Protocol Guided Analysis of Post silicon Traces 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.
Dmitry Goldgof, Ph.D.
Yao Liu, Ph.D.

Date of Approval:
To be determine

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

ACKNOWLEDGMENTS

I am grateful to Dr. Hao Zheng for his precious and constant help on this project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

We consider 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 paper, we provide an algorithm to automatically infer system-level transactions from incomplete, ambiguous, and noisy trace data. We demonstrate the approach on a multicore virtual platform developed within the GEM5 environment.

CHAPTER 1

INTRODUCTION

In this thesis, we will present background information on the area of post-silicon validation and problems in the area in Chapter 2. Then in Chapter 3 an overview of current flow verification is presented. After that, we present our method in Chapter 4, 5 and 6 and implementation in Chapter 6. Chapter 7 will summarize our work and talk about our future works. All the flow specifications and protocols used in implementation process will be explained in Chapter 8.

1.1 Silicon Validation

Validation is the activity of ensuring a product satisfies its specifications, compatible with related software and hardware and meets user expectations. [2]

Silicon Validation is needed because numbers of processor bug are growing and bugs are becoming more diverse and complex.

1.2 Pre- and Post-silicon Validation

The product development cycle often tends to be linear. The phase will start with planning and architecture, followed by RTL and schematic creation and architectural and functional validation (pre-silicon validation) leading up the tape out. Post-silicon validation will start when the first chip arrives. And eventually when all specification are meet, the product will be released to be market. [2]

Pre-silicon validation aims to verify the architecture design before it's implemented on an actual chip. It's mainly done at RTL level on simulators, FPGAs, or emulators. During pre-silicon validation, the cause of fixing bug is relatively low. Any bugs can be fixed by RTL modification and small amount of time The downside is, pre-silicon validation is limited by its speed and coverage. Simulators are usually very slow and only suitable for small portions of the design. FPGAs are up to 3 orders of magnitude faster. Emulator can combine multiple FPGAs to work on a larger portion of the RTL design with cause of limited speed. All the limitations makes pre-silicon only able to verify part of the system.

Post-silicon validation makes use of pre-production silicon integrated circuit (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. However, limited pin-out and other architecture factors makes it impossible to have full observability of the system. Only a limited number of signals can be observed and traced. This limitation brings challenge in both detecting bugs and debugging process.

1.3 Related Work

Paper [2] goes through the definition of silicon validation and reason why it is important. Together with current techniques used for pre- and post-silicon validation.

Our work is closely related to communication-centric and transaction based debug. An early pioneering work is described in [3], 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 [4, 5]. 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 [6]. 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 [7] to address these issues. This approach utilizes a transaction debug pattern specification language [8] 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 [9] elevated to the transaction level.

In [10], 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 [11], 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 50% of the overall design validation cost [12]. 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 [3].

In this thesis, we consider the problem of reconstructing protocol-level behavior from silicon traces in SoC designs. 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. We demonstrate our approach on a multicore virtual platform constructed within the GEM5 environment [13] and another RTL model written in VHDL.

CHAPTER 2

BACKGROUND

2.1 SoC Protocols and Representation of Protocols

An SoC design involves integration of a number of IPs that communicate through complex protocols. Such system-level protocols are typically specified in architecture documents as message flow diagrams. For this paper, we use the words "protocol" and "flow" interchangeably.

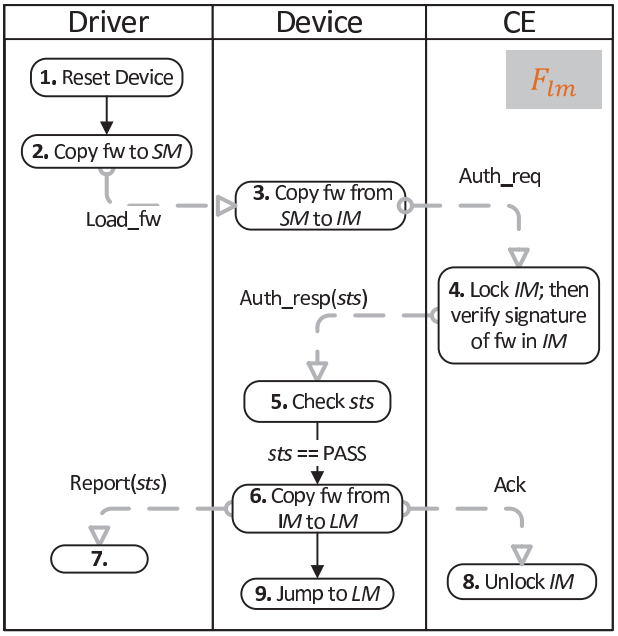


Figure 2.1. A graphical representation of a SoC firmware load protocol [1].

Fig 2.1 shows one diagram for a protocol to authenticate and load a firmware during system boot for firmware upgrade.

In software engineering field, there are two approaches for representing the system protocols: informal and formal representation. Informal representation usually use common graphical notation for representation for better understandability and easier communication with the client. The formal representation 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. Informal format for communications and formal format for verification purpose. Usually it requires manually translation of informal to formal format, which requires massive time and effort when system become large and complex. [**?**] introduced a tool translating informal formal Live Sequence Chart into formal format of Colored Petri-net, which can be great help for future research.

Currently our research is only for verification purpose and our system is too simple to have two format of specification, we only use the formal format of Labeled Petri-net, which will be introduced in Section 3. For the future research, when we need to use large existing system, we will research more about tools that can translate graphical specification to formal specification.

Next two section will talk about sequence diagram as informal representation and Petri-Net as formal representation.

2.2 Sequence Diagram

Sequence Diagram is one of the most used language to provide graphical notation of system's behavior. It represents the life cycle of an processor and the interactions between them. Life cycle of an object is represented by a vertical line and the interaction between objects is represented by a horizontal line with an arrow pointing towards the receiver object. Common used sequence diagram include UML sequence diagram, message sequence diagram and live sequence diagram. [14]

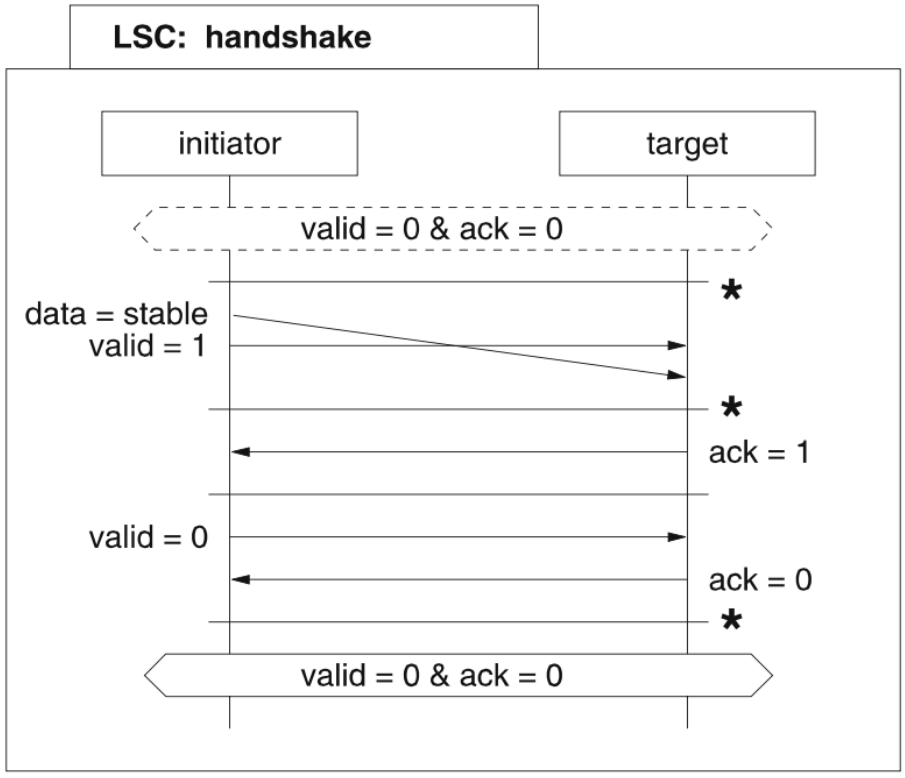


Figure 2.2. Handshake protocol in the graphical LSC representaion

Fig. 2.2 shows an simple exam of Live Sequence Chart (LSC) specification of a handshake protocol between the two processes *initiator* and *target*. A precondition bar requires that the interface be in a state in which *valid* and *ack* be deasserted before any actions of the protocol begin. Once *valid* and *ack* both deassert, however, the interface may begin a new transaction by waiting zero or more clock edges as noted by the hot, starred, clock tick.

At this time, the initiator sets the *data* line and then asserts the *valid* signal. Once the target samples the *valid* signal asserted, it is free to sample the value on the *data* line. After zero or more clock ticks, the target then asserts *ack* to acknowledge receipt of the data value. At this point, a single clock tick must occur before the *initiator* can deassert *valid* and the *target* can deassert *ack* since the data transfer officially occurs on the clock edge on which both *valid* and *ack* are asserted. Again, zero or more clock ticks may occur before

the transaction exits by leaving the interface in a state in which *valid* and *ack* are both deasserted. [15]

2.3 Labeled Petri-Nets

Labeled Petri-nets (LPN) is a formalization of state transition systems that is capable of describing sequencing, concurrency, and choices. Compared with informal representation, it can be verified using mathematical techniques and tools.

Specifically, a labeled Petri-net (LPN) is a tuple (P, T, M_0, V, E, L) where

- P is a finite set of places,
- T is a finite set of transitions,
- M_0 is the set of initially marked places, also referred to as the initial marking.
- V is a finite set of auxiliary variables of integer type,
- E is a finite set of events,
- L is a labeling function that maps each transition $t \in T$ to a triple (g, e, A) where
  - g is a predicate over V,
  - $e \in E$ is an event,
  - A is a set of assignments to some variables in V.

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 a state of a LPN is (M, α) where M is a marking and α is an assignment to all auxiliary variables in V.

The operational semantics of a LPN is defined by transition executions. A transition can be executed after it is *enabled*. A transition $t = (g, e, A)$ is enabled in a state (M, α) if every place in its preset is included in the marking, i.e. $\bullet t \subseteq M$, and the values of the auxiliary

variables make the predicate g be true, i.e. $\alpha \models g$. Execution of t results in a new state (M', α') such that

$$M' = (M - {\bullet t}) \cup t{\bullet},$$

and α' is a new assignment where the values of the auxiliary variables are updated with respect to the assignments A. Furthermore, when t is executed, the event e is emitted.

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*, *init* is the set of initially marked places, also referred to as the *initial marking*, E is a finite set of *events*, and $L : T \to E$ 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 $s \subseteq P$ of a LPN is a subset of places marked with tokens, and it is also referred to as a state of a LPN. The initial marking *init* is also the initial state of the LPN.

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 the marking, i.e. ${\bullet t} \subseteq s$. Execution of t results in a new state s' such that

$$s' = (s - {\bullet t}) \cup t{\bullet},$$

and the emission of event e labeled for t.

The communication protocol in the BMPN shown in Fig. 2.1 is represented by the LPN shown in Figure 2.4. 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 $\langle \texttt{src}, \texttt{dest}, \texttt{cmd} \rangle$ where `cmd` is a command sent from a source component `src` to a destination component `dest`. The solid black places without outgoing edges are *terminals*, which indicate termination of protocols represented by the LPNs. The initial marking is $init = \{p_1\}$. In this LPN model, only the communication

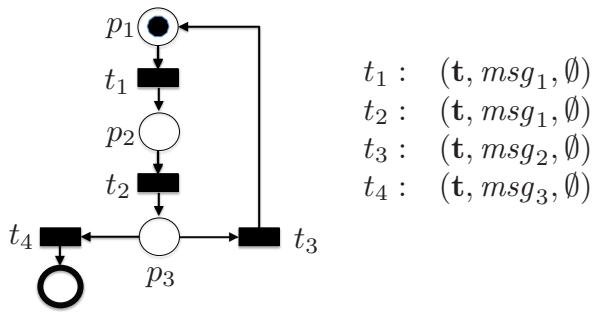


Figure 2.3. An example of a LPN represented system flow

portion of the BPMN specification is represented while the computation portion is ignored. Another example is presented in Fig. 2.3.

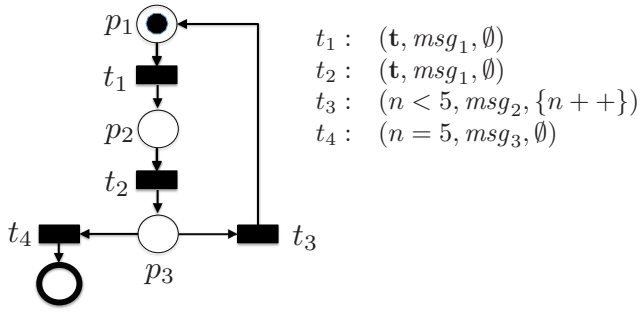


Figure 2.4. Another example of a LPN representing the system flow in Fig. 2.3 with a fixed number of message exchanges.

The above protocol specification can be made more precise by describing how component 2 would respond without using non-determinism. For example, the system architect may wish to specify that component 2 responds with msg_2 to every sequence of two msg_1 for five such sequences in a row. On the sixth such sequence, component 2 responds with msg_3. The LPN modeling such protocol is shown in Figure 2.4. An alternative without using auxiliary variables in the transition predicates is by unrolling the loops for the specified number of times; however this would result in a larger and more complex LPN.

2.4 SoC Protocols and Post-silicon Trace Analysis

In a typical validation setting, the system under debug (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: (1) trace abstraction, and (2) trace interpretation. Trace abstraction maps the hardware 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 will refer to such architectural constructs as *protocol events* or *flow events*. Note that due to limited observability, it may not be possible to map a given set of (observed) hardware signals uniquely to a flow event. Finally, the 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 flow events created during trace abstraction to system-level protocols in order to identify the set of protocol instances (and 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. A *flow execution scenario* is defined as $\{(F_{i,j}, s_{i,j})\}$ where in each element $(F_{i,j}, s_{i,j})$, $F_{i,j}$ is the jth activated instance of flow $F_i \in \vec{F}$, and $s_{i,j}$ is a state of $F_{i,j}$. 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 trace may identify a problem in the protocols themselves, *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

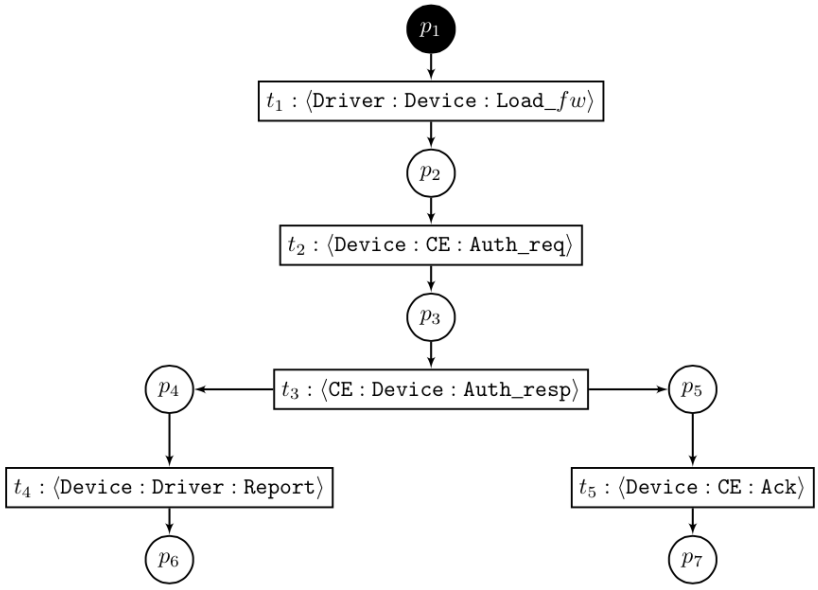


Figure 2.5. LPN formalization. Each event has a form of $\langle \texttt{src}, \texttt{dest}, \texttt{cmd} \rangle$ where `cmd` is a command sent from a source component `src` to a destination component `dest`. The solid black places without outgoing edges are *terminals*, which indicate termination of protocols represented by the LPNs.

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.

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 flows. 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 SUD executions from the observed signal traces.

We formalize the trace interpretation problem in terms of labeled Petri-nets, and discuss our 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 will extend the approach to partial observability.

3.1 Notations and formalization

The set of system flows is a collection $\vec{F}$ of LPNs. 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. Since we assume full observability, we view an *observed trace* $\rho = e_1 e_2 \dots e_n$ as a sequence of events. 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. We call such execution scenarios *compliant* with ρ. Let $accept(F_{i,j}, s_{i,j}, e)$ be a function that determines if event e can be emitted by $F_{i,j}$ in state $s_{i,j}$. Formally, $accept(F_{i,j}, s_{i,j}, e)$ returns $(F_{i,j}, s'_{i,j})$

where $s'_{i,j} = (s_{i,j} - \bullet t) \cup t\bullet$ if there exists a transition t in F_i such that $L(t) = e$ and $\bullet t \subseteq s_{i,j}$. It returns $\emptyset$ otherwise.

3.2 Flow Guided Trace Interpretation

Given an observed trace ρ and the set $\vec{F}$ of LPNs, Algorithm 1 provides 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, we update *Scen* by either extending a member of *scen* or initiating a new protocol instance for each $scen \in Scen$ with respect to e_h in every possible way. If e_h cannot be emitted by any existing or new flow instances, then we report that the trace is `inconsistent`, *i.e.*, there is no possible interleaving of the protocol instances from $\vec{F}$ that is compliant with ρ.

Given a trace of flow events $\rho = e_1 e_2 \dots e_n$, 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 $accept(F_{i,j}, s_{i,j}, e_h) = (F_{i,j}, s'_{i,j})$, 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 $accept(F_{i,j+1}, init_{i,j+1}, e_h) = (F_{i,j+1}, s'_{i,j+1})$, create a new scenario $scen' = scen \cup (F_{i,j+1}, s'_{i,j+1})$, which is added into $Scen'$.

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

If every events in ρ is successfully mapped to some flow instance, this algorithm returns a set of flow execution scenarios such that every flow instance is in its terminal state. On the other hand, inconsistent events can also be encountered. An event e is *inconsistent* if for each flow execution scenario $scen \in Scen$, the following two conditions hold.

```
Create an empty scenario scen
Scen = {scen}
foreach h, 1 ≤ h ≤ n do
    found ← true
    Scen' = ∅
    foreach scen ∈ Scen do
        foreach (F_{i,j}, s_{i,j}) ∈ scen_1 do
            if accept(F_{i,j}, s_{i,j}, e_h) = (F_{i,j}, s'_{i,j}) then
                Let scen' be a copy of scen
                scen' ← (scen' − (F_{i,j}, s_{i,j})) ∪ (F_{i,j}, s'_{i,j})
                Scen' ← scen' ∪ Scen'
                found ← false
            end
        end
        foreach F_i ∈ F⃗ do
            create a new instance F_{i,j+1}
            if accept(F_{i,j+1}, init_{i,j+1}, e_h) = (F_{i,j+1}, s'_{i,j+1}) then
                Let scen' be a copy of scen
                scen' ← scen' ∪ (F_{i,j+1}, s'_{i,j+1})
                Scen' ← scen' ∪ Scen'
                found ← false
            end
        end
    end
    if found == true then
        return Inconsistent
    end
    Scen = Scen'
end
return Scen
```

Algorithm 1: CHECK-COMPLIANCE($\vec{F}$, ρ)

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

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

An inconsistent event is the one produced by SUD execution but cannot be mapped to any flow instances no matter how the trace prior to event e is interpreted. Inconsistent events indicates possible causes of system failures.

Based on the above discussion, the trace interpretation algorithm returns two pieces of information: 1) a set G of flow execution scenarios where every flow instance in every scenario is in its terminal state, 2) a set B of pairs, each of which includes a set of flow execution

scenarios and an inconsistent event. The set B provides valuable information for debuggers to root cause system failures. One of these two sets can be empty. With the full observability, the set G includes a single flow execution scenario derived for a trace. In reality, it is always the case that the SUD is only partially observable. Therefore, due to the lack of information for precise interpretation, a set of flow execution scenarios is typically derived for a given trace as the result of the trace analysis.

To illustrate the basic idea of the trace analysis method, consider the system flow shown in Figure 2.3. Let F_1 denote such flow. Suppose that a hardware system implements flow F_1, and the following trace of flow events is abstracted from an observed signal trace as a result of executing such system.

$$msg_1\ msg_1\ msg_1\ msg_2\ msg_3\ \ldots$$

This trace is interpreted from the first event to the last in order to derive all possible flow execution scenarios. At the beginning, the first event msg_1 is processed. 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 $accept(F_{1,1}, init_1, msg_1) = (F_{1,1}, \{p_2\})$ where $init = \{p_1\}$ is the initial state of F_1. As the result, the flow execution scenario after interpreting the first event msg_1 is $\{(F_{1,1}, \{p_2\})\}$.

Next, the second msg_1 is interpreted on both scenarios This event could be the result of two possible cases. In the first case, this event is the result of the continuing execution of $F_{1,1}$ as $accept(F_{1,1}, \{p_2\}, msg_1) = (F_{1,1}, \{p_3\})$. In the second case, the system may activate another instance of F_1, $F_{1,2}$ such that the second event msg_1 is a result of executing this new instance. Therefore, the interpretation of the first two events msg_1 leads to two flow execution scenarios as shown below.

$$\begin{aligned} &1.\quad \{(F_{1,1}, \{p_3\})\}, \\ &2.\quad \{(F_{1,1}, \{p_2\}), (F_{1,2}, \{p_2\})\} \end{aligned} \tag{1}$$

Now, consider the third msg_1 for each of the two scenario derived in (1). For the execution scenario 1, $F_{1,1}$ is not able to accept msg_1 as it is in state $\{p_3\}$. On the other hand, this event could be the result of activation of a new flow instance. Therefore, this execution scenario can be revised accordingly as

$$\{(F_{1,1}, \{p_3\}), (F_{1,2}, \{p_2\})\}. \tag{2}$$

For the execution scenario 2, event msg_1 could be the result of continuing execution of $F_{1,1}$ or $F_{1,2}$, or it could be a result of activation of a new flow instance. Therefore, three new execution scenarios can be derived as shown below for this event.

$$\begin{aligned} &\{(F_{1,1}, \{p_3\}), (F_{1,2}, \{p_2\})\}, \\ &\{(F_{1,1}, \{p_2\}), (F_{1,2}, \{p_3\})\}, \\ &\{(F_{1,1}, \{p_2\}), (F_{1,2}, \{p_2\}), (F_{1,3}, \{p_2\})\} \end{aligned} \tag{3}$$

Since the flow execution scenario in (2) already exists in (3), the three flow execution scenarios shown in (3) is the result from the interpretation of the first three events msg_1.

The next flow event in the trace msg_2 is analyzed for the flow execution scenarios as shown in (3). For the first scenario $\{(F_{1,1}, \{p_3\}), (F_{1,2}, \{p_2\})\}$, event msg_2 can only be the result from executing $F_{1,1}$ as $accept(F_{1,1}, \{p_3\}, msg_2) = (F_{1,1}, \{p_1\})$. Based on the same reasoning, this event can only be the result from executing $F_{1,2}$ in the second scenario, and it moves $F_{1,2}$ to a new state $\{p_3\}$ too. The interesting case is the third scenario where none of the flow instances can allow msg_2 to happen. This is due to the fact that the flow must be in $\{p_3\}$ for msg_2 to happen. This indicates the third system execution scenario is impossible for the prefix of the flow event trace upto msg_2, therefore this scenario is ignored from further analysis. After analyzing event msg_2, the updated system execution scenarios are shown below.

$$\{(F_{1,1}, \{p_1\}), (F_{1,2}, \{p_2\})\}, \ \{(F_{1,1}, \{p2\}), (F_{1,2}, \{p_1\})\}.$$

The last flow event in the trace is msg_3. Considering the above two possible system executions, neither can allow this event to be produced as none of the flow instances in both system executions is in state $\{p_3\}$. What this means is that the system does *not* implement the flow specification correctly as it produces something not allowed by the specification. By examining the system executions right before the "buggy" event, debuggers may gain more information on when and where the problem might be. The trace interpretation algorithm adds these two scenarios along with the fifth event msg_3 into B, and returns it for debuggers to analyze.

Next, we will use another example to illustrate when all flow events are successful mapped and reach the correct final state. We consider the system flow in Fig. 2.4 which we will call F_1. Suppose that the following flow trace is abstracted from an observed signal trace.

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

Here transition names in the LPN are used to represent the flow events in the trace. The first four events results in the following flow execution scenario

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

For the first event t_3, it results in two execution scenarios below depending on which flow instance emits t_3.

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

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

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

Using Algorithm 1 to handle the remaining four events, the following execution scenario is derived.

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

CHAPTER 4

TRACE ANALYSIS UNDER PARTIAL OBSERVABILITY

In hardware that implements a given system flow specification, a flow event is defined as an event or a sequence of events on a set of signals. Due to the limited number of pins on the boundary of chips available for observation, only a small fraction of system signals can be observed during debug. In this section, we discuss how the trace analysis method presented above can be adapted to deal with signal traces of partial observability.

In general, a signal trace of partial observability corresponds a set of traces of flow events due to the small number of observable signal or ambiguous interpretation of signal events. In the following, we discuss two cases for trace abstraction on partial observability: mapping a single signal event to a flow event or mapping a sequence of signal events to a flow event. A signal event is defined as a state on or an assignment to a set of signals. Hereafter, the term *flow traces* is used to refer to traces of flow events.

4.1 Single signal event

Consider the following example for the first case. 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.

$$e_1 : \quad abc$$

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

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

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

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

During trace abstraction, the first two signal events bc can be mapped to $\{e_1, e_2\}$ since a is not observable, and the last one $b'c$ is mapped to $\{e_3\}$. Therefore, this signal trace is abstracted to four flow traces, $\{e_1, e_2\} \times \{e_1, e_2\} \times \{e_3\}$.

4.2 Sequence of signal events

Next, we consider the case where a flow event is mapped from a sequence of signal events. Now suppose that two other flow events are implemented by sequences of signal events as defined in the list below.

$$e_4 : \quad abc\ \bar{a}bc$$
$$e_5 : \quad abc\ abc\ abc\ \bar{a}bc$$

Given an observed trace of the same observability shown below

$$bc\ bc\ bc\ bc,$$

it is abstracted to the following flow traces.

$$e_4e_4,\ \textvisiblespace e_4\textvisiblespace,\ e_4\textvisiblespace\textvisiblespace,\ \textvisiblespace\textvisiblespace e_4,\ e_5$$

where $\textvisiblespace$ denotes signal events that are not mapped to any flow events. Note that the above abstraction leads to three distinct flow traces as the middle three correspond to the same flow trace.

4.3 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 [16, 17], and finding an automated way of signal selection algorithm can be one of our future research direction. We will briefly describe 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.

4.4 Trace Signal Selection

Our thesis did signal selection manually by the system designer, we tried different sets of signals to observe and chose those sets that leads to fewest numbers of final scenario. However, as our system become more complex, a more automated tool is needed for better and faster signal selection process.

Most of the current signal selection algorithm are done in low-level and using SSR(State Restoration Ratio) as their metric. SRR is used by measures the number of design states reconstructed from the signals observed. However, agreeing with [18], because SSR treats all signal equally and thus always favor big arrays, it is not very helpful to restore useful states for our case. We need a tool that can restore the maximum states in system level and thus lead to less scenarios.

[18] 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 very useful in our case. [19] 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. Trying this method on our system and evaluate its performance is one of our future jobs.

4.5 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 permit 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 permit 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.

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

IMPLEMENTATIONS AND RESULTS

5.1 Simulation simple TLM SoC with 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 [13]. This SoC model, as shown in Fig. 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. Components communicate with each other by sending and receiving various request and response messages. 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.

By combining the informations from all of the nine communication monitors, required communication traces are obtained. As a virtualized SoC platform, GEM5 has three types of request: timing, atomic and functional. Timing request include the modeling of queuing delay and resource contention. Atomic request is a faster than timing request with no delay. Functional request is used for loading binaries, examining/changing variables in the simulated system, and so on. In our case, we specify all our request to be timed as it's the most detailed one.However, some of the system initiation is still atomic and it's not related to our research, so we only took the messages that are timing request from our collected data.

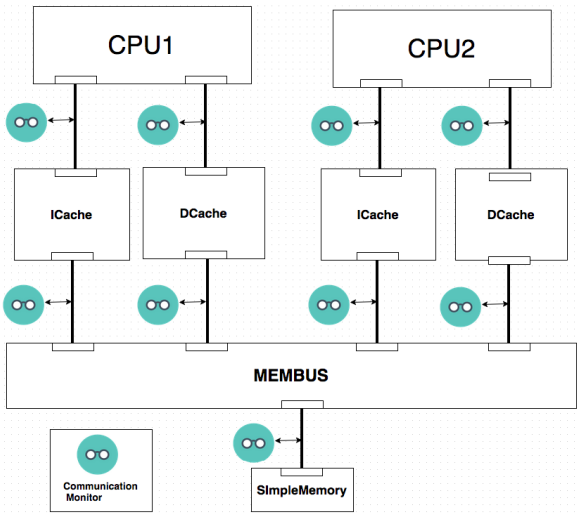


Figure 5.1. SoC platform structure.

For this model, we consider the flow specifications describing the cache coherence protocols supported in GEM5 that is used to build the model in Fig 5.1. The GEM5 cache coherence protocols can be found at [20]. These flow specifications describe data/instruction read operations and data write operations initiated from CPUs. Three such flows describe the cache coherent protocols for each CPU. Since there are two CPUs, there are six flows in the model.

We wrote two simple concurrent programs, one for each CPU, to exercise the flows. They read numbers from a file, perform some operations on these numbers, and store the results back to the file. How GEM5 supports shared memory multi-threaded program execution is unclear. Therefore, no data are shared in both caches in this test. Furthermore, GEM5 does not support true concurrency. When there are two programs running on the CPUs, GEM5 alternates the executions between the two CPUs. 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.

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

	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

We produce a result file including all of the communication messages between every components from two simple programs running simultaneously on each CPU. The program assigned to CPU1 read one file three times for one letter and writing one letter for three times to the same file. CPU2's program will do the same read and write functionalities to the same file, only difference is that CPU2 will write first and then read. One thing we should know about GEM5 is that even when we run these two programs concurrently, it will attempt to produce a concurrent result. The data we collected shows that it's not the real nondeterministic concurrency. What GEM5 did was it allowed two CPUs to execute its own instruction in turn, therefore the order was deterministic. Therefore, no matter how many times we ran the program, it produced the same result. We tried other virtual SoC platform softwares, and this was the best nondeterministic concurrency we can get. Even if this is slightly off with what the real chip should work, it still surve our purpose of testing the correctness of our algorithm.

After this model is executed with the simple concurrent programs, 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

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

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 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 system when executing the program involve the CPUs and their corresponding caches only,

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

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 events passing the same link are not distinguishable due to the limited observability. The monitors are modified such that whenever an event is captured on one of the links, it dumps a set of events passing through the same link into the trace file. Therefore, each line of the trace file corresponds to a set of events. 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 events 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 events. This is due to the fact that a trace of ambiguous events is in fact a set of traces of original events, 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 during the analysis process is 70384, many of which are invalid and removed eventually. However, controlling the number of intermediate execution scenarios 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 events 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 events 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, events in the same group are assumed to be non-distinguishable. The monitors are modified accordingly such that they output all events 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 events.

5.2 Simulation simple RTL SoC with VHDL

Another register transaction level SoC model is constructed. This model is more specific compared to the last model as the register value will be updated every cycle, thus will be

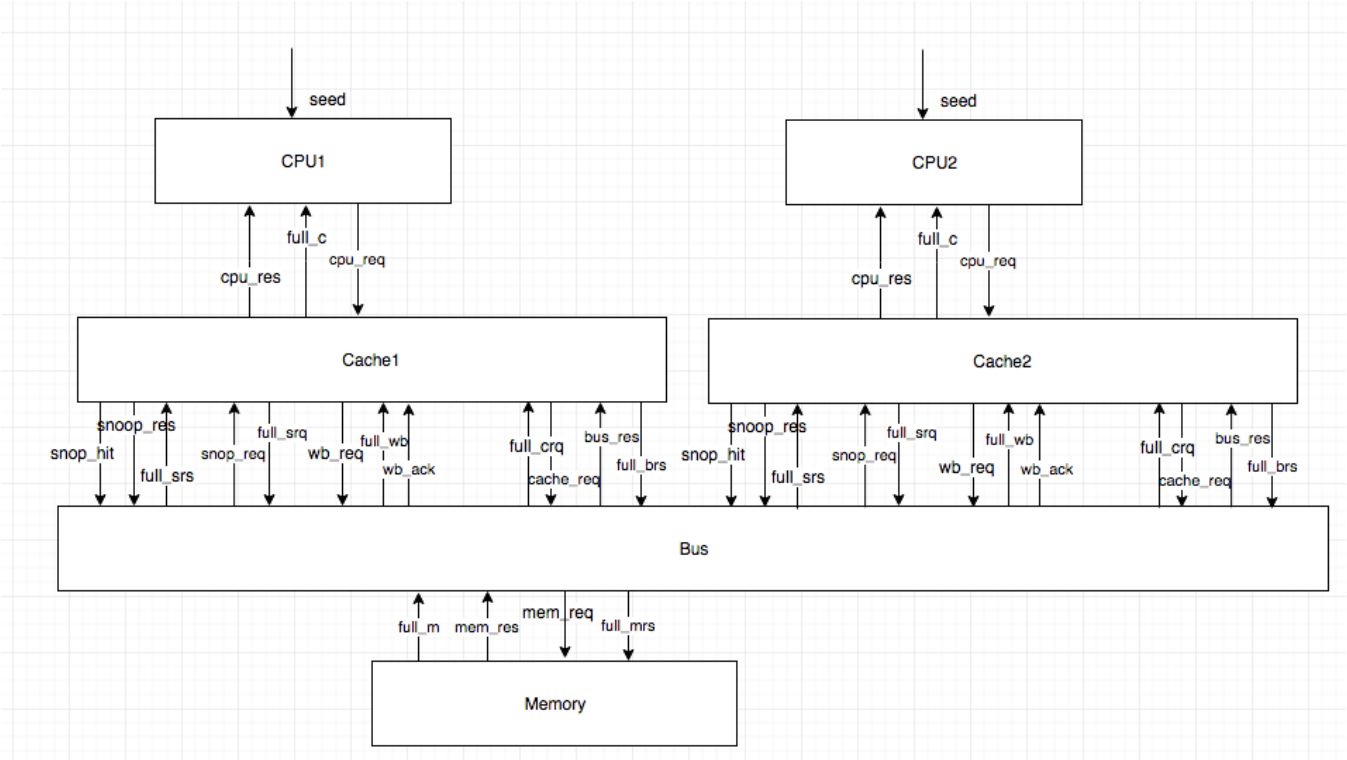


Figure 5.2. SoC platform structure.

much slower than the model on GEM5. Because the lack of similar research, we couldn't find any existing model, so this model is built from scratch just by me. The desired protocols are implemented inside all the components. Because this is a simplified model, the CPU can't run software programs. A test generator is implemented inside CPU to allow simple functions.

As showed in Fig. 5.2, this model consists of two CPU models, each with its own 1KB cache. The caches are connected to a 4MG memory through a memory bus model. At each rising clock cycle, our model will record selected signal value into a vcd(value change dump) format file.

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 is implemented for both CPU. Write back protocol is new and will be invoked when Cache need to flush back dirty datas.

Table 5.4. 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

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 system takes 10 second to run and the peak memory used is 18MB.

Table 5.4 shows the number of flow instances contained in that scenario for the six flows describing cache coherent operations initiated from both CPUs.

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

- Error one: interconnect bus sending same request to Memory more than one time. During the trace analysis, our algorithm shows that the trace will always stop when interconnect send request to Memory and the message cannot be mapped to any existing scenarios. The error message looks exactly the same as the previous message. We traced the bug to interconnect and find out the request wasn't reset correctly.

- Error two: command changed after interconnect bus receive snoop request from Cache. Our trace analysis algorithm find message from cache to memory can't map to any existing scenarios in multiple runs, and all the stacked messages are snoop response. By debugging the cache component, we discovered the wrong implemented cache coherent protocol and fixed the error.

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

APPENDICES

Appendix A Flow specifications and protocols provided by GEM5

A.1 Flow Specifications

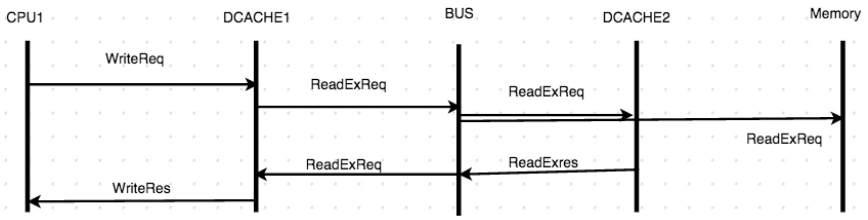


Figure A.1. Flow sequence chart of write operation when requested data is not included in Dcache. ReadExRes can also be sent from Memory if Dcache2 does not have requested data. This sequence chart is symmetric for CPU2.

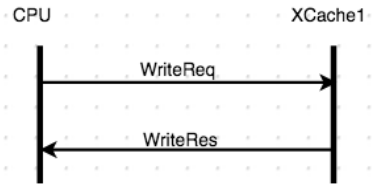


Figure A.2. Flow sequence chart of write operation when XCache has the exclusive right of requested data. XCache can be instruction cache or data cache. This sequence chart is symmetric for CPU2.

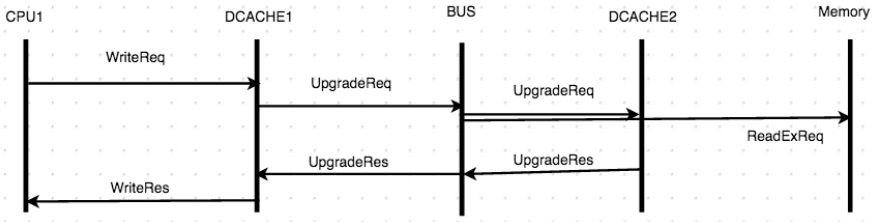


Figure A.3. Flow sequence chart of write operation when requested data is shared by another component. UpgradeRes can also be sent from Memory if Dcache2 does not have requested data. This sequence chart is symmetric for CPU2.

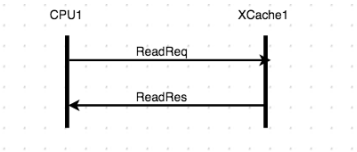


Figure A.4. Flow sequence chart of read operation when XCache has the exclusive right of requested data. XCache can be instruction cache or data cache. This sequence chart is symmetric for CPU2.

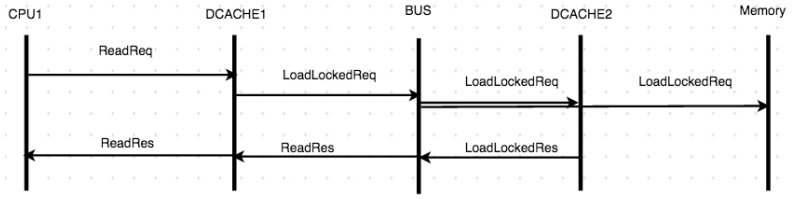


Figure A.5. Flow sequence chart of read operation when requested data is shared by another component. LoadLockedRes can also be sent from Memory if Dcache2 does not have requested data. This sequence chart is symmetric for CPU2.

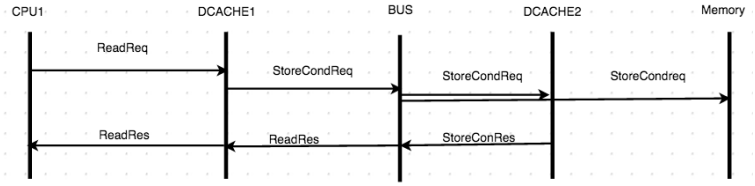
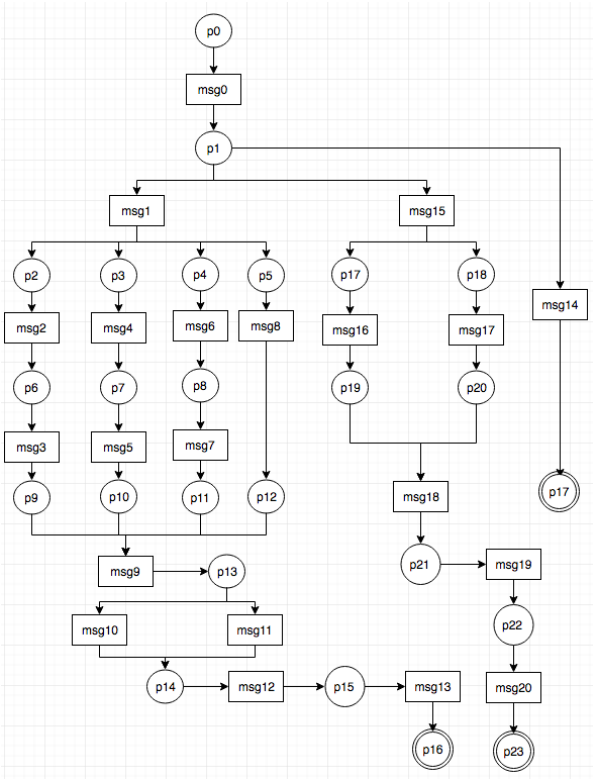


Figure A.6. Flow sequence chart of read operation when requested data is not present. StoreCondRes can also be sent from Memory if Dcache2 does not have requested data. This sequence chart is symmetric for CPU2.

A.2 Protocols

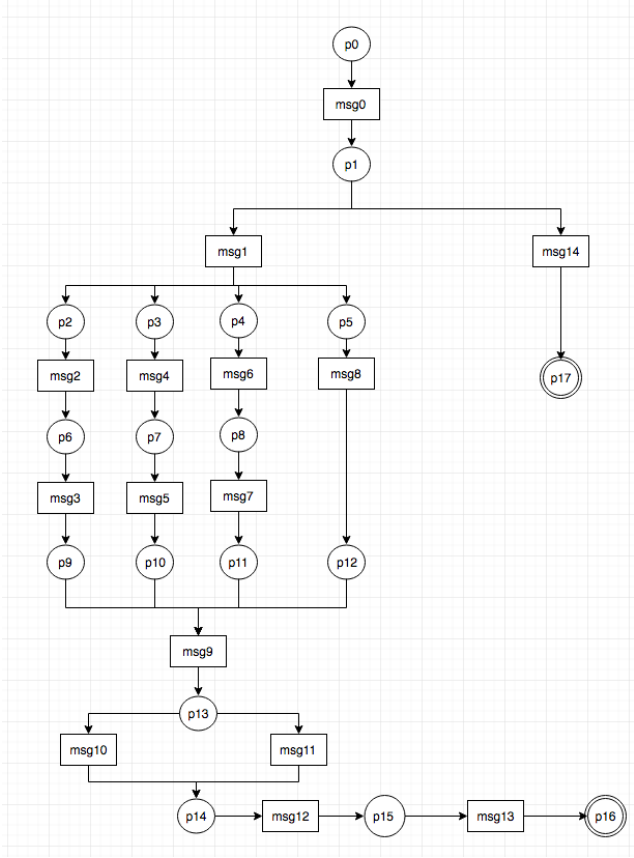
Appendix A (Continued)



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

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

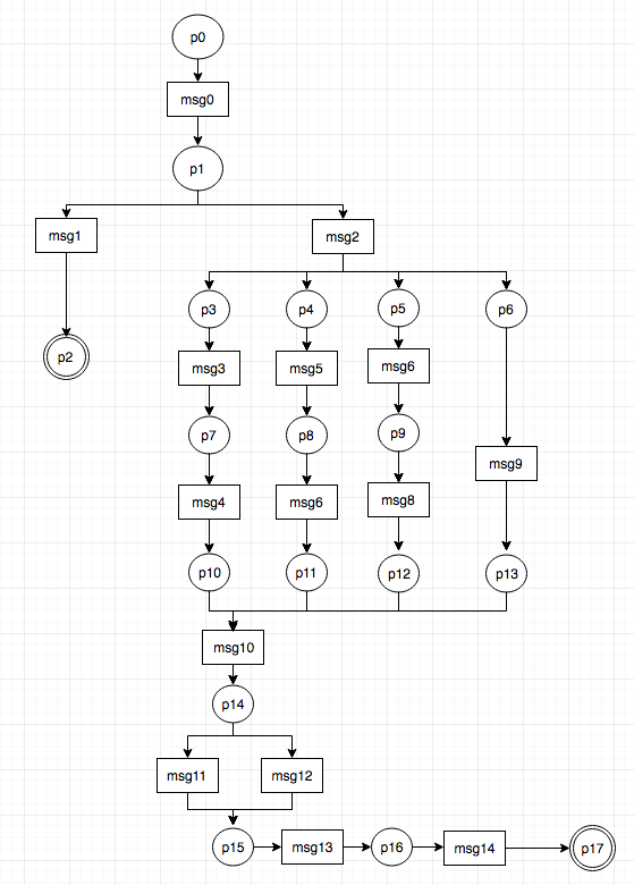
Appendix A (Continued)



$msg0$: (	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.

Appendix A (Continued)



$msg0$: (	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 RTL model in VHDL

B.1 Flow Specification

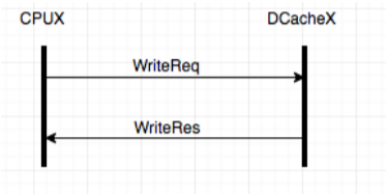


Figure B.1. CPU write when cache has exclusive right of the requested data.

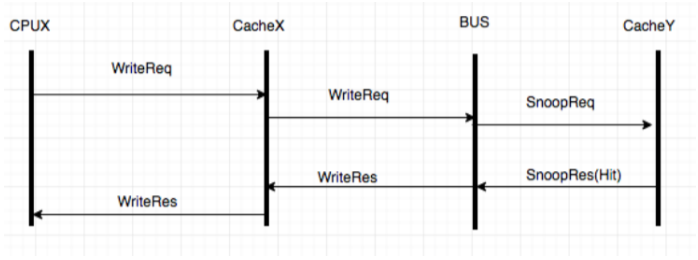


Figure B.2. CPU write when data only exist in the other CPU's cache

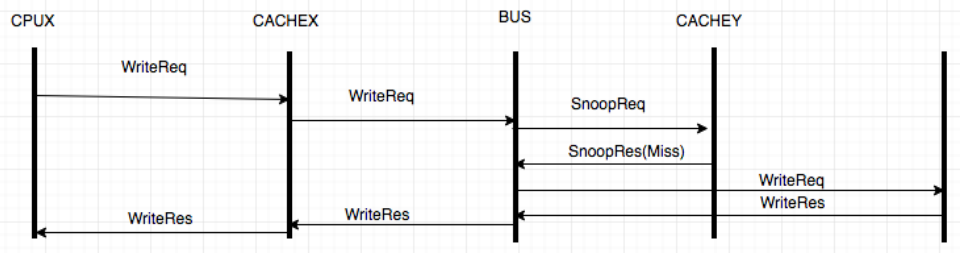


Figure B.3. CPU write when requested data only reside in Memory

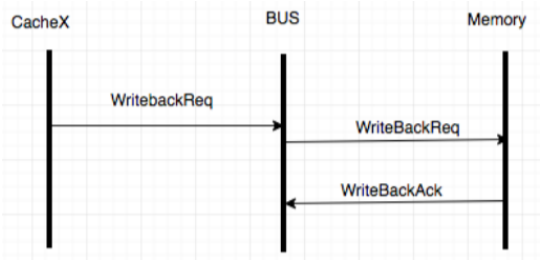


Figure B.4. Cache send write back request to Memory

Appendix B (Continued)

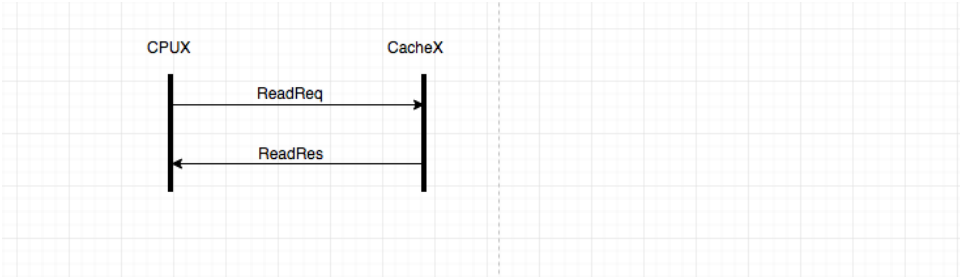


Figure B.5. CPU read when cache has exclusive right of the requested data.

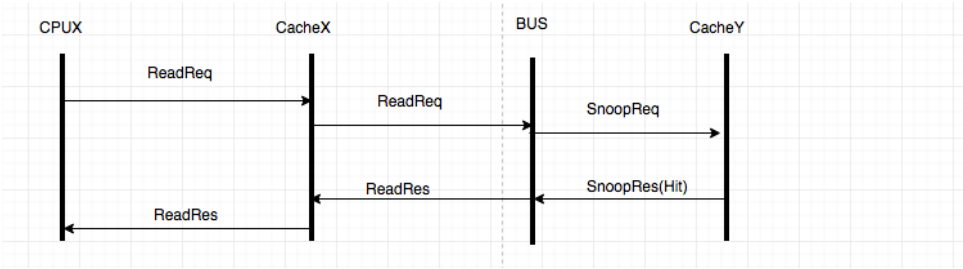


Figure B.6. CPU read when data only exist in the other CPU's cache

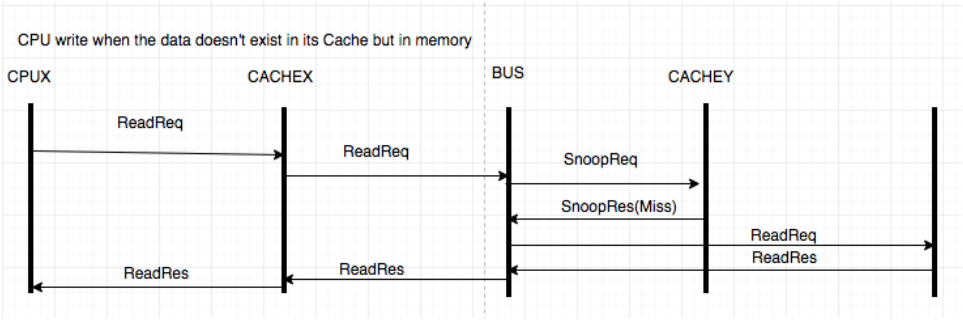
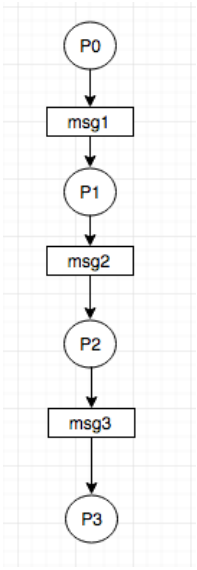


Figure B.7. CPU read when requested data only reside in Memory

B.2 Protocol

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



$msg1$: (Cache1, wt , Bus)

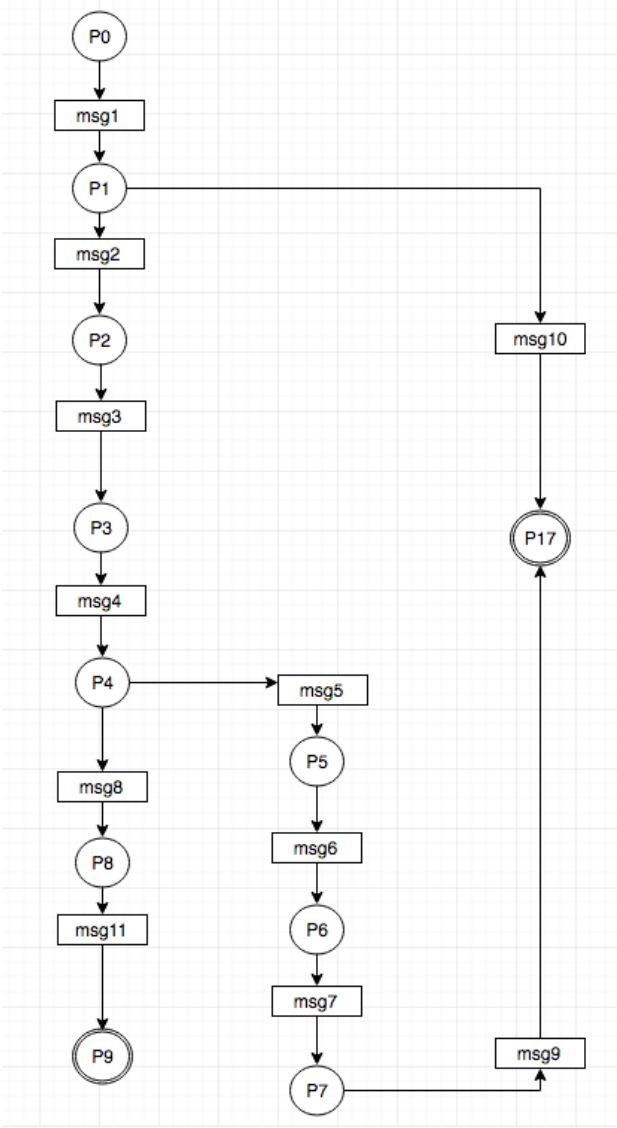
$msg2$: (Bus, wt, Memory)

$msg3$: (Memory, wt, Bus)

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

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

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



$msg1$: (CPU1, wt, Cache1)

$msg2$: (Cache1, wt , CPU1)

$msg3$: (Bus, snp, Cache2)

$msg4$: (Cache2, snp, Bus)

$msg5$: (Bus, wt, Memory)

$msg6$: (Memory, wt, Bus)

$msg7$: (Bus, wt, Cache1)

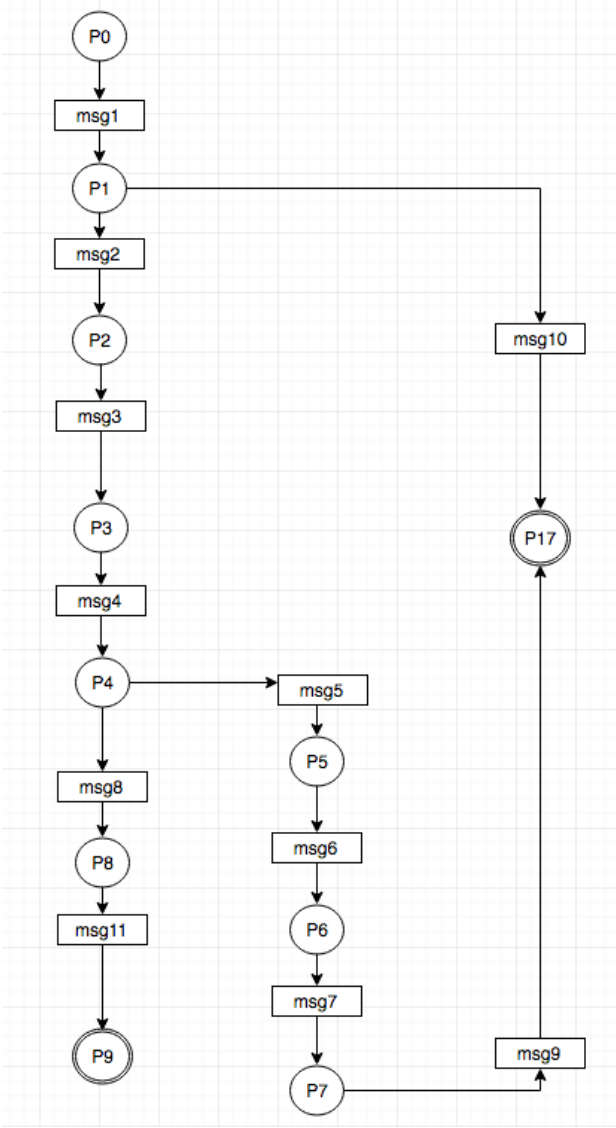
$msg8$: (Bus, wt, Cache1)

$msg9$: (Cache1, wt, CPU1)

$msg10$: (Cache1, wt, CPU1)

$msg11$: (Cache1, wt, CPU1)

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)

$msg2$: (Cache1, rd , Bus)

$msg3$: (Bus, snp, Cache2)

$msg4$: (Cache2, snp, Bus)

$msg5$: (Bus, rd, Memory)

$msg6$: (Memory, rd, Bus)

$msg7$: (Bus, rd, Cache1)

$msg8$: (Bus, rd, Cache1)

$msg9$: (Cache1, rd, CPU1)

$msg10$: (Cache1, rd, CPU1)

$msg11$: (Cache1, rd, CPU1)

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] 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.

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

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Priyadarsan Patra. On the cusp of a validation wall. *IEEE Des. Test*, 24(2):193–196, March 2007.

[13] 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.

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

[15] Annette Bunker, Ganesh Gopalakrishnan, and Konrad Slind. Live sequence charts applied to hardware requirements specification and verification. *International Journal on Software Tools for Technology Transfer*, 7(4):341–350, 2005.

[16] 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.

[17] 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.

[18] 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.

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

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