC and CEC values of observed traces. These results validate the hypothesis described in Section 4.2.3.

Table 6.5 compares the result of the proposed flow event selection algorithm with another system-level flow guided selection approach proposed in [64]. We generate two sets of selections guided by $FIC$ and $CEC$ individually (SEL1 and SEL2) and compare their results with selection SEL3 generated by [64]. The results are evaluated by the previously mentioned two metrics and the numbers of links used. The second row NO-SELECTION represents the analysis result when no selection method is applied and all flow events on the 32 links are observed.

The SEL1 on the third row is generated by our method that is primarily guided by the $FIC$. In the beginning, several sets of flow events with the optimal $FIC$ effects (covering all flows) are generated, and within them, we select the set with the minimal numbers of links required, that is 8 in this situation. The result shows that the $FIC$ is improved to its maximum value 1, while on the other hand, $CEC$ is reduced to 0. This is expected as
CEC is not considered for this selection. Then for the second selection SEL2, we consider the CEC only and its result is shown in the fourth row. It first selects all 32 initiating and terminating events of all flows to enhance CEC. Because several flow events are transferred on the same link (for example, both (CPU_0:Cache_0:wr_req) and (CPU_0:Cache_0:rd_req) are transferred on the same link), these events takes 12 links in total. Moreover, SEL2 selects four flow events that indicate the path of a flow, as discussed in Section 5.3, occupying 4 links. Compared to the NO-SELECTION, The result of SEL2 takes only 16 links and it shows significant improvement on the CEC (almost double) as more start and end events are observed. Consequently, more details regarding the interleaving relationship of fired flow instances are revealed in SEL2. This improvement comes in the cost of fewer flow instances being observed shown by the decrease in FIC value.

The fifth row SEL3 shows the result of using flow selection method from [64]. [64] proposes to rank each flow event by their Frequency Coverage (FC) value in descending order and apply greedy algorithm to select the optimal set of flow events. The FC considers the fact that some flow events are shared by multiple flows thus are always preferred. The concept of FC is very similar to FIC as it enforces the maximum number of flows being covered for a set of flow events. However, the FC in [64] is different in two aspects: (1) it does not consider the number of links used for such selection. Consequently, the capacity improvement may not be as significant as our method where we always select the set with the minimal number of links; (2) Because FC considers each flow event individually, it is possible the selected combination of events achieves high coverage only on certain flows. On the other hand, our method ensures that the selected flow event set covers all flows.

We applied the algorithm in [64] and selected the top 16 (half of the total link number) flow events with the highest FC. The result of SEL3, however, does not show any improvement compared to NO-SELECTION, the FIC actually reduced from 470 to 466 due to the two factors mentioned above. It is also to be noted that for SEL3, each of the se-
lected flow occupies one link individually, taking 16 links in total, which double the number of links selected by SEL1. As a result, the $FIC$ of SEL3 is comparably worse than both SEL1 and NO-SELECTION. On the other hand, none of the initiation and termination events are selected due to their uniqueness to their belonging flows (not shared by any other flows), leading to $CEC$ being 0. This is expected as this algorithm does not consider special meanings of such critical events.

6.3.2 Bit Selection

In this experiments, different selections of trace signals are produced and their impacts on the complexity and accuracy of the trace analysis approach are evaluated. The list below explains the selections at the system level while information on the bit level selection is given in Table 6.6.

- **S1**: The events of all flow specifications and all signals implementing each event are selected. This selection offers a full observation and provides a baseline for comparing with other selections.

- **S2**: The start and end events of all protocols are selected. Furthermore, for each branch in each flow, one unique event is selected.

- **S3**: The start and end events of all protocols are selected. Furthermore, for each branch in each flow, a highly shared event is selected.

- **S4**: The start and end events of all protocols are selected. Instead of selecting events for branches in each flow, signals whose states control the flow branching are selected.

At the bit level, the $Addr$ and $Data$ fields are not considered. On the other hand, the $Val$ bit is always selected so that valid messages can be identified from observed traces.
Table 6.6: Runtime results of trace analysis with different trace signal selections.

<table>
<thead>
<tr>
<th>Selection</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>US</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cmd</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Tag</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Sid</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td># Bits</td>
<td>870</td>
<td>545</td>
<td>401</td>
<td>401</td>
<td>401</td>
</tr>
<tr>
<td># scen</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Time</td>
<td>1.628</td>
<td>1.475</td>
<td>1.044</td>
<td>1.444</td>
<td>1.430</td>
</tr>
<tr>
<td>Mem</td>
<td>0.516</td>
<td>1.10</td>
<td>2GB+</td>
<td>4.2</td>
<td>2GB+</td>
</tr>
</tbody>
</table>

For selections $S_2$, $S_3$, and $S_4$, experiments are performed to evaluate all combinations of $Cmd$, $Tag$ and $Sid$ fields.

In Table 6.6, a ✔ means that all signals implementing a particular field for all selected events in selection $S_X$ are traced. ❌ indicates the results are not available due to the 10-minute time limit exceeded. Otherwise, all those signals don’t get traced. Third row (Bits) shows the total numbers of single-bit signals are traced for different selections. The runtime is in seconds and memory usage is in MB. From the table, we can see that selecting shared events leads to a smaller number of trace signals ($S_3$) compared with selecting unique events ($S_2$). However, if status signals controlling flow branching are selected without selecting any branch events, $S_4$ leads to the smallest trace signal selection.

From the table, it is quite obvious that not selecting $Cmd$ or $Sid$ has severe impacts on the trace analysis as explained in issues 1 and 2 in section 3.4. On the other hand, not selecting $Tag$ has negative impacts, although not as severe. The trace analysis can still finish even though it takes more time and memory. Next, compare the results obtained by selecting $Cmd$ and $Sid$ but no $Tag$ under $S_2$, $S_3$ and $S_4$. The results with $S_4$ are much better than $S_2$ or $S_3$. This is because no branch events were selected for $S_4$, therefore, issues 2 and 3 are avoided. Combined with the benefit of reduced trace signals, $S_4$ appears to be the best option. On
the other hand, not selecting any branch events may cause difficulty in understanding flow execution if a branch is long and a system execution fails to reach the end of that branch.

In the above discussion, selections of **Cmd**, **Tag** and **Sid** are applied to all events as the result of the system-level selection. A finer selection can be used to reduce trace signals if unique events and shared events are considered separately. For unique events, the sources where they are generated are known from flows, therefore Tags need not be traced. Shared events may be results of flow instances initiated by different components, therefore tracing Tags are necessary. On the other hand, tracing **Cmd** or not has little impact on the trace analysis. These points are supported by the results shown in columns under "U S". Under S2, we can see that the runtime performance, the complexity and accuracy of the trace analysis are similar while the trace signals are reduced with the finer selection, as shown by results under "U S". Comparing the results under "U S" against those obtained with only **Cmd** and **Sid** selected, the complexity is significantly dropped. The same conclusion can be drawn for S3 and S4.

From the above discussion, it is necessary to trace signals implementing **Cmd** and **Sid** whenever possible, and trace as many signals implementing **Tag** as allowed to reduce the complexity of the trace analysis even further. If **Tag** or **Sid** is not part of the design, we recommend adding DFx circuitry to trace such information. In the above experiments, the final execution scenarios under different signal selection, if available, contain the correct number of flow instances initiated and the orderings among the flow instances, as generated by the test environment, are correctly captured.
Chapter 7: Mining Message Flows Using Recurrent Neural Networks

A big challenge in SoC validation is that well defined specifications are not always available in a desired form. This is because writing formal specification is time-consuming, and it requires the developers have a comprehensive knowledge on the SoC design. Moreover, as the complexity of the SoC design gradually increases, the connection between the original specification (from design team) and the system behavior (from implementation team) becomes imprecise and disjoint. The lack of formal specifications may lead to potential misunderstandings of the design, and cause unintentional misbehavior to be implemented. It also hinders the debug process since effective debug requires well formed formal specifications. In order to overcome those challenges, an automatic system specification extraction is crucial for effective SoC validation.

In the literature, some existing methods [15, 16, 17, 18, 19, 20] allow extracting message flow specifications from system execution traces. However, they are mainly designed for software, and cannot be directly applied to SoC designs. In this chapter, we consider the concurrent nature of SoC designs, where execution traces are results from executing a number of message flows in parallel. In the proposed specification mining approach, the state-of-art LSTM neural networks, which are effective at capturing sequential dependencies, are trained with the SoC execution traces. Subsequently, sequential patterns are automatically extracted from the trained LSTM models, where message flow specifications can be formed from the mined sequential patterns.

The contribution of this work is a novel sequential pattern mining framework that automatically extracts sequential patterns from inherently concurrent SoC execution traces.
These patterns may be recurrent in individual traces, and repeat themselves in multiple different traces. Even though the type of execution traces and the type of mined sequential patterns are considered in separation in some previous work, to the best of our knowledge, this is the first work that both features are considered for mining.

7.1 Related Work

Whereas numerous algorithms are proposed for inferring specifications from logs of sequential programs [15, 16, 17, 18, 19, 20], they are not specifically designed for concurrent programs. Authors of [20] attempted to extract assertions automatically by applying data mining methodologies to silicon simulations traces. The authors go through several steps to pre-process the low-level gate-level signal trace to event level and then analyze the sequential dependencies among transaction events. This raised abstraction level speeds up the mining process, and significantly improves the quality of the extracted assertions. The work in [19] uses a similar concept and further raises the abstraction level to transaction level. This approach employs symbolic execution for trace generation, thus allowing for generalized parameters of the mined pattern. However, both approaches cannot handle concurrency that commonly exists in a complex SoC execution trace.

Some solutions are proposed to explore specification inference [65, 18, 66] from logs of concurrent system. The CloudSeer proposed in [66] presents a specification inference approach that can be applied to anomaly detection. However, its workflow model construction requires a log file with repeated executions of only one single task, which makes it infeasible for silicon trace mining. For the same reason, solutions in [65, 18] are also limited. Another similar approach, BaySpec [67], uses a dynamic mining approach to extracts formal specifications from Bayesian models. Because most of the existing data mining approaches are template based, they can only find known patterns. BaySpec efficiently solves the above issue by using Bayesian networks to conduct robust and incremental learning. This solution,
unfortunately, cannot be applied to silicon trace specification extraction because BaySpec requires clean trace grouping (i.e., requires functional segmentation), as well as starting and terminating events of each flow, which are hard to be obtained for SoC traces.

The work in [68] talks about an innovative workflow construction method that utilizes Long Short-Term Memory (LSTM). The LSTM is a particular type of recurrent neural network [69, 70] that captures the long-term dependencies in the training data. It treats the software traces as natural language sequences, and processes them with deep neural network models, from which sequential patterns are extracted and merged into set of workflows. Moreover, this solution allows construction of workflows (specifications) from the fully trained neural network model in a clear format. However, such methodology cannot be directly applied to silicon traces for similar reason as mentioned above. Compared to the traces generated from software executions, the silicon simulation traces contains only limited information. As the result, the concurrent data from silicon traces cannot be efficiently handled for pattern extraction using the same methodology in [68]. Some irrelevant patterns may be mined from the concurrent data patterns, adding more noise to the extraction process. Another interesting work in [71] applies state-of-art trace collection method and constructs a Prefix Tree Acceptor (PTA). It then combines the features from both PTA and the neural network model and eventually extracts the set of finite-state automatons describing features of the software model.

7.2 Message Flows and Sequential Patterns

As discussed in the previous section, existing mining methods are not suitable for highly concurrent SoC designs, which usually have multiple tasks executing in a interleaving way. Traces generated by the SoC designs require special handling in order to accurately extract the message flow specifications required by SoC validation. To address that problem, this section presents a message flow mining approach considering the unique natures of SoC
traces leveraging deep recurrent neural networks to capture sequential dependencies among events several for pattern extraction. It also presents several trace processing techniques to optimize the mining performance.

In SoC validation paradigm, the specification mining can be conducted at two levels as described below.

- Fabric Level — Mine flow specifications such as CPU downstream write/read, etc. Since the message flow specifications are implemented in hardware, mined patterns, if valid, are invariant across different execution traces.

- Application Level — Mine patterns among flow specifications that hold across different applications/tests. For example, the firmware loading flow should always happen after the firmware authentication flow. Application level patterns are represented as sequence of flows.

This section focuses only on the fabric level mining. The application-level mining will be left as future work.

First, the sequential patterns considered in our mining framework is defined. A sequential pattern \( p \) is a sequence of events such that

- Its length in terms of the number of events is at least two, and

- All its events are unique.

If ground truth patterns are known, we can define validity of mined sequential patterns with respect to the ground truth patterns. A mined pattern \( p_m \) is valid if there is a ground truth (GT) pattern \( p_t \) such that for every pair of events \( e_i \) and \( e_j \) in \( p_m \),

\[
e_i < e_j \text{ in } p_m \text{ if } e_i < e_j \text{ in } p_t.
\]
A mined pattern $pm$ is *invalid* if it is not valid. Consider a simple example. Suppose that $pt$ and $pm$ below are a ground truth pattern and a mined pattern, respectively. $pm$ is valid with respect to $pt$ as sequential dependencies between any pair of events in $pm$ also exists in $pt$.

$$pm: (0, 13, 15, 23)$$

$$pt: (0, 8, 12, 13, 15, 23, 24, 25)$$

Next, we show an interesting property that helps identify valid patterns from invalid ones. This property captures the cause-effect relation between two events, and it is based on the following observation: *any event in an execution trace is generated by a component in a SoC design in reaction to an input event.* The relation is referred to as the *causality* property. Two events $e_i$ and $e_j$ satisfy the causality property if

$$e_i.\text{dest} = e_j.\text{src}.$$  

Each execution in a message flow specifies a sequence of such relations. Therefore, for a pattern to be valid, every two consecutive events in a mined pattern must satisfy the causality property.

### 7.3 Mining Framework

Figure 7.1 shows the overview of the proposed mining framework. It accepts a set of SoC execution traces, and mines sequential patterns from those traces in three consecutive stages as described below.

#### 7.3.1 Trace Processing

The behaviors of modern SoC designs are often highly concurrent, while multiple flows can be executed simultaneously in an interleaving order. As a result, the causal dependencies
Figure 7.1: Overview of the sequential pattern extraction approach architecture

among consecutive events are very low. It is common that two consecutive events with strong sequential dependencies are from different flow instances, thus completely unrelated. This may cause invalid patterns to be mined. To avoid such situation, this section describes several trace processing methods to reduce the false dependencies among unrelated events.

During the trace processing stage, we first identify the set of $M$ unique events and assign an unique ID for each of them, ranging from 0 to $M - 1$. An unique event is defined by the combination of the static information ($\text{src, dest, cmd}$). Runtime information encoded in events can be diverse, as it is highly dependent on the structure of the specific SoC part the underlying event belongs to. When such runtime information becomes available, the mining result can be greatly improved. However, to keep generality of our mining framework, the only runtime information leveraged is the memory addresses encoded in events. We propose to take advantage of such runtime information, and slice original traces into subtraces where events become more correlated. Each subtrace is considered separately from others. As a result, if two consecutive events are unrelated, they are likely to be assigned to different subtraces, and the false dependencies introduced by them will be eliminated, therefore exposing the true causal dependencies. The way we slice the long interleaved trace is referred as trace slicing in this work. This process is nontrivial, as the types of data
field available for each event varies depending on the specific architecture of the part of SoC design they are located. Therefore, it is hard to precisely identify the flow instance an event belongs to, let alone grouping events of the same flow instance into subtraces. This section presents two slicing techniques.

7.3.1.1 Address-Based Trace Slicing

This technique considers the data field \( \texttt{addr} \) that is usually available for events generated for memory related tasks. For example, the event of any downstream read or write flows contains the target block address to be read or write. While multiple flow instances may share the same target block address, events with the same \( \texttt{addr} \) are not guaranteed to be from the same task. However, when events carries different \( \texttt{addr} \) values, they are always irrelevant, thus can be separated. For example, assume there are 3 unique events \((0, 1, 2)\), and the following trace is obtained:

\[
\{\{e_1(10)\}, \{e_2(10), e_1(15)\}, \{e_3(10), e_2(15)\}, \{e_1(15)\}\} \quad (7.1)
\]

Each bracket indicates the set of events executed at the same clock cycle, and the value inside the parentheses indicates the value of their \( \texttt{addr} \) data filed. For example, \( e_1(10) \) represents the instance of event \( e_1 \) with address 10. By separating events with different the \( \texttt{addr} \) values, this trace is sliced into two subtraces, with their \( \texttt{addr} \) value being 10 and 15 respectively:

\[
\begin{align*}
\text{addr} = 10 : & \quad \{\{e_1\}, \{e_2\}, \{e_3\}\} \\
\text{addr} = 15 : & \quad \{\{e_1\}, \{e_2\}, \{e_1\}\}
\end{align*}
\]
7.3.1.2 Causality Slicing

This trace slicing technique considers the causality property discussed in Section 7.2 and tries to group sequence of events based on their causal relationship. This algorithm starts with an empty set \( st \) to hold subtraces. Then, it iterates through each individual event \( e_x \), and check if it satisfy the causality property with any existing subtraces. Suppose a subtrace \( (e_0 \ldots e_i) \) and an event \( e_j \) such that \( e_i\text{.dest} = e_j\text{.src} \), then they satisfy the causal property, and \( e_j \) can be attached to the end of that subtrace. Otherwise, a new subtrace is created with \( e_j \) added as the the first event, and that new subtrace is then added the set \( st \).

To illustrate the basic idea of causality slicing, we take a simple example trace \( \rho \), where

\[
\rho = (e_0, e_1, e_2, e_3)
\]

The source and destination for each event is shown in Table 7.1.

<table>
<thead>
<tr>
<th>Event</th>
<th>Src</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_0 )</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>( e_1 )</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>( e_2 )</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>( e_3 )</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

The destination of \( e_0 \) matches with the source of \( e_2 \). As the same way, \( e_1 \) matches with \( e_3 \), forming to two subtraces, as shown below:

\[
\text{trace}_0 : \ (e_0, e_2) \\
\text{trace}_1 : \ (e_1, e_3)
\]
This example shows the ideal scenario for the causality slicing. However, an event may satisfy the causality property with multiple subtraces. When this situation happens, and only limited information is available to determine the subtrace that event belongs to, the causality slicing algorithm combines the satisfying subtraces into one, and add the event to the combined subtrace. This is to maintain the original order of trace, and in the mean time avoid introducing incorrect dependencies.

7.3.2 LSTM Training

Neural networks have shown its significant effects in various fields in recent years. In this dissertation, we explore the recurrent neural network (RNN), a special type of neural work that is designed to capture sequential dependencies.

7.3.2.1 Architecture

In this dissertation we use a particular type of RNN called Long Short Term Memory (LSTM) networks. LSTM is suitable for capturing sequential dependencies embedded in silicon trace as it provides the ability to solve the "long-term dependencies" compared to the normal RNNs. The common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. At each time stamp, a single LSTM block uses these gates to decide the portions of the information to retain and update, and produce the new output.
Figure 7.3: The overall structure of the proposed LSTM model

$h_t$ for its connected block. Each LSTM unit contains a set of weights that controls how the gate operate, Figure 7.2 shows the structure of a LSTM unit. And the training process of a LSTM model is the process of assigning the proper value to the set of weights, using complex complex algorithms. In this work we use the categorical cross-entropy loss to calculate the model error and update the weight of each gate correspondingly during the training.

Figure 7.3 shows the overall structure of the proposed LSTM model. It is composed of two hidden layers, an input layer and an output layer, using standard encoding-decoding algorithms. Every layer contains a set of recurrent LSTM units. This dissertation uses one unit for each type of unique event, hence every layer is constructed with $M$ units, shown in Figure 7.3. $M$ is the total number of unique events in the training traces.

Given an input sequence $S = (e_0, e_1 \ldots e_h)$, the input layer encodes each event into a one-hot vector $\vec{x}$. And the output layer decodes the final output into a probability distribution using a standard multinomial logistic function. The probability distribution is represented as a set of $P[(e_{h+1} = e) \mid S]$ for each unique event $e$. Equivalently $P[(e_{h+1} = e) \mid S]$ also represents the probability of pattern $(e_0, \ldots, e_h, e)$. 

88
7.3.2.2 Training

In the proposed approach, the patterns are grouped by their lengths (the numbers of events in a pattern). We create a set of unique LSTM models, one for each separate pattern type individually to allow for better mining accuracy. As the desired pattern types vary on different SoC designs, this approach allows configurabilities of the types of patterns it needs to mine. The required pattern types to mine can be specified by the debuggers based on their understanding of the SoC design. Assume the user defines request patterns with length ranging from 2 to \( q \), then the proposed work will initiate \((q - 1)\) models. Each of the model presented as \( M_w \) such that \( w \) is the input sequence size, and \( w \) ranges from 1 to \( q - 1 \).

During the training process, each LSTM model \( M_w \) takes a set of training pairs in the format of \((S, e_x)\) where \( S \) is the input training sequence of the LSTM model, and \( e_x \) is the expected event that should be fired right after, serving as the training label for input \( S \). Let \( \rho \) be a sequence of \( k \) events as our training trace:

\[
\rho = (e_0, e_1 \ldots e_k)
\]

Then the following training pairs are generated from \( \rho \) to train \( M_w \):

\[
\{(e_0 \ldots e_{w-1}), e_w\} \ldots ((e_{k-w} \ldots e_{k-1}), e_k)\}
\]

We repeat this process for all LSTM models.

7.3.3 Pattern Extraction

After all LSTM models are properly trained, this work propose to extract patterns from each model in an chained nature, as shown in Figure 7.1. We start with the model \( M_1 \) that has the smallest input sequence length value of 1. This model takes one unique input
sequence $S$ of length 1 at a time, and generate the probability distributions $P(S, e)$ for each unique event $e$. $P(S, e)$ represents the probability of event $e$ being observed after sequence $S$. Equivalently, it shows the probability of sequence $(S, e)$. We define threshold (represented as $\theta$) to be the minimum probability to characterize $(S, e)$ as a pattern. Based on this condition, the proposed work can extract a set of patterns after all input sequences are considered. This process can be repeated for all LSTM models, where we consider all sequence inputs, and extract patterns from the output. However, generating all input sequences can be computational expensive as the input sequence length increases, leading to potential space explosion. We propose to chain the LSTM models in an ascending order with regard to their pattern length such that the extracted patterns in previous model is used as the input of the next model. For model $M_w$ with input sequence length being $w$, its generated patterns (by concatenating the the input sequence and the one event output) has length of $w + 1$, which is the same format of input sequence for model $M_{w+1}$. In this way, any generated sequences with probability below the threshold is no longer considered, reducing the computation cost of considering potentially incorrect sequences.

### 7.4 Experimental Results

To evaluate the presented framework, this experiment uses the same SoC design in Figure 6.1 executed with Test 1. We collect traces from 200 executions to train the LSTM models. Each execution is fired with a different random seed to generate a set of diverse traces. And these execution traces are used to train a set of LSTM models with different pattern lengths. For this experiment, the maximal pattern length is set to be 8. This indicates that 7 different LSTM models are trained. The training process is conducted on the USF computer cluster with about 100 manycore GPU co-processors. The training process of each model takes approximately 20 – 30 minutes.
Figure 7.4 shows the results obtained using three different types of traces: the original traces, address sliced traces, and causality sliced traces. And for each type of traces, we experiment with four different thresholds: 0.2, 0.4, 0.6 and 0.8, each represented with a different color in the figure. The definition of $\theta$ is the probability threshold for mining as described in section 7.3.3. The x-axis represents the pattern lengths, and the y-axis shows the numbers of mined patterns for each pattern length.

From the figure, it can be seen that the number of mined patterns significantly increases after applying slicing techniques. Moreover, when $\theta = 0.8$, no patterns can be extracted from...
the original trace, while both sliced traces generate over 100 patterns with different lengths. It also shows that less number of patterns are mined as the threshold increases. Every time the threshold increases by 0.2, the number of mined patterns from the original traces is reduced by more than 50%. Such trend is also observed for the sliced traces, although with lower reduction rates.

Table 7.2 and 7.3 show the detailed result for the original traces and causality sliced traces, respectively, with the threshold $\theta = 0.2$. We use V, IV, F, and NF to denote valid patterns, invalid patterns, ground-truth patterns that are mined, and ground-truth patterns that are not mined, respectively.

Comparison between Table 7.2 and Table 7.3 shows the significant effect of trace slicing techniques. The total number of valid patterns increase from 28 to 257. Notice that while the trace slicing technique reduces the overall sequential dependencies considered by grouping related events into separate subtraces, it increases the probability of the patterns mined from

<table>
<thead>
<tr>
<th>Length</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>V&amp;F</td>
<td>21</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IV&amp;F</td>
<td>24</td>
<td>50</td>
<td>65</td>
<td>74</td>
<td>83</td>
<td>99</td>
<td>121</td>
</tr>
<tr>
<td>V&amp;NF</td>
<td>97</td>
<td>140</td>
<td>139</td>
<td>130</td>
<td>118</td>
<td>88</td>
<td>44</td>
</tr>
</tbody>
</table>

Table 7.2: Mined patterns using the original traces with $\theta = 0.2$

<table>
<thead>
<tr>
<th>Length</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>V&amp;F</td>
<td>56</td>
<td>45</td>
<td>36</td>
<td>34</td>
<td>39</td>
<td>32</td>
<td>19</td>
</tr>
<tr>
<td>IV&amp;F</td>
<td>16</td>
<td>38</td>
<td>59</td>
<td>80</td>
<td>99</td>
<td>130</td>
<td>183</td>
</tr>
<tr>
<td>V&amp;NF</td>
<td>62</td>
<td>101</td>
<td>104</td>
<td>96</td>
<td>79</td>
<td>56</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 7.3: Mined patterns using the sliced traces with $\theta = 0.2$
Table 7.4: Mined patterns after applying causality filtering

<table>
<thead>
<tr>
<th>Length</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>V&amp;F</td>
<td>56</td>
<td>45</td>
<td>36</td>
<td>34</td>
<td>39</td>
<td>32</td>
<td>19</td>
</tr>
<tr>
<td>New IV&amp;F</td>
<td>4</td>
<td>14</td>
<td>21</td>
<td>31</td>
<td>37</td>
<td>44</td>
<td>66</td>
</tr>
<tr>
<td>V&amp;NF</td>
<td>62</td>
<td>101</td>
<td>104</td>
<td>96</td>
<td>79</td>
<td>56</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 7.5: Mined patterns after applying additional threshold

<table>
<thead>
<tr>
<th>Length</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>V&amp;F</td>
<td>79</td>
<td>80</td>
<td>70</td>
<td>57</td>
<td>57</td>
<td>42</td>
<td>23</td>
</tr>
<tr>
<td>IV&amp;F</td>
<td>8</td>
<td>31</td>
<td>59</td>
<td>89</td>
<td>108</td>
<td>120</td>
<td>146</td>
</tr>
<tr>
<td>V&amp;NF</td>
<td>39</td>
<td>66</td>
<td>50</td>
<td>73</td>
<td>61</td>
<td>46</td>
<td>21</td>
</tr>
</tbody>
</table>

the subtraces including both valid and invalid ones. As a result, while the causal dependencies among the sliced traces become stronger, leading to more valid patterns being discovered, the numbers of invalid patterns mined also increases. Such effect is shown by the bold number for the increased valid patterns and red (italic) number for the increased invalid patterns.

Table 7.3, showing the best result obtained by now, reveals two issues. First, the number of the invalid patterns is relatively high, especially for patterns with longer lengths. Second, the valid patterns mined are a small subset of overall mined patterns. We propose to address these issues with the following techniques:

7.4.1 Causality Filtering

This technique considers the causality property explained Section 7.2, and eliminates patterns that contain at least one consecutive event pair with no causal dependencies. Table 7.4 shows the new IV&F (in bold font) after this filtering technique is applied to the previous result in Table 7.3. It can be seen that more than 50% of invalid patterns are removed while the numbers of other types of patterns remain the same.
7.4.2 Additional Threshold for Model Input Generation

To avoid generating invalid patterns, the proposed approach conducts threshold filtering at every step. In this way, not only the patterns with probability below the threshold is eliminated, but also any longer patterns containing such patterns are no longer considered. However, this approach fails to consider the case where a low probability pattern could be a sub-pattern of another pattern with a high probability. To avoid such situation, we only consider the patterns above the threshold as valid. Also, a lower threshold $\theta'$ is used to generate inputs for the next model to extract longer patterns. Such change allows more patterns to be considered for each model, while still maintains the quality of the extracted patterns (with the same threshold). Table 7.5 shows the results of applying the new threshold $\theta' = 0.05$. Compared to Table 7.4, the numbers of valid patterns increases by 147 in total. However, as more inputs are considered on each step, it also mines more invalid patterns as shown by the red (italic) numbers in row 4.

7.4.3 Initiating Event Filtering

As discussed in section 7.2, a valid pattern represents the dependencies among certain events in a flow specification. For each execution of a flow, multiple valid patterns could be mined. However, there exists one valid pattern that completely matches the execution, thus covering all dependencies in all other mined patterns for that particular flow. For this specific experiment, there are 58 such patterns implemented in the SoC design, including 4 patterns that contains 2 events, 10 patterns containing 5 events, and 44 patterns with its length being 8. This section discusses a techniques that can be applied to extract such patterns for each flow.

According to the definition in 3.1, a flow execution is presented as a sequence of event firings from the initial state to the end state. As a result, an execution of any flow always satisfies two conditions: (1) it is activated by an initiating event; (2) it ends with a termi-
nation event. If those starting/terminating events can be identified from execution traces, we can limit the scope of valid patterns to be mined. In this work we propose to extract the start events from the execution traces. Given a trace $\rho$, we first collect the set of unique events included. And for each event $e$, we find its first appearance in the trace. Then we iterate through all events occurred before event $e$’s first appearance. If there exist no event $e_j$ such that $e_{.src} = e_{.dest}$, implying the causal relationships between $e$ and $e_j$, then $e$ is an start event. We apply the same process to all traces and collect a set of starting events.

Using this set of starting events, we are able to filter out patterns that are not initiated by the starting events. Notice that the filtered patterns are not necessary incorrect. Some of them are subsequences of longer patterns that are correct. We do not consider such patterns in this situation because we only want to find the longest pattern that covering all dependencies in all other mined patterns for each particular flow. This technique is able to filter all invalid patterns, and it helps us to extract 30 patterns out of 58 ground truth patterns. It includes 2 two events patterns, 5 four events patterns and 23 eight events pattern.

7.5 Limitations

As discussed in the previous section, this work suffers from two problems. First, the numbers of the invalid patterns is relatively high, especially for patterns with longer lengths. Second, the valid patterns mined only are a small subset of overall mined patterns. The main cause behind this inaccurate mining result is the intrinsic concurrent nature of the SoC traces. While the execution traces of an SoC are results from executing a number of message flows in parallel, it is highly likely two consecutive messages in such trace is completely unrelated. This introduces a lot of incorrect dependencies, leading to invalid patterns being mined. Moreover, the weak causal dependencies among the SoC traces makes it harder to capture the valid patterns.
We proposed several trace processing techniques in this dissertation in order to increase the causal dependencies of each trace, and were able to generate some promising results. However, this issue still persists and future works need to be done to further improve the quality of the mining result.
Chapter 8: Conclusion and Future Work

This dissertation describes a communication-centric framework that enables a more efficient system-level post-silicon debug for complex SoC designs. It enables system-level communication behavior reconstruction from partially observed silicon traces. This framework includes three main components. First, it includes an off-chip trace analysis method which infers possible scenarios about internal executions of message flows corresponding to an observed silicon trace that is possibly incomplete and lossy. For post-silicon debug, the flow execution scenarios produced by our method can provide some more structured information on system operations, which is more understandable to system validators. This information, combined with debugger’s insight, can greatly help to locate design defects more easily as well as provide a measurement of validation coverage. Moreover, this proposed framework returns a set of flow events that cannot be mapped to any scenario which can be used to decide whether the specifications are correctly implemented by the SUD.

Secondly, the proposed framework includes an on-chip communication monitoring infrastructure that enables detection and offloading of communication transactions on-the-fly during the SoC execution. It shows great effects on enhancing the SoC observability as well as assisting the efficient and accurate trace analysis. Then, it implemented a communication event selection method that is designed to choose a subset of debug-critical events for more effective observation of a set of message flows. This off-chip event selection approach helps to capture comprehensive information, thus allowing for a better understanding of the communications among components during system execution.
We demonstrate the framework with experiments on a cycle-accurate and pin-accurate multicore SoC prototype. Experimental results show promising effects of the proposed debugging framework. In the meantime, it shows that the on-chip monitoring infrastructure incurs very little overhead in area and logic complexity.

Additionally, this dissertation proposes an approach that automatically extracts message flow specifications from SoC transaction-level traces. Comprehensive and well defined specifications are the foundation of the above trace analysis framework and many other SoC design activities. This approach develops several trace processing techniques and hides concurrency in SoC execution traces while preserving essential sequential dependencies. It utilizes the innovative LSTM models to capture sequential dependencies in the traces. We apply this approach on the executions traces of a non-trivial multicore SoC prototype and evaluate the quality of mined specifications. It is demonstrated that the trained neural network model has a high correct rate on mining the message flow specifications integrated in an SoC design.

8.1 Future Works

In the future, we plan to perform in-depth studies on using the described framework on SoC designs with diverse interconnects and further optimize the framework to offer higher observability with reduced hardware overhead.

Due to limited observability, our proposed approach may derive a large number of different flow execution scenarios for a given signal trace. One of the promising but barely explored research field is trace coalescing. Trace coalescing aims to infer, from a combined set of traces under different observabilities, a more comprehensive and accurate flow execution scenario, such to facilitate debug and error scenario construction. While it is hard to observe all essential information in one execution run due to the hardware overhead, the DfD features commonly exist for silicon allows flexible configurability of what to observe. Moreover, post-silicon execution is orders of magnitude faster compared pre-silicon simulation. As a result,
multiple traces of different aspects of a system run can be easily obtained during post-silicon debug for trace coalescing. In the future, we plan to explore the the effects of trace coalescing and evaluate its effect on the trace analysis result.

Insights from system validators can also help to eliminate some false scenarios due to the partial observability. An interesting future direction is the formalization of the validators’ insights using temporal logic on flows so that the validators can express their intents more precisely and concisely.

For the flow mining approach, we plan to explore different trace slicing techniques to enable better mining result. A potential future direction is to consider more data fields that are available for each events, thus to group related traces more precisely, reducing the incorrect dependencies in the original traces. Moreover, we plan to consider the causality property during the training process. That is, the LSTM model will be trained only on input sequences that satisfy the causality property. This also helps eliminate false dependencies in the traces. We also plan to extend the mining approach to application level where we mine the dependencies among different specifications.
References


[44] Amir Masoud Gharehbaghi and Masahiro Fujita. Transaction-based post-silicon de-
bug of many-core system-on-chips. In Quality Electronic Design (ISQED), 2012 13th

communication-centric debug of embedded multi-processor socs. In VLSI Design, Au-


[48] Alan P Su, Jiff Kuo, Kuen-Jong Lee, Jer Huang, Guo-An Jian, Cheng-An Chien, Jiun-In
Guo, and Chien-Hung Chen. Multi-core software/hardware co-debug platform with arm
coresight, on-chip test architecture and axi/ahb bus monitor. In Proceedings of 2011
International Symposium on VLSI Design, Automation and Test, pages 1–6. IEEE,
2011.

[49] Kanad Basu and Prabhat Mishra. Efficient trace signal selection for post silicon valida-
IEEE, 2011.

[50] Xiaobang Liu and Ranga Vemuri. Fast heuristics for near-optimal signal restoration in
post-silicon validation. In 2018 IEEE Computer Society Annual Symposium on VLSI

[51] Xiaobang Liu and Ranga Vemuri. Assertion coverage aware trace signal selection in
post-silicon validation. In 20th International Symposium on Quality Electronic Design


Appendix A: Protocol Specifications in Message Sequence Chart Provided by GEM5

Figure A.1: Flow sequence chart of write operation when requested data is not included in Dcache.

Figure A.2: Flow sequence chart of write operation when XCache has the exclusive right of requested data.

Figure A.3: Flow sequence chart of write operation when requested data is shared by another component.
Figure A.4: Flow sequence chart of read operation when XCache has the exclusive right of requested data.

Figure A.5: Flow sequence chart of read operation when requested data is shared by another component.

Figure A.6: Flow sequence chart of read operation when requested data is not present in the Cache.
Appendix B: Protocol Specification in LPNs Provided by GEM5

Figure B.1: Flow specification of a cache coherent write operation initiated from CPU1 to instruction cache.
Figure B.2: Flow specification of a cache coherent read operation initiated from CPU1 to instruction cache.
Figure B.3: Flow specification of a cache coherent read operation initiated from CPU1 to data cache.
Appendix C: Protocol Specification in Message Sequence Charts for the RTL Model

Figure C.1: CPU write when cache has exclusive right of the requested data.

Figure C.2: CPU write when data only exist in the other CPU’s cache

Figure C.3: CPU write when requested data only reside in Memory

Figure C.4: Cache send write back request to Memory
Figure C.5: CPU read when cache has exclusive right of the requested data.

Figure C.6: CPU read when data only exist in the other CPU’s cache

Figure C.7: CPU read when requested data only reside in Memory

The read and write protocols in RTL model are very similar to what we used in GEM5 simulator. However, the command name used here is different.
Appendix D: Protocol Specification in LPNs for the RTL Model

There will be 3 protocols in total: read, write and write back protocol. All the write operations are implemented in protocol presented in Figure D.2. When the request activate cache coherent protocol, like in Figure C.2, it will end in state 17. The rest will end in state 9. All read operations are implemented in protocol presented in Figure D.3. Specification in Figure C.6 will end in state 17. The rest of the specification without activating cache coherence protocol end in state 9.

Figure D.1: Flow specification of a cache write back operation initiated from Cache1.

\[ msg1 : ( \text{Cache1, Bus, wb} ) \]
\[ msg2 : ( \text{Bus, Memory, wb} ) \]
\[ msg3 : ( \text{Memory, Bus, wb} ) \]
Figure D.2: Flow specification of a cache coherent write operation initiated from CPU1 to Cache.
Figure D.3: Flow specification of a cache coherent read operation initiated from CPU1 to Cache.

msg1: (CPU1, Cache1, rd)  
msg2: (Cache1, Bus, rd)  
msg3: (Bus, Cache2, snp)  
msg4: (Cache2, Bus, snp)  
msg5: (Bus, Memory, rd)  
msg6: (Memory, Bus, rd)  
msg7: (Bus, Cache1, rd)  
msg8: (Bus, Cache1, rd)  
msg9: (Cache1, CPU1, rd)  
msg10: (Cache1, CPU1, rd)  
msg11: (Cache1, CPU1, rd)
Appendix E: Copyright Permissions

The permissions below are for the use of material in Chapter 3 and Chapter 6.

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author’s approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author’s university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity’s name goes here]’s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.
Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author’s approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author’s university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity’s name goes here]’s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.
The permission below is for the use of material in Chapter 4 and Chapter 6.

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © [Year of original publication] IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis online.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]’s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.
The permission below is for the use of material in Chapter 5 and Chapter 8.

**Title:** A Communication-Centric Observability Selection for Post-Silicon System-on-Chip Integration Debug

**Conference:** 20th International Symposium on Quality Electronic Design (ISQED)

**Author:** Yuting Cao; Hao Zheng; Sandip Ray

**Publisher:** IEEE

**Date:** 6-7 March 2019

Copyright © 2019, IEEE

**Thesis / Dissertation Reuse**

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author’s approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author’s university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity’s name goes here]’s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Copyright © 2019 Copyright Clearance Center, Inc. All Rights Reserved. Privacy statement. Terms and Conditions. Comments? We would like to hear from you. E-mail us at customer@copyright.com