Protocol Guided Trace Analysis for Post-Silicon Debug Under Limited Observability

by

Yuting Cao

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Hao Zheng, Ph.D.
Swaroop Ghosh, Ph.D.
Srinivas Katkoori Ph.D.

Date of Approval:
To be determine

Keywords: silicon, validation, trace, analysis, observability, signal selection

ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my advisor Prof. Hao Zheng for the continuous support of my Master 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 thesis. I could not have imagined having a better advisor and mentor for my Master study.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Swaroop Ghosh, and Dr. ... for their encouragement, insightful comments, and hard questions.

Finally, I must express my very profound gratitude to my parents and to my boyfriend Jae-won Jang or providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This thesis considers the problem of reconstructing system- level behavior of an SoC design from a partially observed signal trace. Solving this problem is a critical activity in post- silicon validation, and currently depends primarily on human creativity and insights. In this thesis, we provide an algorithm to automatically infer system-level transactions from incomplete, ambiguous, and noisy trace data. This thesis also demonstrates the approach on a multicore virtual platform developed within the GEM5 environment and RTL model.

CHAPTER 1

INTRODUCTION

This chapter briefly explains definitions of pre- and post-silicon validation, discusses current challenges in post-silicon validation and possible solutions. This chapter also reviews related works in post-silicon validation and reasons why proposed trace based analysis algorithm in this thesis is necessary.

1.1 Pre- and Post-silicon Validation

Integrated circuits (ICs) are designed from a system specification that can be used to ensure certain behaviors and functionalities of the system. Then IC designers converts the specification into a transaction level description, and with more details, eventually converts to an register transfer level (RTL) description. The RTL describes the exact behavior of the digital circuits on the chip, as well as the interconnections to inputs and outputs [22]. Once the RTL model is verified, it is fabricated on silicon and eventually is released to the market. A more detailed steps of IC design flow is shown in Figure 1.1. To ensure the correctness of the design, validation is conducted on the system at end of each step of the design process. As the design goes through each step, the abstraction level of the system decreases. Consequently, the system become more complicated, and thus harder to refine when bug occurs. This requires the debuggers to find bugs as early as possible to reduce the cost.

As Moore's law continues, IC designs are becoming more complex. As a result, the numbers of system bugs are growing and the type of bugs are becoming more diverse and harder to root cause. Recent studies have shown that validation in modern IC develop

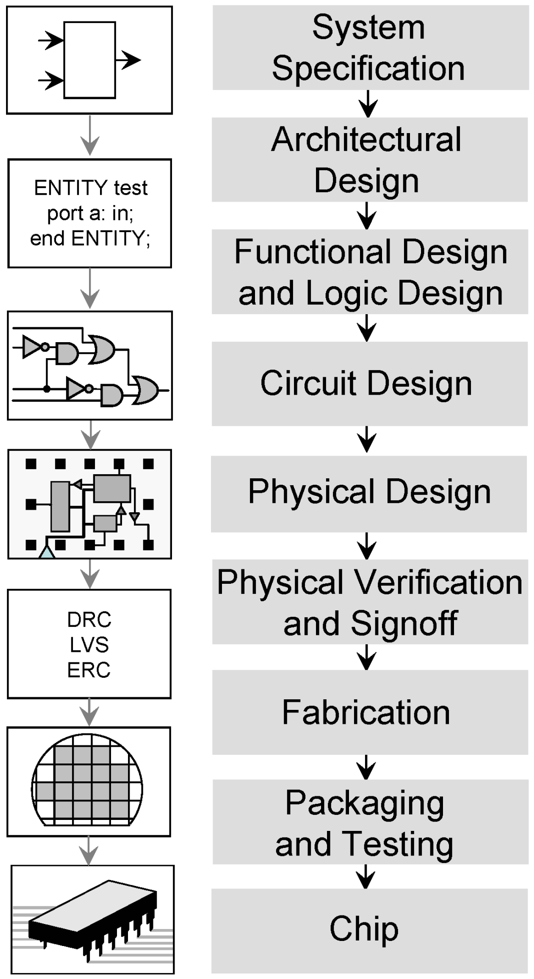


Figure 1.1. Major steps in the IC design flow [22]

process takes up to 70% of design time and is increasing [4]. Here we define Validation as the activity of ensuring a product satisfies its specifications, compatible with related software and hardware and meets user expectations [3].

Many effort have been focused on Pre-silicon validation, where it aims to verify the architecture design before it's implemented on an actual chip. Pre-silicon validation is a very important research top because the cost to debug and refine the system is relatively low compared to changing the design after it is fabricated on the silicon chip. Pre-silicon validation techniques includes simulating the system at RTL level on simulators, A field-programmable gate array (FPGA), or emulators. Event-driven simulator is a very commonly used technique in pre-silicon validation, this technique simulates the circuit behavior with test vectors to check the correctness of design's functionalities. This requires all possible behaviors of the design to be considered to design the test vector. Consequently, as the circuit size increases, the number of possible behaviors grows exponentially, making exhaustive simulation impractical. As it is the same situation for the rest of the pre-silicon validation techniques,only small portions of the design can be tested during pr-silicon validation, achieving acceptable coverage [5]. Another drawback of pre-silicon validation is the speed limitation. The fact that simulation is mainly done on software, makes the simulation multiple orders of magnitude slower compared to the actual circuit speed. To shorten the time required for simulation, debuggers can implement designs into FPGA, achieving up to 3 orders of magnitude faster, but this speed is still relatively slow compared to actual chip speed. Emulator can combine multiple FPGAs to work on a larger portion of the RTL design, but it does not help with the speed limitation. All these limitations makes it impossible to guarantee the first silicon error-free, hence it is necessary to push part of testing to post-silicon validation.

Post-silicon validation makes use of pre-production silicon IC to ensure that the fabricated system 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 RTL simulations, and even provide speed-up of several orders of magnitude over other pre-silicon platforms (*e.g.*, FPGA, system-level emulation, etc.). This makes it possible to explore deep design states which cannot be exercised in pre-silicon, and identify errors missed during pre-silicon validation and debug.

1.2 Post-silicon Debug Techniques and Challenges

Post-silicon debug is very labor-intensive and may take months to finish. As showed in Figure 1.2, 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 produce larger , denser, and more complex designs [6].

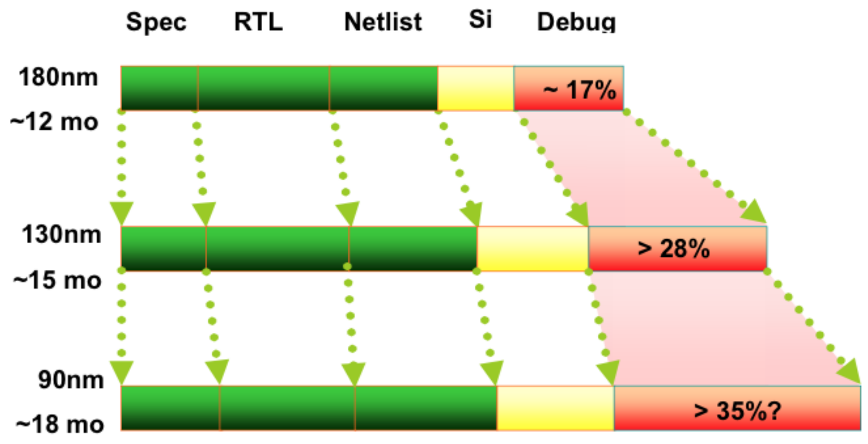


Figure 1.2. Silicon Debug vs. time-to-market

Despite the importance of post-silicon validation, there are not enough research done to improve its sufficiency. Post-silicon debugging is difficult because the amount of internal states engineers can observe is limited. Several chip-debugging techniques are proposed, like probe needles and eletron-beam probing. However, the decreasing silicon sizes and multiple metal layers in modern silicon structure makes debugging techniques based on visual inspection or direct physical contact very difficult [7]. Another commonly used tools are embedded software, mostly done by monitoring processor bus activity, this method provides only limited observability of the internal hardware domain[8].

A promising alternative is to gather internal signal information by inserting design-for-debug(DfD) into the circuit design. Two of the most commonly used techniques are scan chains and trace buffers. Detailed definitions and debug technologies based on these two techniques will be discussed in next section.

1.2.1 Scan Based Techniques and Challenges

The goal of scan based technique is to reuse the internal scan chains that is placed in the CUD. This is original designed to increase the controllability and observability of the system during manufacturing test by using the functional pins as scan pins to load multiple scan chains concurrently to reduce test time [1].

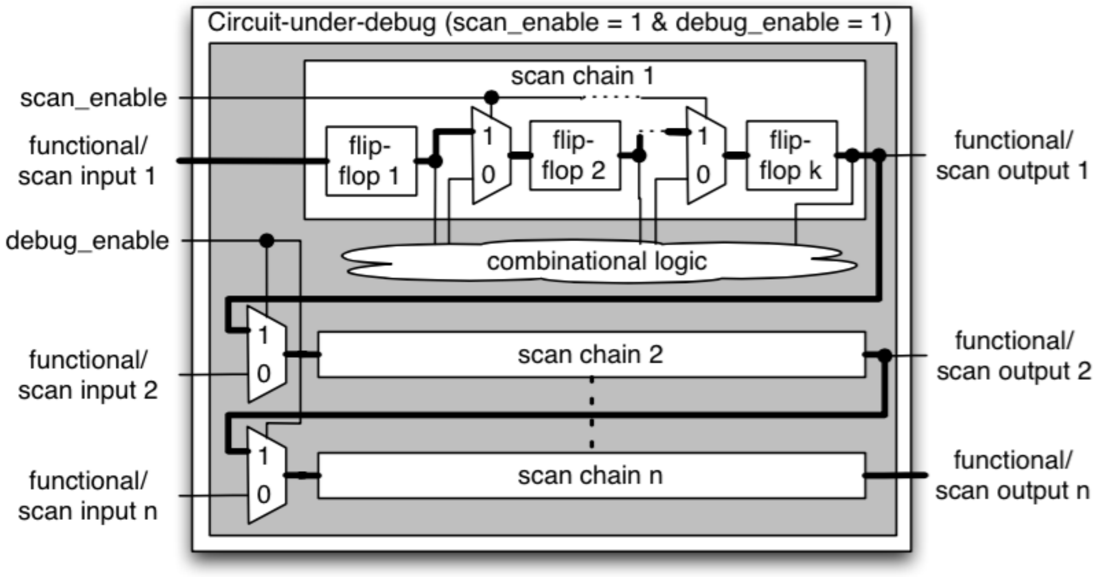


Figure 1.3. Scan-based techniques [1]

For post silicon validation purpose, these scan chains are concatenated as shown in Figure 1.3, where internal states will be loaded and unloaded through a serial interface. During the post silicon debug process, when internal state of the system is needed, debuggers will stop the system, enable scan chains to capture and offload the internal state elements (scan dump). After scan dump is finished, the system should be able to be resumed from where it was stopped.

During post silicon validation, when the system is deterministic, allowing test experiment to be able to be stopped and resumed from any state of interest, scan chains can be very useful. However, most of the system failures are not reproducible and there is little knowledge about the cause of failure. Moreover, system nowadays always contains multiple clock cycles, stopping the system without causing any effect on the system is impractical. For these reasons, scan chains are not practical for complicate systems.

1.2.2 Trace Based Techniques and Challenges

Trace buffers is an embedded memory structure that can store certain amount of signal data. Once the trace buffer is full, the data can be offloaded through low speed device pins for analysis purpose.

The limitations of scan chains can be address by using embedded logic analyzers (ELA) using trace buffer. Figure 1.4 shows an example of structures of ELA. There are four com-

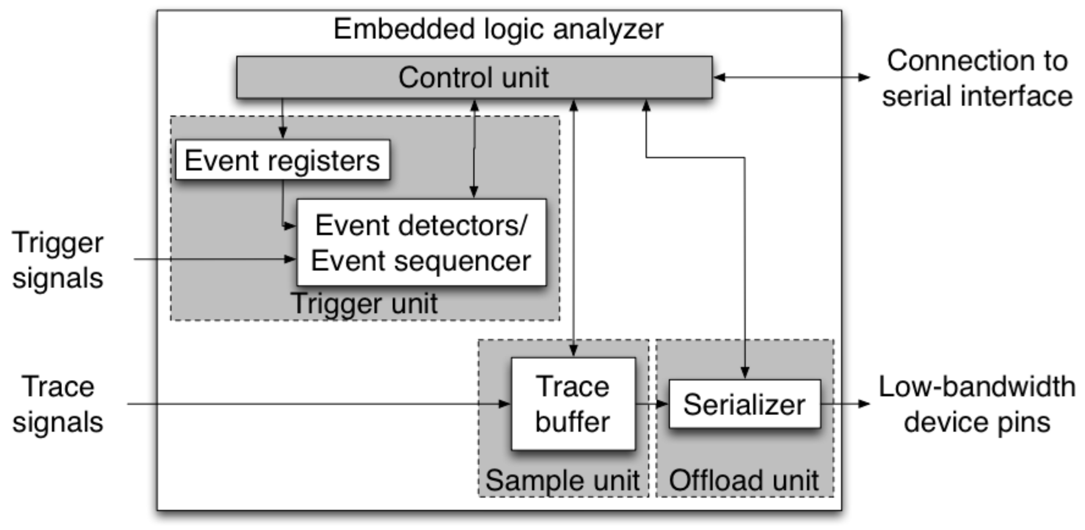


Figure 1.4. Structure of an embedded logic analyzer [1]

ponents inside the ELA: control unit, trigger unit, sample unit (trace buffer) and offload unit. Control unit is in charge of all the other unit inside ELA. Trigger unit monitors a set of trigger signals to detect trigger event and thus activate the sample unit to start the data acquisition. The sample unit contains a trace buffer to record data of selected signals while offload unit output the data through low-bandwidth device pins.

The amount of data can be acquired by trace buffer is limited by two aspects:

- trace buffer *width*: limits numbers of observable trace signals
- trace buffer *depth*: limits numbers of samples to be stored.

Compared with scan chains, trace buffer allows temporal observability, making trace analysis possible even when the location of bug is not known. However, because of the limitation of the trace buffer *width*, the amount of signals can be observed is limited. This problem can be mitigated using trace information filtering [9] and compression techniques [10].

1.3 Related Work

Our work in this thesis is closely related to communication-centric and transaction based debug. An early pioneering work is described in [11], which advocates the focus on observing activities on the interconnect 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 [12, 13]. However, this line of work is focused on the network-on-chip (NoC) architecture for interconnect using the run/stop debug control method.

A similar transaction-based debug approach is presented in [14]. Furthermore, it proposes an automated extraction of state machines at transaction level from high level design models. From an observed failure trace, it performs backtracking on this 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 the constraints on the internal variables is used to refine the set of transaction traces to remove the infeasible traces. This approach requires user inputs to identify impossible transaction sequences, and may not find the states causing the failure if the transaction traces leading to the observed failure state is long. Backtracking from the observed failure state requires pre-image computation, which can be computationally expensive. A transaction-based online debug approach is proposed in [15] to address these issues. This approach utilizes a transaction debug pattern specification language [16] to define properties that transactions should meet. These transaction properties are checked at runtime by programming debug units in the on-chip debug infrastructure, and the system can be stopped shortly after a violation is detected for any one of those properties. In this sense, it can be viewed as the hardware assertion approaches in [17] elevated to the transaction level.

In [18], 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 across a wide spectrum of the 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 a large set of events and transactions observable. 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 now defined at the abstract level of events and transactions rather than the raw signals, and thus can be evaluated more efficiently. In [19], 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.

1.4 Motivation

Post-silicon validation 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 techniques for streamlining and automating post-silicon validation activities.

A key component of post-silicon validation of SoC designs is to correlate traces from silicon execution with the intended system-level transactions. An SoC design is typically composed of a large number of pre-designed hardware or software blocks (often referred to as "intellectual properties" or "IPs") that coordinate through complex protocols to implement the system-level behavior. Any execution trace of the system involves a large number of interleaved instances of these protocols. For example, consider a smartphone executing a usage scenario where the end-user browses the Web while listening to music and sending and receiving occasional text messages. Typical post-silicon validation use-cases involve exercising such scenarios.

An execution trace would involve activities from the CPU, audio controller, display controller, wireless radio antenna, etc., reflecting the interleaved execution of several communication protocols. On the other hand, due to observability limitations, only a small number of participating signals can be actually traced during silicon execution. Furthermore, due to electrical perturbations, silicon data can be noisy, lossy, and ambiguous. Consequently, it is non-trivial to identify all participating protocols and pinpoint the interleaving that results in an observed trace.

With the increasing complexity of modern SoC designs nowadays, debugging protocols inside IP blocks by themselves is not enough anymore. The complexity of the SOC increasingly resides in the interactions between the IP blocks. Therefore, debug must be conducted at a higher system level, where the computation threads and communication threads interact. Because the interconnect implements the communication, and hence the synchronization between the IP blocks is the natural focus for system-level debug [11].

1.5 Contributions

In this thesis, we consider the problem of reconstructing system-level behavior from silicon traces in SoC designs that is abstracted in state of protocol specifications. Given a collection of system-level communication protocols and a trace of (partially observed) hardware signals, our approach infers, with a certain measure of confidence, the protocol instances (and their interleavings) being exercised by the trace. The approach is based on a formalization of system-level transactions via labeled Petri-Nets, which are capable of describing sequencing, concurrency, and choices over system events. We develop algorithms to infer system-level transactions from traces with missing, noisy, and ambiguous signal values.

The proposed approach can give silicon debugger an overview of the system behaviors, which can be used to check if the system performs the desired behavior and thus decide the correctness of the design. Moreover, this approach maps the signal trace into flow instances. By checking if a trace can be successfully mapped to flow instances, the debuggers can judge if the those specifications are correctly implemented in the silicon. When error occurs, our proposed method can provide interesting information in system level to help root cause the problem.

For the following part of this thesis, we present background information on the area of post-silicon validation in Chapter 2. Then in Chapter 3 an overview of current flow verification is presented. After that, our proposed method and detailed algorithm are explained in Chapter 3 and Chapter 4. To demonstrate the correctness of importance of this method, two case studies are constructed and explained in Chapter 5. Chapter 6 summarizes our work and talk about our future works. All the flow specifications and protocols used in implementation process will be explained in Appendix.

CHAPTER 2

BACKGROUND

2.1 Representations of SoC Protocols

In software engineering field, there are two approaches for 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 need 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 is developed for verification purpose. Usually it requires manual translation of an informal description to a formal description, which consumes large amount of time and effort as modern system involves massive amount of complicate specifications. [21] introduces a tool that translate 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 thesis, we use the words "protocol" and "flow" interchangeably.

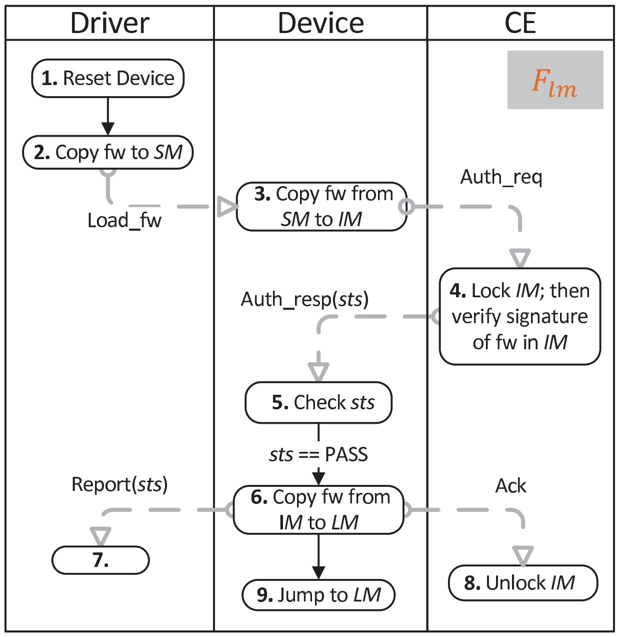


Figure 2.1. A graphical representation of a SoC firmware load protocol [2].

Figure 2.1 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) [22].

To start this protocol, the Driver resets Device and copies the needed firmware to a place in System Memory (SM) and notice the Device to load it. With the location of firmware provided from Driver, Device can retrieve firmware to Isolated Memory (IM) and sends 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 replys 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 jump 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 filed is sequence diagrams. It represents the life cycle of an processor and the interactions between them. Commonly used sequence diagram include UML sequence diagrams, message sequence diagrams and live sequence charts [21].

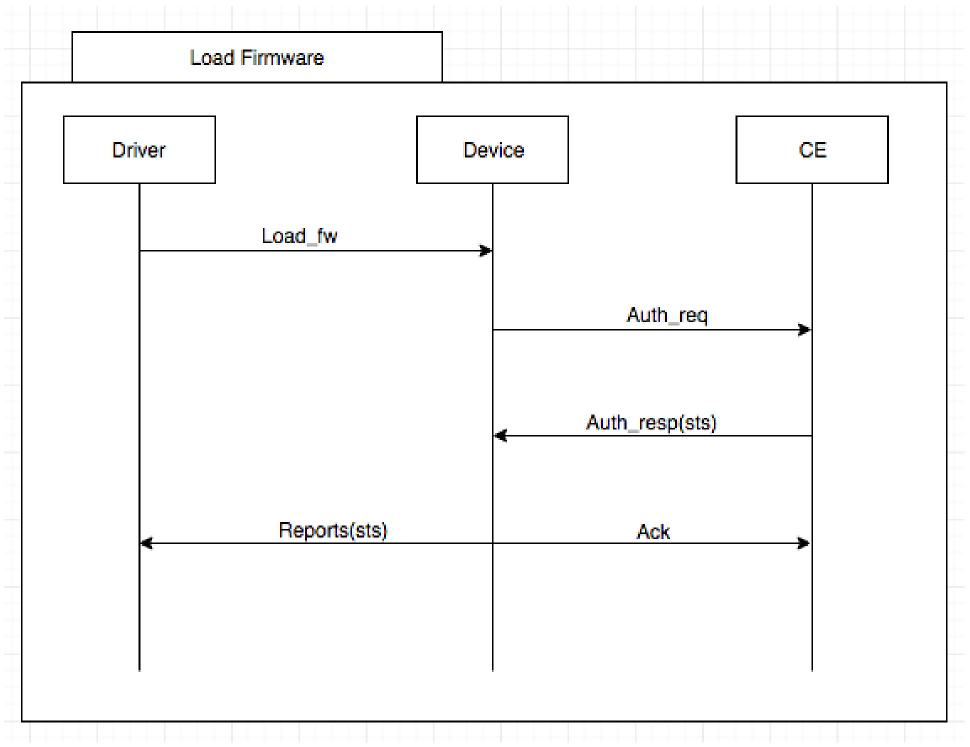


Figure 2.2. Protocol in Figure 2.1 represented in graphical live sequence chart

Figure 2.2 shows the live sequence chart representation of the protocol in Figure 2.1. In this graph, we can clearly see the relative time and content of communications between each components. Unlike the BPMN format in Figure 2.1, sequence diagrams is more abstract as the internal activities of components are not shown.

2.2 Labeled Petri-Nets

This thesis 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 thesis 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, LPN is more machine friendly, and can be analyzed using mathematical techniques and tools.

Formally, an LPN is a tuple (P, T, s_0, E, L) where

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

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, and it is also referred to as a state of a LPN. The initial marking s_0, the set of initially marked places, is also the initial state of the LPN.

The communication protocol shown in Figure 2.1 is represented by the LPN shown in Figure 2.3. This format, compared to sequence diagrams, is even more abstract. It removes all the structure information of a system, and represents only communications 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 has a form of $\langle \texttt{src}, \texttt{dest}, \texttt{cmd}, \texttt{addr} \rangle$ where `cmd` is a command sent from a source component `src` to a destination component `dest`, and `addr` is the address related to the command , this can be served as an unique id of the request in some situation. The protocol presented in Figure 2.3 used the format of $\langle \texttt{src}, \texttt{dest}, \texttt{cmd} \rangle$. Here the `addr` is ignored as the command does not have any address related. In the original protocol specification, the places without outgoing edges are *terminals*, which indicate termination of protocols

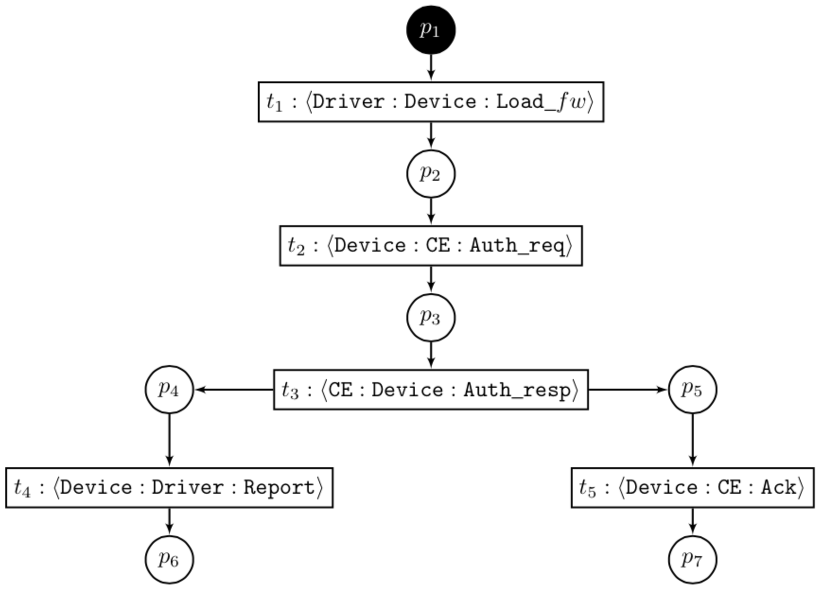

p_1
t_1 : ⟨Driver : Device : Load_fw⟩
p_2
t_2 : ⟨Device : CE : Auth_req⟩
p_3
t_3 : ⟨CE : Device : Auth_resp⟩
p_4
p_5
t_4 : ⟨Device : Driver : Report⟩
t_5 : ⟨Device : CE : Ack⟩
p_6
p_7

Figure 2.3. LPN formalization of protocol in Figure 2.1

represented by the LPNs. The initial marking is $s_0 = \{p_1\}$. In this LPN model, only the communication portion of the protocol specification is represented while the computation portion is ignored.

The operational semantics of a LPN is defined by transition executions. A transition can be executed after it is *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 $enabled(s_i)$. Execution of $t \subseteq enbled(s)$ 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

$$t_0 \; t_1 \; t_2 \; .....t_i \; ...$$

results in a sequence of events

$$e_0 \ e_1 \ e_2 \ .....e_i \ ..., \text{such that } \forall i \geq 0, t_i \in enabled(s_i) \wedge s_{i+1} = t_i(s_i)$$

Therefore, information exchanges among components in a design can be modeled by sequences of LPN transition executions.

CHAPTER 3

FLOW GUIDED TRACE INTERPRETATION

In this chapter, we describe a trace analysis method where the observed signal traces are interpreted at the level of system protocol specifications. In general, the trace analysis can offer debuggers a structured view of communications among the IP blocks during the SUD execution by deriving the types and numbers of system flows activated during System Under Debug (SUD) executions from the observed signal traces.

We formalize the trace interpretation problem in terms of labeled Petri-Nets, and discuss algorithms to address the problem. For pedagogical reasons, here we assume full observability of all hardware signals involved in the flow events. In the next chapter we extend the approach to consider partial observability.

3.1 Post-silicon Trace Analysis

In a typical validation setting, the SUD 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, 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 system-level protocols in order to identify the set of protocol 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 generates a set of possible system flow execution scenarios, which is defined in next section. A flow execution scenario indicates that at a certain point of SUD execution, what types of flows and the number of instances of a particular flow are activated and their corresponding current states.

The observed traces may help to identify problems in the protocols, *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 protocol, *i.e.*, a trace inconsistent with any possible interleaving of the protocol executions. Identifying these problems involves significant human expertise, and can often take days to weeks of effort. The trace analysis method and algorithm presented in this chapter intends to address that hurdle.

3.2 Flow Execution Scenarios

The set of system flows in LPN donates $\vec{F}$. A *flow execution scenario* is defined as a set $\{(F_{i,j}, s_{i,j})\}$ where $F_{i,j}$ is the jth instance of flow $F_i \in \vec{F}$, and $s_{i,j}$ is a state of $F_{i,j}$. A flow execution scenario indicates the set of protocols and the number of instances of a particular protocol are activated and their corresponding current states. It represents a system state during system execution abstracted on system flow specificatioins. From debugger's point of view, communication protocols can be related. For example, a firmware loading protocol always happens before a firmware execution protocol. If a firmware execution protocol happens before firmware loading protocol, that possibly indicates an error in the system implementing such portocols. This information can be used as an assertion during the debug process. For this purpose, flow execution scenario also represents the partial order relations that define the relative orderings between initiation and termination of different flow instances. This relation can provide helpful information for more efficient debug.

Since we assume full observability, we view an *observed trace* $\rho = e_1 e_2 \dots e_n$ as a sequence of flow events. Let

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

be a function to decide if event e can be admitted by flow instance $F_{i,j}$ in state $s_{i,j}$. The function returns the corresponding new state if event e can be admitted, elsewise it returns $\emptyset$. This function is used in the trace analysis algorithm later in this chapter.

Given an observed trace ρ, the goal of trace interpretation is to construct a set of candidate flow execution scenarios whose execution can create the sequence of events in ρ. In other words, ρ 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 ρ is successfully mapped to some flow instance, we can say that ρ 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 *scen*, the following two conditions hold.

1. For each $(F_{i,j}, s_{i,j}) \in scen$, $accept(F_{i,j}, s_{i,j}, e_h) = \emptyset$, and

2. For each $F_i \in \vec{F}$, $accept(F_i, init_i, e_h) = \emptyset$.

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

3.3 Flow Guided Trace Interpretation Algorithm

Given an observed flow trace ρ and the set $\vec{F}$ 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 *Scen*) of a set of candidate flow execution scenarios compliant with each prefix of ρ. At each iteration, for each event e_h in the observed trace, the algorithm updates *Scen* by either updating the state of a member of *scen* or by initiating a new flow instance for each $scen \in Scen$ with respect to e_h in every possible way. If e_h cannot be accepted by any existing or new flow instances in *Scen*, this indicates that trace ρ is inconsistent withe *Scen*. If event e_h is inconsistent with all existing execution scenarios, then the algorithm reports that the trace is `inconsistent` with *Scen*.

Given a trace of flow events $\rho = e_1 e_2 \ldots e_n$, the trace interpretation algorithm starts with an empty set of of flow execution scenario $Scen = \emptyset$. Then, for each e_h where $1 \leq h \leq n$ starting $h = 1$, and for each $scen \in Scen$, the following two steps are performed.

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

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

```
Create an empty scenario $scen$
$Scen = \{scen\}$
foreach $h,\ 1 \leq h \leq n$ do
    $found \leftarrow$ true
    $Scen' = \emptyset$
    foreach $scen \in Scen$ do
        foreach $(F_{i,j}, s_{i,j}) \in scen$ do
            $s'_{i,j} \leftarrow accept(F_{i,j}, s_{i,j}, e_h)$
            if $s'_{i,j} \neq \emptyset$ then
                Let $scen'$ be a copy of $scen$
                $scen' \leftarrow scen' - (F_{i,j}, s_{i,j})) \cup (F_{i,j}, s'_{i,j})$
                $Scen' \leftarrow scen' \cup Scen'$
                $found \leftarrow$ false
            end
        end
        foreach $F_i \in \vec{F}$ do
            create a new instance $F_{i,j+1}$
            $s'_{i,j+1} \leftarrow accept(F_{i,j+1}, init_{i,j+1}, e_h)$
            if $s'_{i,j+1} \neq \emptyset$ then
                Let $scen'$ be a copy of $scen$
                $scen' \leftarrow scen' \cup (F_{i,j+1}, s'_{i,j+1})$
                $Scen' \leftarrow scen' \cup Scen'$
                $found \leftarrow$ false
            end
        end
    end
    if $found ==$ true then
        return $\{Scen, e_h\}$
    end
    $Scen = Scen'$
end
return $\{Scen, \epsilon\}$
```

Algorithm 1: Check-Compliance($\vec{F}$, ρ)

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

Based on the above discussion, the trace interpretation algorithm generates two possible results:

- $\{Scen, \epsilon\}$ when ρ is compliant with the flow specification $\vec{F}$ where $Scen$ is a set of flow execution scenarios, each of which is derived from the observed trace, and ϵ is an empty event indicating non-existence of inconsistent events.
- $\{Scen, e_h\}$ when inconsistent event occurs where $Scen$ is a set of partially derived scenarios and e_h is the corresponding inconsistent event. This result provides valuable information for debuggers to root cause system failures.

3.4 Illustration

To illustrate the basic idea of the trace analysis algorithm, consider the system flow shown in Figure 2.3. Let F_1 denote such flow. Suppose that the following flow trace is abstracted from an observed flow trace.

$$t_1\ t_2\ t_1\ t_2\ t_3\ t_3\ t_4\ t_5\ t_5\ t_4 \ldots \tag{3.1}$$

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 $accept(F_{1,1}, init_1, t_1) = p_2$ 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 $accept(F_{1,1}, p_2, t_2) = p_3$. 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.

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

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:

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

Next, processing the following eventt_5 leads to execution scenarios derived from those shown above :

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

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

$$\begin{gathered}\{(F_{1,1}, \{p_6, p_7\}),\ (F_{1,2}, \{p_4, p_7\})\}\\ \{(F_{1,1}, \{p_4, p_7\}),\ (F_{1,2}, \{p_6, p_7\})\}.\end{gathered} \tag{3.2}$$

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 protocol specification activated and numbers of flow instances of each protocol. Moreover, depends on the correlation between protocols, together with recorded order of each flow instance's start and finish time, debugger can judge if the system functions correctly.

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

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

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

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

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

CHAPTER 4

TRACE ANALYSIS UNDER PARTIAL OBSERVABILITY

In hardware that implements a given system flow specification, a flow event is assumed to be implemented as an event or a sequence of events on a set of hardware signals. However, in post-silicon debug, it is impossible to observe all or a large number of signals.

In this chapter, the analysis algorithm is extended by adapting trace analysis method presented in the previous chapter 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.

4.1 Mapping Individual Signal Events to Flow Events

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 are 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 observable signal traces.

Consider the following example for mapping individual signal events to flow events. Suppose that 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.

$$e_1 : \; abc$$

$$e_2 : \; \bar{a}bc$$

$$e_3 : \; a\bar{b}c$$

Suppose that only signals b and c are observable, and we obtain the following trace

$$\rho = bc\ bc\ \bar{b}c$$

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, $\bar{a}bc$. The first signal event abc can be mapped to e_1. While $\bar{a}bc$ 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 $\bar{b}c$ can be mapped to $a\bar{b}c$ and $\bar{a}\bar{b}c$, respectively. In this case $a\bar{b}c$ is mapped to e_3. On the other hand, $\bar{a}\bar{b}c$ can not 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 ρ is abstracted to four possible flow traces: $\{e_1, e_2\} \times \{e_1, e_2\} \times \{e_3\}$.

4.2 Mapping Sequences of Signal Events to Flow Events

Next, we consider the case where a flow event is implemented by a sequence of signal events when the flow event models 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 with handshake protocol consists of two steps: (1) component A sets the valid bit to 1 together with the command, (2) and component B sets the acknowledgement signal to 1.

Now suppose that two flow events are implemented by two sequences of signal events as defined in the list below.

$$e_4 : \quad abc\ \bar{a}bc$$

$$e_5 : \quad abc\ abc\ abc\ \bar{a}bc$$

Again, assume that a is not observable, and suppose that an observed trace on signals b and c is obtained as shown in (4.1).

$$\rho = bc\ bc\ bc\ bc \tag{4.1}$$

The mapping function takes the signal trace ρ, index of next signal event i and mapping table of flow events and signal events as input, and returns a set of pairs (e, i') where e is a flow event mapped from possible sequence of signal events starting from signal event at index i and i' is the index of next signal event to be considered. Starting from the current signal event with index i, consider all prefixes of increasing length up to Max, map each $pref(\rho, k)$ to a flow event. If successful, return (e, i') such that $i' = i + k$. Here Max is the length of the longest sequence of signal events that implement a flow event.

In this example, assume the index for the first signal event is 0. The given mapping relationship between the flow events and the signal events shows the length of a signal event for a flow event is either two or four. Therefore in this case value of Max's is 4. The function takes $\rho, Mapping\ table$ and i with value set to 0 as input.

Start with the first signal event $pref(\rho, 1) = bc$, it can not be matched to any flow event. Next round, the new sequence $pref(\rho, 2) = bc\ bc$ is considered, by looking up the mapping table, this can be mapped to e_4, and $i' = i+2$ is 2. After that, sequence $pref(\rho, 3) = bc\ bc\ bc$ can not be mapped to any flow event. Last, as the length of new sequence *bc bc bc bc*' reaches Max 4. This new sequence can be mapped to e_5, and the index i' of the next signal event to consider is $i + 4 = 4$. Subsequently, the function terminates and returns the set of pairs:

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

The pair $(e_5, 4)$ reaches the end of the signal traces, therefore the mapping ends. For $(e_4, 2)$, the mapping function is applied to ρ with index i changed to 2 and it returns $\{(e_4, 4)\}$.

After combining the previous result, two flow traces are derived from ρ as shown below.

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

4.3 Generalized Trace Analysis Algorithm

As shown in the previous section, more than one flow trace can be derived from a signal trace under partial observability. The number of flow traces can grow exponentially as the system complexity increases and the number of signals being able to be observed remains the same. Flow traces are usually very long and thus requires a lot of time to interpret, the large number of flow traces can drastically increase the analysis time. To address this problem, our research combines trace abstraction with trace interpretation and a new generalized algorithm is presented next. The detailed algorithm pseudocode is shown in Algorithm 2.

In this generalized algorithm, multiple instances are used. In line number 2, *Scens* is a set of pairs $(Scen, i)$. Each pair represents scenario *Scen* extracted for signal trace from beginning to index i. The pairs are ordered by i in ascending order. In line number 6, $Flow\_Map$ is the mapping table between flow event and signal events, and is a set of (e, σ) where e is the flow event, and σ is the correspond sequence of signal events. In the following sections, $\sigma[i]$ is used to present ith signal event in σ. In line number 3, $Scen\_final$ is created as a set of pairs $(Scen, m)$ represent final scenario after finish processing all signal traces, together with size of the signal events. In line number 18, $Scens\_d$ is created to hold a set of pairs $(Scen, i)$ that are deleted from *Scens*

A new $Map(\rho, h, Flow\_Map)$ is created to find matching flow events from signal events start at index h. This function returns a set of pairs of K, each pair in K consists of (e, i) where e is a matched flow event, and i is the corresponding positions of next signal event to be considered.

Here the mapping algorithm uses the breadth first searching algorithm, meaning for each signal event, the algorithm considers all possible flow traces that can be mapped to

```
Create an empty scenario $scen$
$Scens = \{(scen, 0)\}$
$Scens\_final = \emptyset$
while $Scens \neq \emptyset$ do
    $get\ (Scen, i) \in Scens$
    $(E, I) \leftarrow Map(\rho, i, Flow\_Map)$
    foreach $(e, i') \in (E, I)$ do
        $Scens' \leftarrow Flow\_Analysis(\vec{F}, Scen, e)$
        if $Scens' \neq \emptyset$ then
            if $i' = |\rho|$ then
                $Scens\_final = Scens\_final \cup (Scens', i')$
            else
                $Scens = Scens \cup (Scens', i')$
            end
        end
    end
    $Scens = Scens - (Scen, i)$
    $Scens\_d = Scens\_d \cup (Scen, i)$
end
if $Scen\_final = \emptyset$ then
    return $(Scens\_d, true)$
else
    return $\{Scen\_final, false\}$
end
```

Algorithm 2: GENERALIZED-CHECK-COMPLIANCE($\vec{F}$, ρ)

```
1  $Result = \emptyset$
2  $pref = \epsilon$
3  for $i \to 0...min(Max, |\rho| - 1)$ do
4      $pref = \rho[h, h + i]$
5      foreach $(e, \sigma) \in Flow\_Map$ do
6          if $i == |\sigma|$ then
7              $flag = true$
8              foreach $k \in 0...i - 1$ do
9                  if $\sigma[k] \not\Rightarrow pref[k]$ then
10                     $flag = false$
11                     $break$
12                 end
13             end
14             if $flag == true$ then
15                 $Result = Result \cup (e, h + i)$
16             end
17         end
18     end
19 end
20 return $Result$
```

Algorithm 3: $Map(\rho, h, Flow\_Map)$

and calculate the next signal event index. When all possible flow events are extracted, the algorithm goes through each of the possible index of signal event and do the same process again. Detailed algorithm is shown in Algorithm 3. Symbol $\Rightarrow$ in line number 9 is a function means implying. For example $r_i \Rightarrow s_i$ means a fully observed signal event r_i can imply partially observed signal event s_i In line number 4, $\rho[h, h + i]$ is used as a function that returns a sequence of signal event starting from index h to index h+i in ρ

Another function $Flow\_Analysis(\vec{F}, Scen, e)$ is also used. This function takes current scenario set $Scen$, event e and flow specification $\vec{F}$ as input, and returns a new set of scenarios produced after executing event e. Details of the function are shown in Algorithm 4.

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

Algorithm 4: $Flow\_Analysis(\vec{F}, Scen, e)$

This algorithm terminates in two conditions: (1) When all signal events are processed and the algorithm returns ($Scen\_final$,$false$). $Scen\_final$ is the set of final scenarios, and $false$ indicates no existent of inconsistent event. (2) When inconsistent event happens, the algorithm returns the set of deleted scenarios $Scen\_d$ where inconsistent event happened and boolean with value $true$ indicates existent of inconsistent event.

4.4 Difficulties and Solutions

It is clear from above that a partial trace is viewed as a set of flow traces, and Algorithm 1 can be suitably extended to work with flow traces to obtain the set of candidate flows. However, applicability of the algorithm in practice can be gated because the number of potential flow execution scenarios generated under partial observability may be enormous. Note that this is not a limitation of the algorithm; if the observability of critical events is poor there simply *are* too many flow execution scenarios compliant with the observed trace.

Nevertheless, we need to address the issue to make trace interpretation (whether automatic or not) practicable. There are two potential approaches: (1) better selection of post-silicon trace observability, and (2) use of system insights during validation. Trace signal selection itself is an important and orthogonal topic [23, 24], and finding an automated way of signal selection algorithm can be one of our future research direction. We briefly describes impact of signal selection and how the debuggers' insights of a system's architecture can help to address the complexity issue in the trace interpretation in next two sections.

4.5 Trace Signal Selection

Trace signal selection in this research is done manually by the debugger. This is process is labor-intensive and heavily relied on the debugger's insights. During the experiment conducted by this research, different sets of signals was selected and tested on the algorithm. The best sets of signals can be selected by comparing the numbers of different scenarios returned after analyzing the signal traces. This ad-hoc method, even though the debuggers' knowledge is of great help for debugging process, can not guarantee the quality of the selected trace signals. For bugs that are not expected, it is nearly impossible to select signals that are related to them during the design phase. Therefore, we need to have a least some trace signals that are selected in an automated manner without debuggers' insights to achieve effective bug detection.

Most of the current signal selection algorithm are done in low-level and using SSR (State Restoration Ratio) as their metric. SRR measures signal selection algorithm by counting the number of design states able to be reconstructed from the signals observed. However, as explained in [25], because SSR treats all signal equally and thus always favor big arrays, it may not be very helpful to restore useful system states for our case. We need a tool that can restore the maximum states in system level and thus lead to less scenarios.

[25] proposed an interesting algorithm based on Google's PaperRank system. This algorithm rank each signal based on their connectivity to other instances and chose those most valuable signals. However, this algorithm does not support system level assertions hence may not be able to chose the best signals to restore system level state in our case. [26] mentioned another way of signal selection in system level using linear program formulation. This method focus on communications between IPs and try to maximize the coverage of each protocol messages. Combining this method on our system and evaluate its performance is one of our future jobs.

4.6 Interactive Trace Interpretation

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.

This approach can be flexible in that it allows a debugger to analyze the observed traces in a trial-and-error manner if the precise knowledge of the system (micro-)architecture is hard to come by. For instance, the debugger might initially make a very restricted assumption on how the SUD 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 SUD to be observed. This idea, closely related to trace signal selection, is critical for post-silicon validation, and a detailed discussion can only be presented in a separate paper.

CHAPTER 5

CASE STUDIES

This chapter demonstrates the uses of our proposed algorithm on two different models. In the first section, the trace analysis algorithm is applied to a transaction level model of a simple SoC built on the simulator GEM5. In the next section, a more detailed RTL model is conducted to further test the efficiency and effectness of our algorithm.

5.1 A Transaction-Level Model of a Simple SoC in GEM5

To determine the efficiency of the trace analysis method for a realistic example, a transaction level model of a SoC is constructed within the GEM5 environment [20]. The GEM5 simulator is a modular platform for computer-system architecture research, encompassing system-level architecture as well as processor micro-architecture. This SoC model, as shown in Figure 5.1, consists of two ARM Cortex-A9 cores, each of which contains two separate 16KB data and instruction caches. The caches are connected to a 1GB memory through a memory bus model. The memory bus works as a system agent that is in charge of routing messages to maximize the system parallelization. Inside of the SoC, components communicate with each other by sending and receiving various request and response messages through links. In order to observe and trace communications occurring inside this model during execution, monitors are attached to links connecting the components. These monitors record the messages flowing through the links they are attached to, and store them into output trace files.

In this SoC design, there are nine interfaces in total and each interface is attached with a monitor. By combining the information from all of the nine communication monitors,

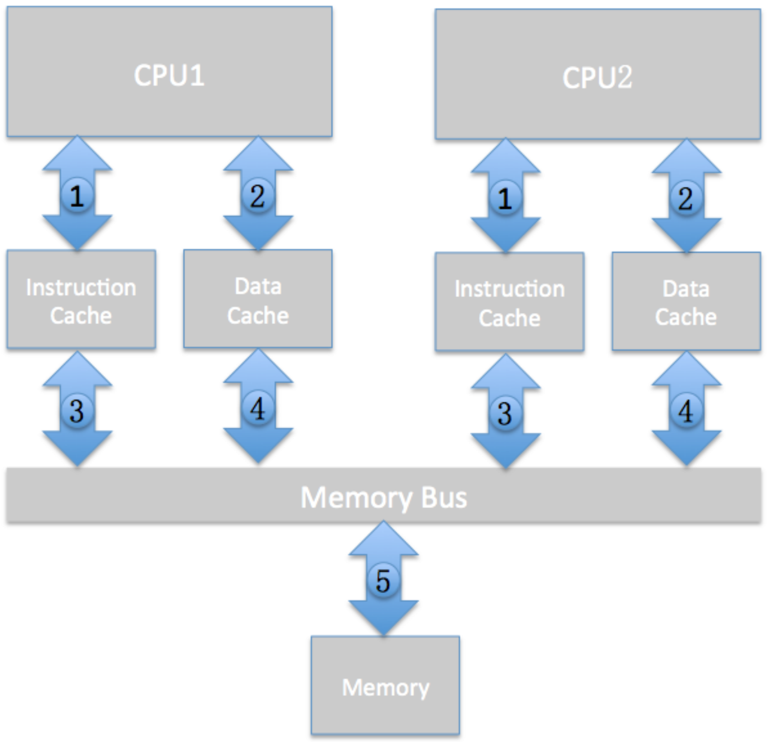


Figure 5.1. Structure of the SoC TLM

the trace of the communication activities inside the SoC can be obtained. As a virtualized SoC platform, GEM5 has three types of accesses supported by its communication interface: timing, atomic and functional. The type of access decides the time to process a request. Timing accesses is the most detailed one, it reflects the simulator's best effort for realistic timing and includes the modeling of queuing delay and resource contention. Once a timing request is successfully sent, at some point in the future the device that sent the request will get a response [27]. Atomic access is a faster access compared to timing request with no delay. This accesses are used for fast forwarding and warming up caches and return an approximate time to complete the request without any resource contention or queuing delay. When an atomic access is sent, the response is provided when the function returns [27]. Functional accesses are used for loading binaries, examining and changing variables in the simulated system, and allowing a remote debugger to be attached to the simulator. Those functions are not used for our simple SoC. In our experiment, we specified all the communications to be timing access so the system has the most realistic timing.

For this model, we consider the flow specifications describing the cache coherence protocols supported in GEM5 that is used to build the model in Figure 5.1. The GEM5 cache coherence protocols can be found at [27]. These protocols and their corresponding LPN descriptions can also be found in Appendix A. These flow specifications describe data/instruction read operations and data write operations initiated from CPUs. Each CPU has three protocols implemented, one read and two write protocols. Since there are two CPUs, there are six flows in the model.

We wrote one simple concurrent programs with two threads, one for each CPU to exercise the flows. The program assigned to CPU1 reads one file three times and does some modifications to the content of the file. CPU2's does the same read and write functionalities to the same file, the only difference is that CPU2 modifies the file first and then perform the reading function.

After this model is executed with the simple concurrent program, the trace analysis is applied to traces with different observabilities collected from this model. The runtime results are shown in Table 5.1. The first column shows the results from analyzing the trace with the full observability, while the next three show the result from analyzing traces with different partial observability assumptions.

In the first experiment, full observability is assumed. After the SoC model finishes executing the program, there are totally 343581 messages collected in the trace file. Not all of the messages are relevant to the flow specification as many are used by GEM5 to initialize its simulation environment. After removing those irrelevant messages, the number of messages in the trace file is to reduced to 121138.

The time taken to remove the irrelevant messages from the trace is negligible. The total runtime and the peak memory taken by the trace analysis algorithm on the reduced trace are 3 seconds and 12MB, respectively. Only one flow execution scenario is extracted, and

	F-Obs.	P-Obs. No Amb.	P-Obs. Amb. 1	P-Obs. Amb. 2
Time	3	2.78	896	< 1
Mem	12	10	420	9

Table 5.1. Runtime Results of Trace analysis. Time is in seconds and memory usage is in MB.

Table 5.2. The number of flow instances derived by the trace analysis with the full observability.

Flows	#Instances
CPU1 Data Read	17582
CPU1 Instruction Read	4002
CPU1 Write	3370
CPU2 Data Read	17386
CPU2 Instruction Read	3955
CPU2 Write	3308

Table 5.2 shows the number of flow instances contained in that scenario for the six flows describing cache coherent operations initiated from both CPUs.

In the second experiment, partial observability is taken into account with the four monitors attached to the links between two CPUs and their caches are disabled. Then, the trace is generated by the remaining five monitors from the SoC model executing the same program. The new trace contains 15089 messages. Similarly, only one flow execution scenario is extracted, and the numbers of the flow instances contained in that execution scenario are shown in Table 5.3. From these results, the numbers of the flow instances are dropped significantly compared to the results extracted from the trace with the full observability as shown in Table 5.2. This difference is due to that some communications occurred in the

Table 5.3. The number of flow instances derived by the trace analysis with certain monitors disabled.

Flows	#Instances
CPU1 Data Read	829
CPU1 Instruction Read	169
CPU1 Write	82
CPU2 Data Read	803
CPU2 Instruction Read	190
CPU2 Write	83

system when executing the program involve the CPUs and their corresponding caches only, and the traffic on the links between the CPUs and their corresponding caches is not observable. Therefore, the instances of the flow specifications characterizing these communications do not exist in the trace. In other words, all extracted flow instances in Table 5.3 characterize the communications that pass through the memory bus in the system model. The runtime and memory usage as shown in the third column in Table 5.1 are similar to those for analyzing the trace of the full observability.

In the third experiment, further partial observability is taken into consideration. In this experiment, only the five links involving the memory bus are still considered. However, an assumption is made that all messages passing the same link are not distinguishable due to the limited observability. The monitors are modified such that whenever a message is captured on one of the links, it dumps a set of messages passing through the same link into the trace file. Therefore, each line of the trace file corresponds to a set of messages. After applying the trace analysis to this trace, a total of 13944 flow execution scenarios are extracted. This large number, compared to the results from the first two experiments, is due to the ambiguous interpretation of the messages with limited observability.

The whole experiment takes about 15 minutes and 420 MB to finish as shown in column 4 in Table 5.1, significantly higher than the numbers for analyzing traces where there is no ambiguity in the observed messages. This is due to the fact that a trace of ambiguous messages is in fact a set of traces of messages with full observability, which lead to large numbers of execution scenarios either during or at the end of the analysis. In this experiment, the peak number of execution scenarios encountered during the analysis process is 70384, many of which are invalid and removed eventually. However, controlling the number of intermediate execution scenarios found during the trace analysis is critical in order for the analysis to be tractable. Here, insights from validators could help, but are not used in this experiment.

As shown above, the ambiguous interpretation of messages can lead to large numbers of intermediate and final execution scenarios, which not only make the trace analysis more time consuming but also make it difficult to gain insightful understanding from the derived execution scenarios. Careful selection of what to observe may have big impact on results from the trace analysis. In this last experiment, we relax the assumption made in the previous experiment such that the messages passing each link are partitioned into two groups, one for read operations and one for write operations. Similar to the assumption made in the previous experiment, messages in the same group are assumed to be non-distinguishable. The monitors are modified accordingly such that they output all messages in the same group into the trace file if an event from that group is captured. After the trace analysis on this new partially observed trace is finished, only one execution scenario is derived where the distribution of the numbers of flow instances is the same as those shown in Table 5.3. The peak number of execution scenarios encountered during the trace analysis is 4. The total runtime and memory usage are negligible as shown in the last column in Table 5.1. Compared to the results from the previous experiment, the precision and the performance of the trace analysis are improved dramatically as a result of careful selection of observable messages.

One problem we countered during the experiment was that how GEM5 supports shared memory multi-threaded program execution is unclear and each CPU has its separate, non-interleaving address space. Therefore, no data are shared between caches in this test. Furthermore, GEM5 does not support true concurrency. When there are two programs running on the CPUs, GEM5 alternates the instruction executions between the two CPUs. Therefore, no matter how many time the multithread program runs, the observed behaviors are always deterministic. To simulate asynchronous concurrency with the interleaving semantics, those two simple programs are instrumented with pseudo-blocking commands, one placed before each statement. A pseudo blocking command includes a random number generator that returns either 0 or 1 and a loop that only exits when the returned random number is 0. To address the above limitations, we advances our experiment into a more detailed cycle accurate RTL model of SoC. Details of this new model is explained in next section.

5.2 A Cycle-Accurate RTL Model for a Simple SoC

The model in the previous section is done in very high abstraction level and trace abstraction method introduced in Chapter 4 is not fully used. In this section, we construct a more detailed RTL model for a simple Soc. For this model the trace information is collected at bit level, thus an extra translation step is required. This model is cycle accurate, and simulates the SoC behavior that is more accurate than that TLM.

5.2.1 Model Implementation

Due to the lack of similar research, we cannot find any existing model with accurate flow specifications, therefore this model is built from scratch. Here the flow specifications from GEM5 are slightly modified and applied to the RTL model shown in Figure 5.2. This model consists of two CPU models, each with its own 1KB cache. The caches are connected to a 4MB memory through a memory bus model. Currently, the CPUs are treated as test

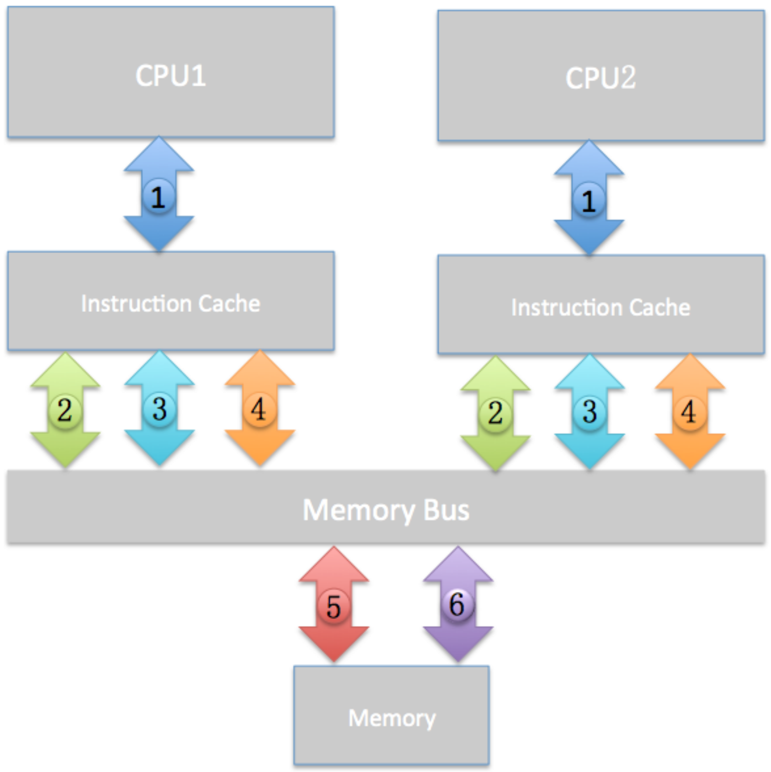


Figure 5.2. SoC platform structure.

environment where software programs are simulated to trigger various flows. This model is simulated using an open-source simulator for the VHDL language called GHDL.

To collect communication activities of the system, the test environment collects values of selected signals into an output trace file on each rising clock edge, output it into a txt trace file. GHDL itself also outputs a trace file in format of vcd (value change dump) where any value changes on all signals are collected, this format can be opened by wave viewer softwares and provide a graphical representation of changes in a recorded signal's amplitude, Figure 5.3 shows an example of trace in waveform format. The trace analysis algorithm is modified to be able to take both format of trace files as input and perform trace abstraction function on.

There are six types of interfaces shown in Figure 5.2, each contains multiple signals to ensure the correct communication behaviors of the SoC. There are two types of signals, `std_logic` that stores only one bit of value and `std_logic_vector` that stores a list of bits.

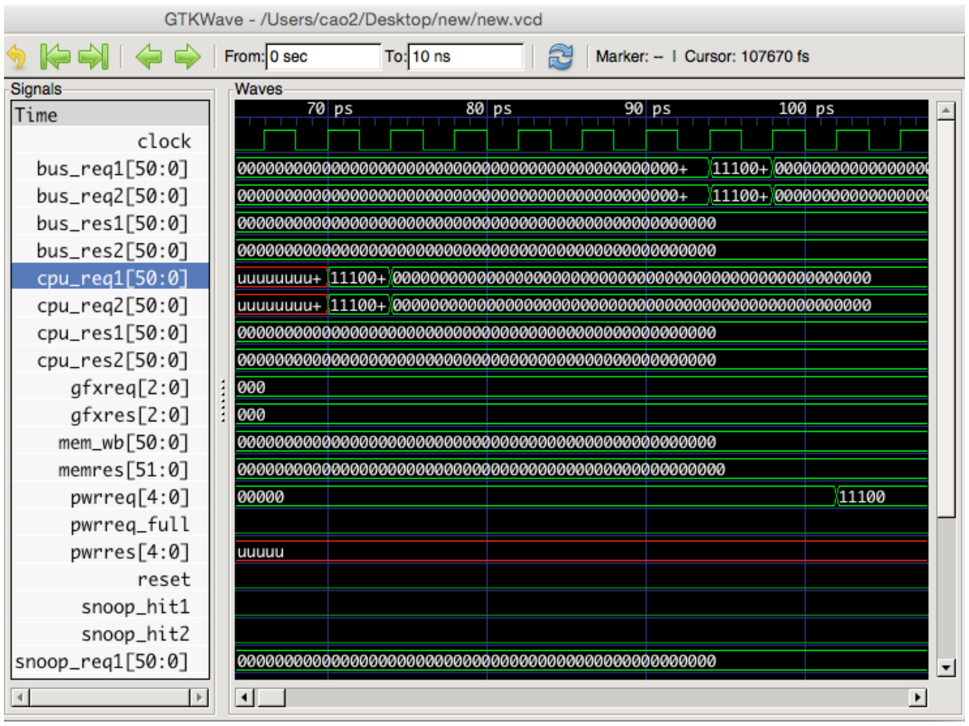


Figure 5.3. Signal trace in waveform format

This model contains two types of `std_logic_vector`, one with width of 51 that is used by interface 1, 2, 3, 4 and 5, another type of `std_logic_vector` with width of 52 that is used by interface 6. For `std_logic_vector` with 51 bits, as shown in format 1 of Figure 5.4, the most

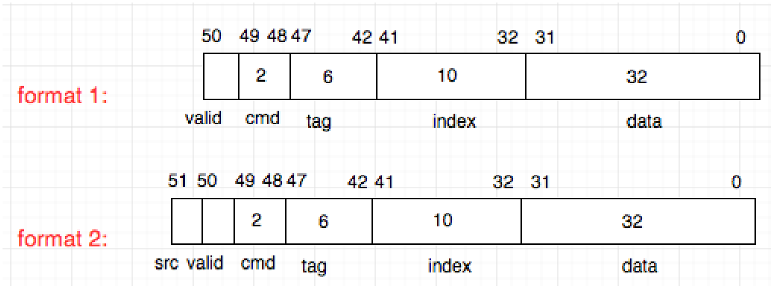


Figure 5.4. Format definition of messages

significant bit indicates validity of messages. The next two bits 49-48 indicate the command. These two bits can represent four different commands. Right now our system supports only two types of commands, we use 01 to represent read command and 10 to represent write command. Bits 47-32 represent 16 bits address, and the last 32 bits represent data. For `std_logic_vectors` with 52 bits, as shown in format 2 of Figure 5.4, the most significant bit represent source of the request. For example, if a request is from CPU1, that bit will be 1, otherwise it will be 0, the rest of the bits are the same as the first type of `std_logic_vector`.

The communication between each component are implemented by using handshake protocols to ensure no data lost. And to implement non-blocking communication, we use a first in first out (FIFO) buffer inside each component for each incoming signals. The model shown in Figure 5.2 has 6 different interfaces, here we explain each interfaces with signal names and its functionality. The first interface (1), as shown in Figure 5.5, is responsible for communications between CPU and Cache. There are three signals included in this interface, each of which is explained in Table 5.4.

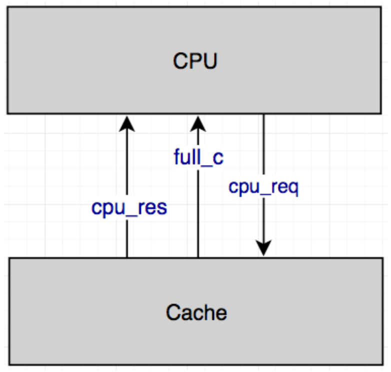


Figure 5.5. Interface definition of link 1 in Figure 5.2

Table 5.4. Signals explanations for interface 1

Signal Name	Width	Definition
cpu_req	51	cpu request from CPU to Cache
cpu_res	51	cpu response from cache to CPU
full_c	1	full indicator of *cpu_req* FIFO inside Cache

There are three interfaces between Cache and Memory Bus. Detailed structures of these three interfaces are shown in Figure 5.6. In this figure, interface (2) is used for communications about snoop function, interface (3) is used for write back function, and interface (4) is in charge of cache requests bus responses. The color of the signal name is the same as its interface color in Figure 5.2. Signals included in these three interfaces are explained in Table 5.5.

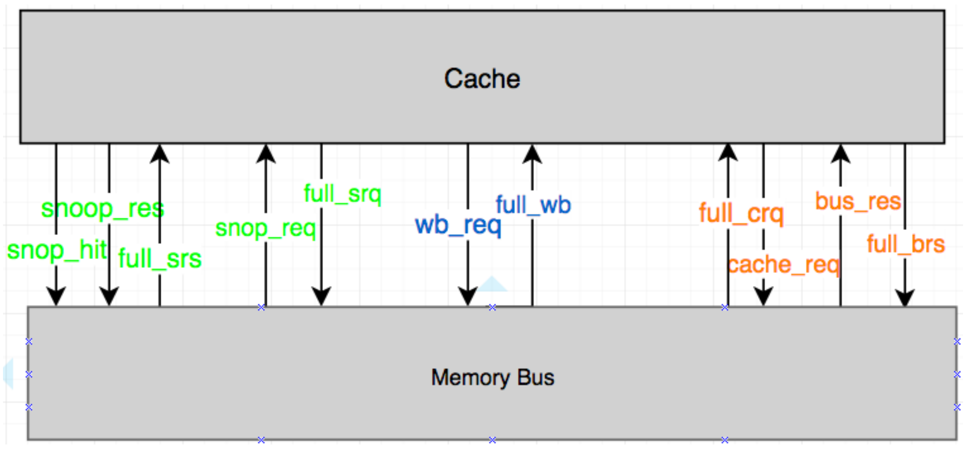


Figure 5.6. Interface definition of link 2, 3 and 4 in Figure 5.2

Two interfaces are constructed between memory bus and memory. Interface five is designed for handling data request functions, and is composed of signals in color red in Figure 5.7. Interface six is exclusively for write back functions, and is represented in color purple in Figure 5.7. Detailed explanations of each signal are explained in Table 5.6.

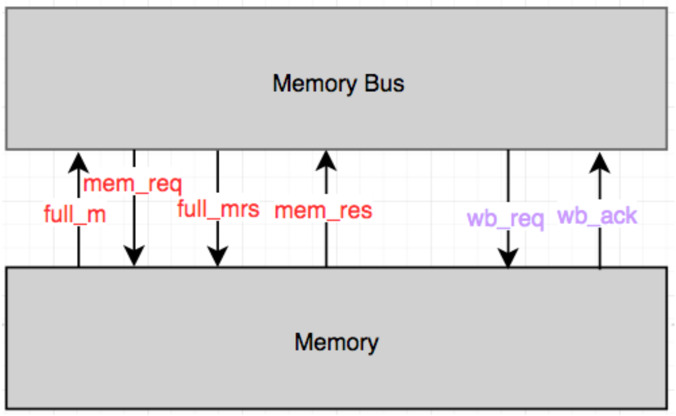


Figure 5.7. Interface definition of link 5 and 6 in Figure 5.2

This model has 6 protocols implemented. It's based on the protocols provided by GEM5 with some modification. 3 types of protocols: read, write and write back are implemented

Table 5.5. Signals explanations for interface 2, 3 and 4

Interface	Signal Name	Width	Definition
2	*snp_hit*	1	snoop hit signal
2	*snp_res*	51	snoop response sent from Cache to Memory Cache
2	*full_srs*	1	full indicator of *snp_res* FIFO inside Memory Bus
2	*snp_req*	51	snoop request sent from Memory Bus to Cache
2	*full_srq*	1	full indicator of *snp_req* FIFO inside Cache
3	*wb_req*	51	write back request from Cache to CPU
3	*full_wb*	1	full indicator of *wb_req* FIFO inside Memory Bus
4	*full_creq*	1	full indicator of *cache_req* FIFO inside Memory Bus
4	*cache_req*	51	Cache request send to Memory Bus
4	*bus_res*	51	bus response from Memory Bus to Cache
4	*full_brs*	1	full indicator of *bus_res* FIFO inside Cache

in both CPU. Write back protocol is a new protocol and can be invoked when Cache need to flush back dirty cache lines.

To test the correctness of the cache coherence protocol, we hard coded a test generator inside of each CPU. For every clock cycle, the test generator inside each CPU will randomly generate a read or write operation. In order to better active and cache coherent protocol, only first 3 bits of the 16 bits request address are random generated, while the rest of the bits are predefined. By limiting the address to a certain range, it's more likely that one CPU will request data that exists in the other CPU.

In this experiment model, full observability is assumed. After the SoC model finishes 2000 flows for each cpu, there are totally 122704 messages collected in the trace file. The trace analysis takes 10 second to finish processing all these messages and the peak memory

Table 5.6. Signals explanations for interface 5 and 6

Interface	Signal Name	Width	Definition
5	$full\_m$	1	full indicator of mem_req FIFO inside Memory
5	mem_req	52	memory request from Memory Bus to Memory, first bit indicate which cache initiate this requirement
5	$full\_mrs$	1	full indicator of mem_res FIFO inside Memory Bus
5	mem_res	52	memory response from Memory to Memory Bus
6	wb_req	51	Write back request from Memory Bus to Memory
6	wb_ack	1	acknowledgement of write back finished in Memory

used is 18MB. This test case works correctly with no inconsistent message occurred and Table 5.7 shows the number of each flow instances activated and finished from both CPUs.

To better test our system, we hard coded another simple software that performs Peterson's Algorithm inside each CPU. Detailed algorithm is displayed in Figure 5.2.1. This algorithm contains four shared variables: $flag0$, $flag1$, *turn*, and *shared*. Each CPU want to enter the critical section waits until $flag$ or *turn* has the desired value. When entering the critical section, the variable *shared* will be incremented by one. By running this algorithm N times, the final value of *shared* should be 2N, as there will be two CPUs increment *shared* N times. This value can be used to check the correctness of the system.

Table 5.7. The number of flow instances derived by the trace analysis with the full observability.

Flows	#Instances
CPU1 Read	5090
CPU1 Write	4910
CPU1 Write Back	1270
CPU2 Read	4932
CPU2 Write	5068
CPU2 Write Back	1211

```
bool flag[2]={$false, false$}
int turn
int shared
```

```
CPU0:                              CPU1:
flag[0]=true                       flag[1]=true
turn =1                            turn =0
while $flag[1] \wedge turn == 1$ do   while $flag[0] \wedge turn == 0$ do
    //busy wait                        //busy wait
end                                end
//enters critical section          //enters critical section
shared++                           shared++
flag[0]=false                      flag[1]=false
//leaves critical section          //leaves critical section
```

Figure 5.8. Peterson's Algorithm on two CPU

5.2.2 Debugging Experience

During the building of the system, our proposed analysis algorithm is used as a debugging method to locate implementation problems. Following types of errors are detected by the algorithm and fixed.

Bug one: duplicated messages

When the model was first built, each of the CPU was designed to only issue one read request and one write request. When the trace is produced and analyzed using the trace analysis algorithm. The analysis was halted and returned a set of partially executed scenarios and one inconsistent message. This error happens no matter how many time we simulate the system. Two different inconsistent messages occurred and are shown below.

$$(bus, mem, rd)$$
$$(bus, mem, wt)$$

This two inconsistent messages belongs to two different flow specifications, the only similarity is that they are both sent from the same source memory bus. To better locate the cause of this problem, we checked the system level flow event happened before the occurrence of the inconsistent message. This translated flow event from signal event shows that the inconsistent message are always exactly the same as its previous message and is always sent from memory bus.

By looking into the memory bus's structure, the root cause is discovered and fixed by resetting the memory request once it is received by memory.

Error two: incorrect command

During the debugging process, the trace analysis algorithm was halted because inconsistent message. When running the system multiple time and apply analysis algorithm on it, the inconsistent message $(membus, mem, rd)$ keep happens.

We took one specific case and try to find the possible reason. In this case, when the inconsistent message happened, only two write flow instances are activated. The write flow specification is LPN format in shown below.

$$(cpu0 : cache0 : wt)$$
$$(cache0 : bus : wt)$$
$$(bus : cache1 : snp)$$
$$(cache1 : bus : snp)$$
$$(mem : bus : wt)$$
$$(bus : cache0 : wt)$$
$$(cache0 : cpu0 : wt)$$

The first read instance is in state two (after accepting message $(cache0 : bus : wt)$), the second read instance is in state four (after accepting message $(cache1 : bus : snp)$). Because the inconsistent message is sent from memory bus. Here we conduct a set the messages that can be accepted by the current scenarios: $(bus : cache1 : snp)$ that can be accepted by read instance one and $(bus : cache1 : snp)$ that can be accepted by read instance two. By comparing these two expected messages with the inconsistent message, we see that the inconsistent message $(membus, mem, rd)$ has the same source and destination with the expected message $(membus, mem, wt)$, here we make a guess that the command may be changed for read flow instance two. This change can be made by its source memory bus. By checking the internal structure of the memory bus, we could not find any possible causes, so we went one message ahead, and check the cache structure. Inside the cache, we found out that the command bits in the snoop response is always 01, thus read. After making change on the process that handles snoop request, the problem was solved.

Error three: protocol failure

When running Peterson's Algorithm and apply our analysis algorithm on the produce trace, no inconsistent message happened and only one scenario is produced by the algorithm. However, by checking the numbers of each finished flow specifications, we found that no cache coherence protocol was activated. Based on debuggers experience, in order for Peterson's

Algorithm to work, each CPU need to constantly request data that is shared by the other CPU, therefore there has to be at least one cache coherence flow activated.

Moreover, no matter how long we run the simulation, the number of read flow keep increasing while others remain the same. This narrows the the root cause to the only while loop in the program where it keeps checking values of $flag$ to see if the CPU can enter the critical section. This means that the shared variable $flag$ and $turn$ are not holding the correct value.

As there are no existence of inconsistent messages, it is unlikely the protocol specifications is wrongly implemented, so we conclude the protocol itself may contain some loopholes. By looking at the flow events, we found out that when two CPUs require data from the same address, both CPU thinks they have the exclusive right of that data. This is because both CPU issue request the same time, they will not find the required data anywhere but the Memory, therefore both gets the data from memory and believe they own the data exclusively. The problem is solved by adding another register inside the Memory Bus to check the same request address situation. For example, when CPU0 requests an address that CPU1 just requested but have not finished yet, CPU0 will wait until CPU1 finished its flow and then precede with a snoop request to CPU1.

After that, we rerun the system and found fair amount of cache coherence protocol activated, and the program ends after small amount of time. However, when each CPU run Peterson's Algorithm for three times, the final value of $shared$ was 5, comparing to the expected 6.

By analyzing the flow traces, we could not find any error. Because this problem resides in the actual value of the data, which is not considered at all in our algorithm, it is very hard to root cause the problem. Here we propose to record the order the each flow's start and finishing time. By analyzing this we may be able to find the real problem. Another reason why recording the order of flows is beneficial is that for future research, we can add assertions about order relation into the analyzing process. This will allow us to find the error location before it cause an an consequence. The order relation assertion can be provided with debugger's insights.

CHAPTER 6

CONCLUSION AND FUTURE WORKS

This thesis presents a method for post-silicon validation by interpreting observed raw signal traces at the level of system flow specifications. The derived flow execution scenarios provide more structured information on system operations, which is more understandable to system validators. This information can help to locate design defects more easily, and also provides a measurement of validation coverage.

Due to partial observability, this approach may derive a large number of different flow execution scenarios for a given signal trace. Insights from system validators can help to eliminate some false scenarios due to the partial observability. An interesting future direction is formalization of the validators' insights using temporal logic on flows so that the validators can express their intents more precisely and concisely.

The trace analysis approach presented in this thesis needs to be iterated with different observations selected in different iterations in order to eliminate the false scenarios and to root cause system failures as quickly as possible. The observation selection and stitching signal traces of different observations together for the above goal will also be pursued in the future.

APPENDICES

Appendix A Flow specifications and protocols provided by GEM5

A.1 Protocol Specifications in Message Sequence Chart

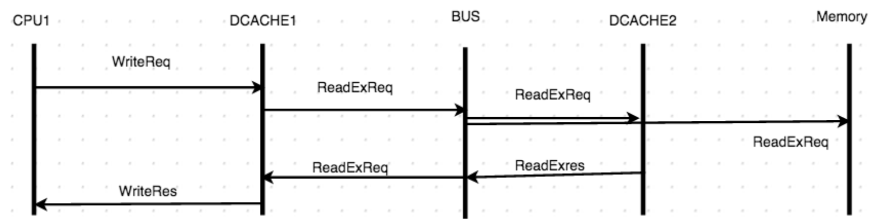


Figure A.1. Flow sequence chart of write operation when requested data is not included in Dcache. ReadExRes can also be sent from Memory if Dcache2 does not have requested data. This sequence chart is symmetric for CPU2.

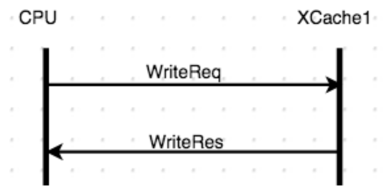


Figure A.2. Flow sequence chart of write operation when XCache has the exclusive right of requested data. XCache can be instruction cache or data cache. This sequence chart is symmetric for CPU2.

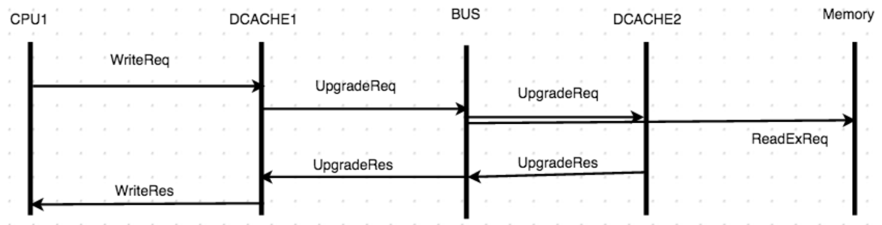


Figure A.3. Flow sequence chart of write operation when requested data is shared by another component. UpgradeRes can also be sent from Memory if Dcache2 does not have requested data. This sequence chart is symmetric for CPU2.

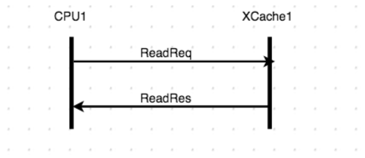


Figure A.4. Flow sequence chart of read operation when XCache has the exclusive right of requested data. XCache can be instruction cache or data cache. This sequence chart is symmetric for CPU2.

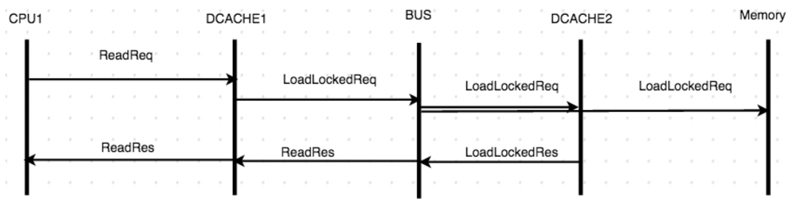


Figure A.5. Flow sequence chart of read operation when requested data is shared by another component. LoadLockedRes can also be sent from Memory if Dcache2 does not have requested data. This sequence chart is symmetric for CPU2.

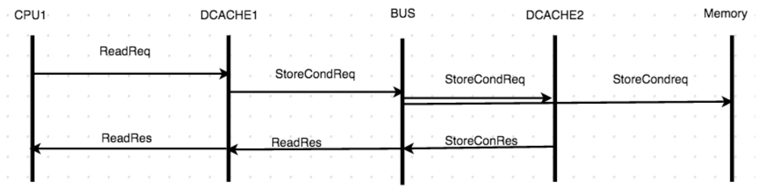
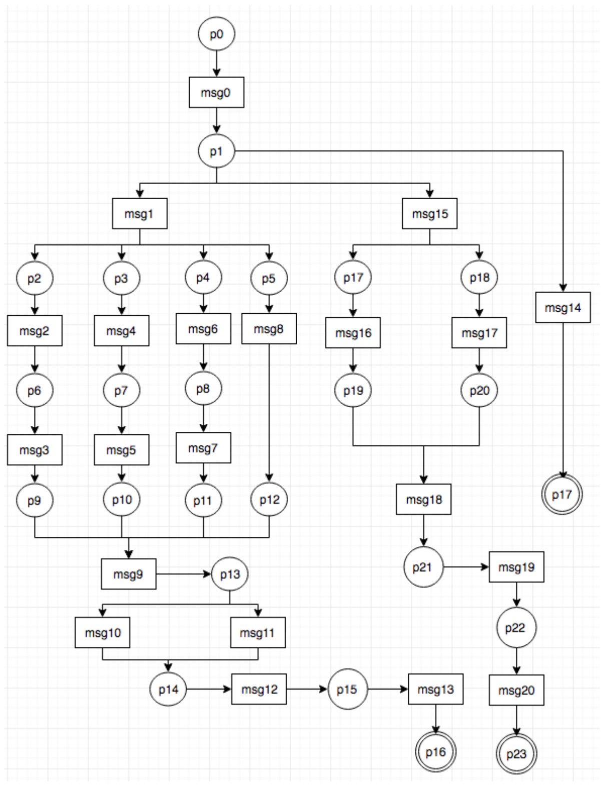


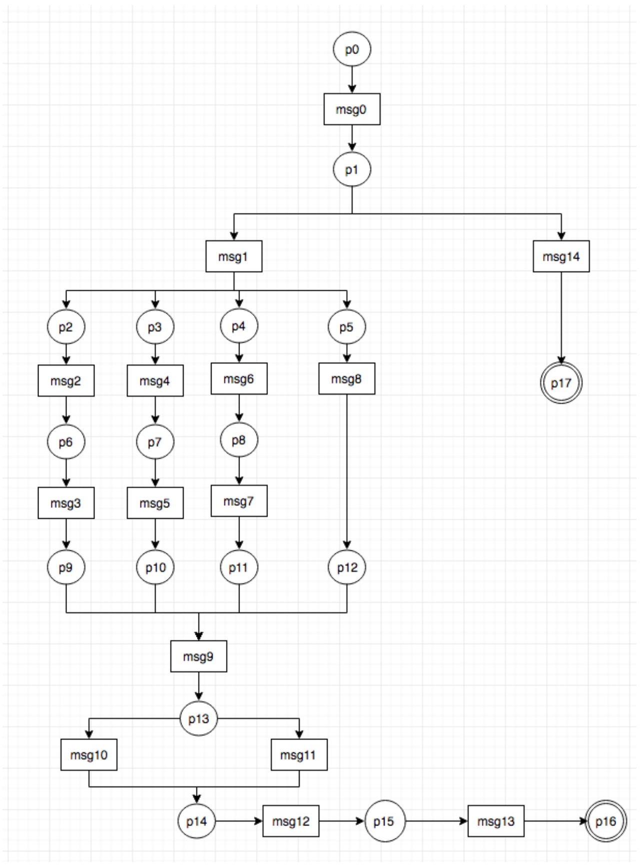
Figure A.6. Flow sequence chart of read operation when requested data is not present. StoreCondRes can also be sent from Memory if Dcache2 does not have requested data. This sequence chart is symmetric for CPU2.

A.2 Protocol Specification in LPNs



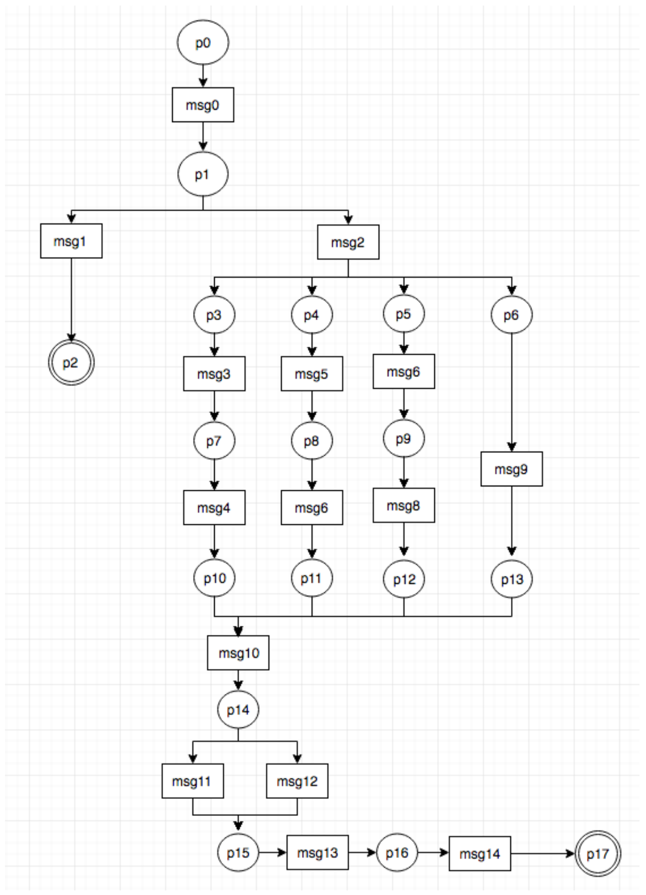
msg_0 : (	CPU1,	writeReq,	icache1)
msg_1 : (	dcache1,	readExreq ,	Bus)
msg_2 : (	Bus,	readExreq,	dcahce2)
msg_3 : (	dcache2,	readExreq,	cpu2)
msg_4 : (	Bus,	readExreq,	icahce2)
msg_5 : (	icache2,	readExreq,	cpu2)
msg_6 : (	Bus,	readExreq,	icahce1)
msg_7 : (	dcache1,	readExreq,	cpu1)
msg_8 : (	Bus,	readExreq,	Memory)
msg_9 : (	true)		
msg_{10} : (	Memory,	readExres,	Bus)
msg_{11} : (	icache2,	readExres,	Bus)
msg_{12} : (	Bus,	readExres,	dcache1)
msg_{13} : (	icache1,	writeRes,	CPU1)
msg_{14} : (	icache1,	writeRes,	CPU1)
msg_{15} : (	dcache1,	UpgradeReq,	Bus)
msg_{16} : (	Bus,	UpgradeReq,	icahce2)
msg_{17} : (	Bus,	UpgradeReq,	Memory)
msg_{18} : (	icache2,	UpgradeRes,	Bus)
msg_{19} : (	Bus,	UpgradeRes,	dcache1)
msg_{20} : (	icache1,	writeRes,	CPU1)

Figure A.7. Flow specification of a cache coherent write operation initiated from CPU1 to instruction cache. This flow is symmetric for CPU2.



$msg0$: (	CPU1,	ReadReq,	icache1)	$msg8$: (	Bus,	StoreCondreq,	Memory)
$msg1$: (	dcache1,	StoreCondreq ,	Bus)	$msg9$: (	true)		
$msg2$: (	Bus,	StoreCondreq,	icahce2)	$msg10$: (	Memory,	ReadRes,	Bus)
$msg3$: (	icache2,	StoreCondreq,	cpu2)	$msg11$: (	icache2,	ReadRes,	Bus)
$msg4$: (	Bus,	StoreCondreq,	dcahce2)	$msg12$: (	Bus,	ReadRes,	dcache1)
$msg5$: (	dcache2,	StoreCondreq,	cpu2)	$msg13$: (	icache1,	ReadRes,	CPU1)
$msg6$: (	Bus,	StoreCondreq,	dcahce1)	$msg14$: (	icache1,	ReadRes,	CPU1)
$msg7$: (	icache1,	StoreCondreq,	cpu1)				

Figure A.8. Flow specification of a cache coherent read operation initiated from CPU1 to instruction cache. This flow is symmetric for CPU2.



$msg0$: (	CPU1,	ReadReq,	dcache1)	$msg8$: (	icache1,	LoadLockedreq,	cpu1)
$msg1$: (	dcache1,	ReadRes,	CPU1)	$msg9$: (	Bus,	LoadLockedreq,	Memory)
$msg2$: (	icache1,	LoadLockedreq ,	Bus)	$msg10$: (	true)		
$msg3$: (	Bus,	LoadLockedreq,	dcahce2)	$msg11$: (	Memory,	ReadRes,	Bus)
$msg4$: (	dcache2,	LoadLockedreq,	cpu2)	$msg12$: (	icache2,	ReadRes,	Bus)
$msg5$: (	Bus,	LoadLockedreq,	icahce2)	$msg13$: (	Bus,	ReadRes,	icache1)
$msg6$: (	icache2,	LoadLockedreq,	cpu2)	$msg14$: (	dcache1,	ReadRes,	CPU1)
$msg7$: (	Bus,	LoadLockedreq,	dcahce1)				

Figure A.9. Flow specification of a cache coherent read operation initiated from CPU1 to data cache. This flow is symmetric for CPU2.

Appendix B Flow Specifications for RTL model

B.1 Protocol Specification in Message Sequence Charts

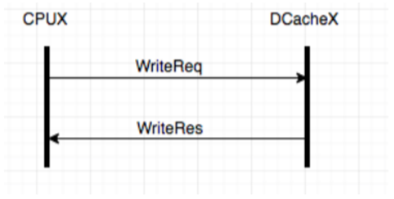


Figure B.1. CPU write when cache has exclusive right of the requested data.

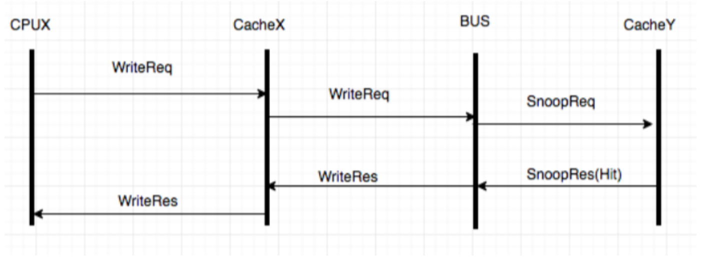


Figure B.2. CPU write when data only exist in the other CPU's cache

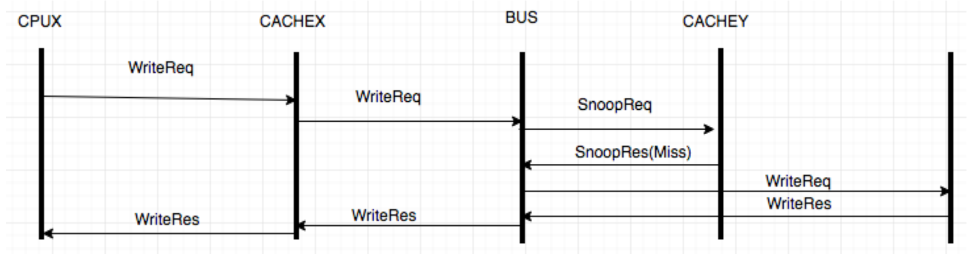


Figure B.3. CPU write when requested data only reside in Memory

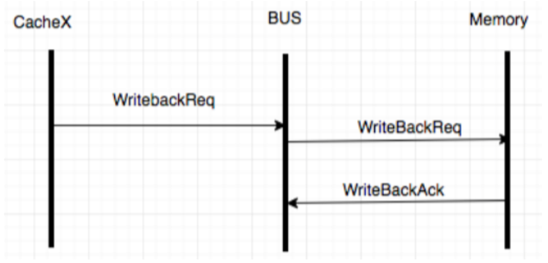


Figure B.4. Cache send write back request to Memory

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.

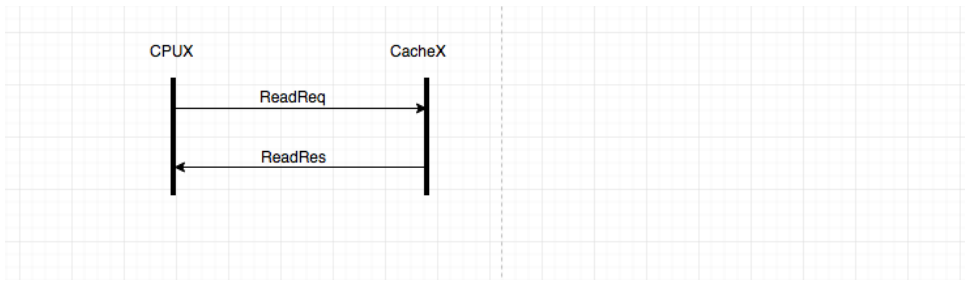


Figure B.5. CPU read when cache has exclusive right of the requested data.

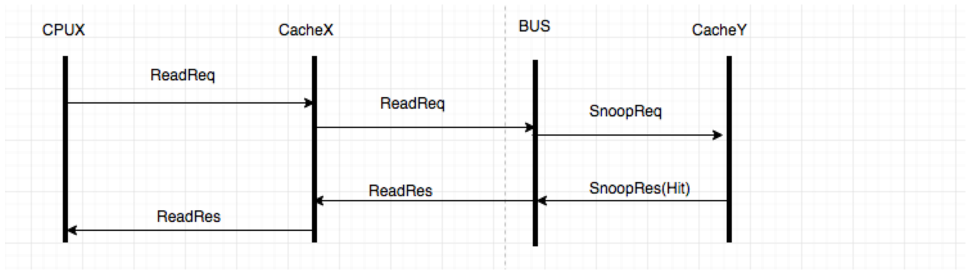


Figure B.6. CPU read when data only exist in the other CPU's cache

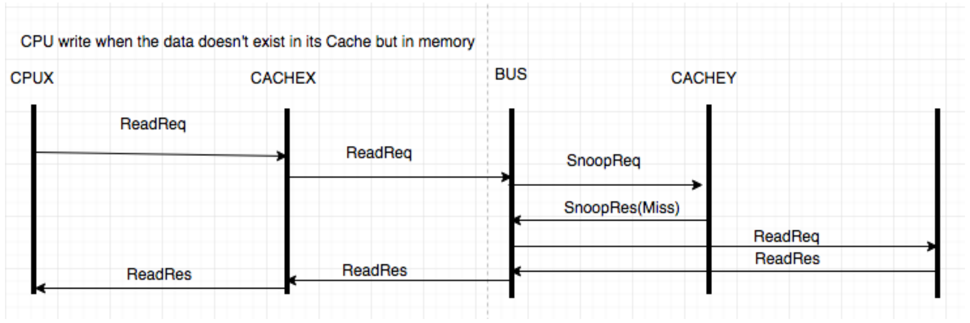
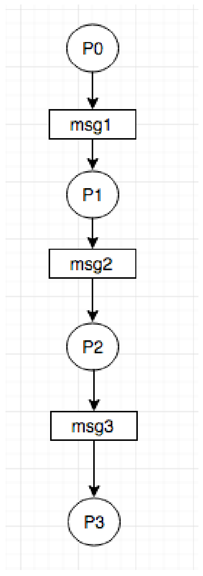


Figure B.7. CPU read when requested data only reside in Memory

B.2 Protocol Specification in LPNs

There will be 3 protocols in total: read , write and write back protocl.

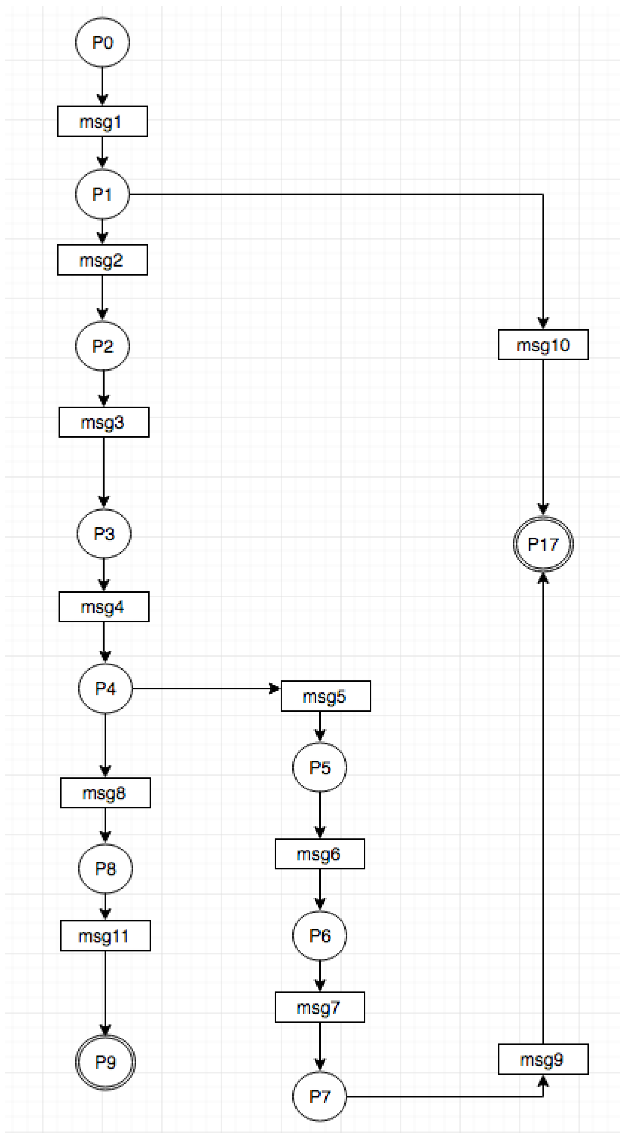
All the write operations are implemented in protocol presented in Figure B.9. When the request activate cache coherent protocol, like in Figure B.2, it will end in $state17$. The rest will end in $state9$.



$msg1$: (	Cache1,	wb ,	Bus)
$msg2$: (	Bus,	wb,	Memory)
$msg3$: (	Memory,	wb,	Bus)

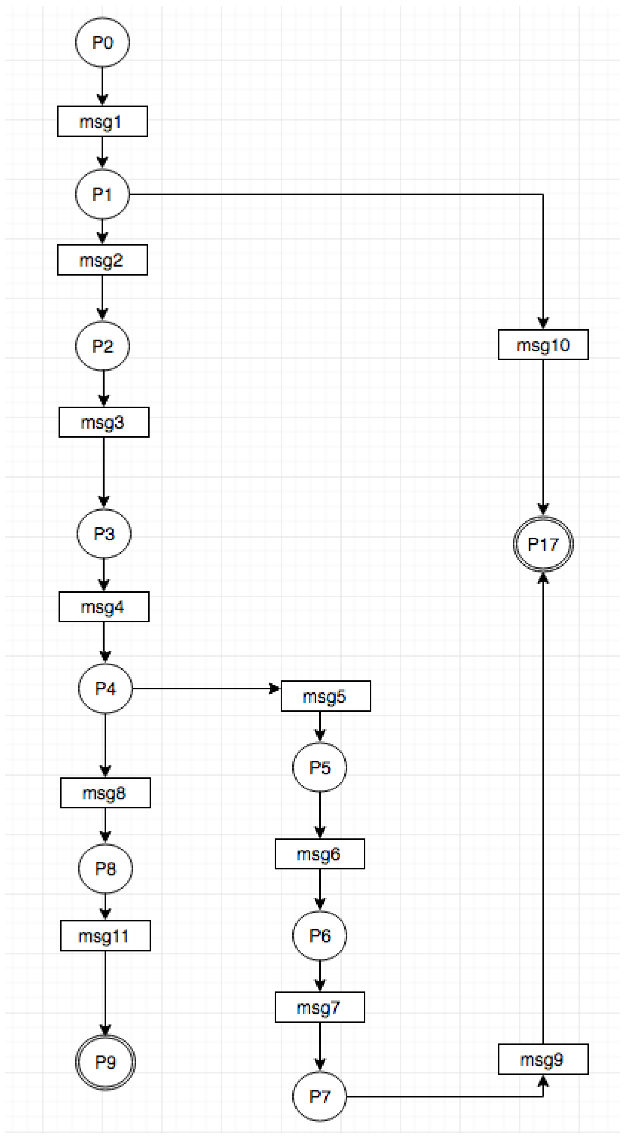
Figure B.8. Flow specification of a cache write back operation initiated from Cache1. This flow is symmetric for CPU2.

All read operations are implemented in protocol presented in Figure B.10. Specification in Figure B.7 will end in $state17$. The rest of the specification without activating cache coherence protocol end in $state9$.



$msg1$: (	CPU1,	wt,	Cache1)	$msg7$: (	Bus,	wt,	Cache1)
$msg2$: (	Cache1,	wt ,	CPU1)	$msg8$: (	Bus,	wt,	Cache1)
$msg3$: (	Bus,	snp,	Cache2)	$msg9$: (	Cache1,	wt,	CPU1)
$msg4$: (	Cache2,	snp,	Bus)	$msg10$: (	Cache1,	wt,	CPU1)
$msg5$: (	Bus,	wt,	Memory)	$msg11$: (	Cache1,	wt,	CPU1)
$msg6$: (	Memory,	wt,	Bus)				

Figure B.9. Flow specification of a cache coherent write operation initiated from CPU1 to Cache. This flow is symmetric for CPU2.



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

Figure B.10. Flow specification of a cache coherent read operation initiated from CPU1 to Cache. This flow is symmetric for CPU2.

LIST OF REFERENCES

[1] Nicola Nicolici and Ho Fai Ko. Design-for-debug for post-silicon validation: Can high-level descriptions help? In *High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International*, pages 172–175. IEEE, 2009.

[2] S. Krstic, Jin Yang, D.W. Palmer, R.B. Osborne, and E. Talmor. Security of soc firmware load protocols. In *Hardware-Oriented Security and Trust (HOST), 2014 IEEE International Symposium on*, pages 70–75, May 2014.

[3] P. Patra. On the cusp of a validation wall. *IEEE Design Test of Computers*, 24(2):193–196, March 2007.

[4] Harry D Foster. Why the design productivity gap never happened. In *Proceedings of the International Conference on Computer-Aided Design*, pages 581–584. IEEE Press, 2013.

[5] Xiao Liu and Qiang Xu. *Trace-Based Post-Silicon Validation for VLSI Circuits.* Springer, 2014.

[6] Miron Abramovici, Paul Bradley, Kumar Dwarakanath, Peter Levin, Gerard Memmi, and Dave Miller. A reconfigurable design-for-debug infrastructure for socs. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, pages 7–12, New York, NY, USA, 2006. ACM.

[7] B. Vermeulen and S. K. Goel. Design for debug: catching design errors in digital chips. *IEEE Design Test of Computers*, 19(3):35–43, May 2002.

[8] Rick Leatherman, Bruce Ableidinger, and Neal Stollon. Processor and system bus on-chip instrumentation. In *Proc. Embedded Systems Conference*. Citeseer, 2003.

[9] Miron Abramovici, Paul Bradley, Kumar Dwarakanath, Peter Levin, Gerard Memmi, and Dave Miller. A reconfigurable design-for-debug infrastructure for socs. In *Proceedings of the 43rd annual Design Automation Conference*, pages 7–12. ACM, 2006.

[10] Ehab Anis and Nicola Nicolici. Interactive presentation: Low cost debug architecture using lossy compression for silicon debug. In *Proceedings of the conference on Design, automation and test in Europe*, pages 225–230. EDA Consortium, 2007.

[11] Kees Goossens, Bart Vermeulen, Remco van Steeden, and Martijn Bennebroek. Transaction-based communication-centric debug. In *Proceedings of the First International Symposium on Networks-on-Chip*, NOCS '07, pages 95–106, Washington, DC, USA, 2007. IEEE Computer Society.

[12] Bart Vermeulen and Kees Goossens. A network-on-chip monitoring infrastructure for communication-centric debug of embedded multi-processor socs. In *VLSI Design, Automation and Test, 2009. VLSI-DAT '09. International Symposium on*, VLSI-DAT '09, pages 183–186, 2009.

[13] Kees Goossens, Bart Vermeulen, and Ashkan Beyranvand Nejad. A high-level debug environment for communication-centric debug. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 202–207, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[14] Amir Masoud Gharehbaghi and Masahiro Fujita. Transaction-based post-silicon debug of many-core system-on-chips. In *ISQED*, pages 702–708, 2012.

[15] Mehdi Dehbashi and Grschwin Fey. Transaction-based online debug for noc-based multiprocessor socs. In *Proceedings of the 2014 22Nd Euromicro International Conference*

on Parallel, Distributed, and Network-Based Processing, PDP '14, pages 400–404, Washington, DC, USA, 2014. IEEE Computer Society.

[16] Amir Masoud Gharehbaghi and Masahiro Fujita. Transaction-based debugging of system-on-chips with patterns. In *Proceedings of the 2009 IEEE International Conference on Computer Design*, ICCD'09, pages 186–192, Piscataway, NJ, USA, 2009. IEEE Press.

[17] Marc Boule, Jean-Samuel Chenard, and Zeljko Zilic. Assertion checkers in verification, silicon debug and in-field diagnosis. In *Proceedings of the 8th International Symposium on Quality Electronic Design*, ISQED '07, pages 613–620, Washington, DC, USA, 2007. IEEE Computer Society.

[18] Eli Singerman, Yael Abarbanel, and Sean Baartmans. Transaction based pre-to-post silicon validation. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 564–568, New York, NY, USA, 2011. ACM.

[19] Yael Abarbanel, Eli Singerman, and Moshe Y. Vardi. Validation of soc firmware-hardware flows: Challenges and solution directions. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, DAC '14, pages 2:1–2:4, New York, NY, USA, 2014. ACM.

[20] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[21] Binsan Khadka. Transformation of live sequence charts to colored petri nets (lsctocpn). Master's thesis, University of Massachusetts Dartmouth, January 2007.

[22] Stephen A White. Process modeling notations and workflow patterns. *Workflow handbook*, 2004:265–294, 2004.

[23] Ho Fai Ko and N. Nicolici. Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(2):285 –297, feb. 2009.

[24] K. Basu and P. Mishra. Efficient trace signal selection for post silicon validation and debug. In *VLSI Design (VLSI Design), 2011 24th International Conference on*, pages 352 –357, jan. 2011.

[25] Sai Ma, Debjit Pal, Rui Jiang, Sandip Ray, and Shobha Vasudevan. Can't see the forest for the trees: State restoration's limitations in post-silicon trace signal selection. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, pages 1–8, Piscataway, NJ, USA, 2015. IEEE Press.

[26] Matthew Amrein. System-level trace signal selection for post-silicon debug using linear programming. Master's thesis, University of Illinois, May 2015.

[27] The gem5 simulator: A modular platform for computer-system architecture research. http://www.gem5.org/docs/html/gem5MemorySystem.html.