Flow-Directed Trace Analysis for Post-Silicon Validation

Hao Zheng, Yuting Cao
Computer Science and Engineering
University of South Florida
Tampa, Florida 33620

Sandip Ray, Jin Yang
Strategic CAD Labs
Intel Corporation
Portland, Oregon

***Abstract*—Validation of multicore systems-on-chip (SoC) is very challenging, and main reasons include the limited internal operation observability and non-determinism due to multiple clock domains. Very often, the observed signal traces offer limited values for debuggers to understand system internal behavior. This paper presents a trace analysis approach for post-silicon validation. This approach takes system flow specifications describing system behavior at a high level and an observed signal trace, and interprets the trace with respect to the system flow specification. The result from the trace interpretation is a set of flow instances such that their execution during a test explains the observed signal trace. This approach transforms partially observed and non-deterministic signal traces to a representation that is more understandable, and offers more helpful information to locate design defects and a measure to evaluate validation coverage.**

I. Introduction

Systems-on-chips (SoCs) have become more complex as more functionalities are integrated on chips. To reduce design time, more external IPs are integrated into current SoC designs. At the high level, a SoC can be viewed as a network of IP blocks communicating by a large number of protocols. A typical SoC design consists of multiple programmable processor cores where a number of software threads can be executed simultaneously. The resulting designs usually operate in multiple clock domains, and show a high degree of concurrency. Due to the sophisticated communication mechanisms employed in current SoC designs to support complex functional behaviors, traditional processor-centric validation has to be enhanced to focus inter-core communications.

To make validation even more challenging, the high degree of integration in current SoC designs causes traditional chip observation points to become inaccessible. A SoC design usually contains millions of signals. In post-silicon validation, thousands of those signals are tapped for observation. Due to the limited availability of the chip pins, only about a hundred of the tapped signals can be traced. This severely limits observability hinders post-silicon debuggers' capability to root cause an observed failure. To make the situation even worse, the observed traces often display non-deterministic behavior due to many concurrent operations occurred in multiple clock domains in a design. The above factors make debugging highly difficult in failure root-causing, and need to be addressed.

In a typical design flow, a design starts with some high-level specification, particularly for communication protocols. The abstract protocol specification usually captures information exchanged among IP blocks and how it is exchanged to support functional behavior. Graphical representations of protocols such as Message Sequence Charts [1] or Live Sequence Charts [2] describe protocols intuitively. However, its success in hardware industry is limited. Recently, message flows in BPMN [3] have been being advocated as a formalism for specifying architecture requirements or system-level use cases such as power management, security, etc. In general, a flow describes a set of message sequences across multiple IP blocks following a system communication protocol. Specification of system flows BPMN is intuitive and easier for human to understand and to reason about system behavior. An example of flow specification in BPMN is shown in Figure 1.

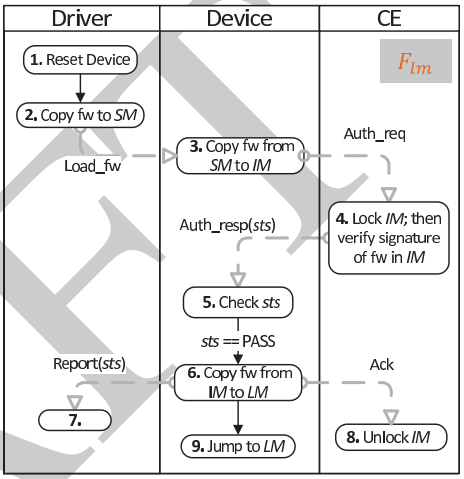


Fig. 1. A graphical representation of a SoC firmware load protocol in BPMN [3].

As indicated above, interpreting and understanding the raw signal traces observed from executing the chip is extremely difficult due to the limited observability and non-determinism. This paper proposes a trace analysis method which analyzes and interprets the observed signal traces at the level of system flow specifications. During the step of analysis and interpretation, this method tries to find out possible executions of system flows such that the observed signal trace is a result of such executions. The resulting scenarios of flow executions often

give debuggers more clear and better structured system-level views of the internal operations of the system under debug. With a better understanding of system operations in a test environment, therefore, it would make easier for debuggers to pinpoint and root cause system failures. In addition to debugging, another important benefit is better coverage evaluation as a scenario of flow executions captures the information about the types and number of instances of flows executed by the chip in a test environment. That information can help to determine the quality and coverage of validation.

II. Labeled Petri-Nets

Although system flows depicted in BPMN as in Fig. 1 are intuitive for system architects to understand communications across multiple components, they are not suitable for algorithmic analysis. In this section, we present an alternative formalism, *labeled Petri Nets*, to represent communication aspects of system flows with the intuitive descriptions of internal operations for IP blocks abstracted away whenever possible. Labeled Petri nets are capable of describing sequencing, concurrency, and choices. This formalism has well defined operational semantics, efficient analysis analysis algorithms and and has been used widely in modeling and analyzing communication protocols, concurrent programs, etc.

Specifically, a labeled Petri-net (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,
- $L : T \rightarrow E$ 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.

A simple communication protocol specification between two components represented by a LPN is shown in Figure 2. 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. The solid black places without outgoing edges are *terminals*, which indicate termination of protocols represented by the LPNs. In this LPN, the initial state $init = \{p_1\}$. In this

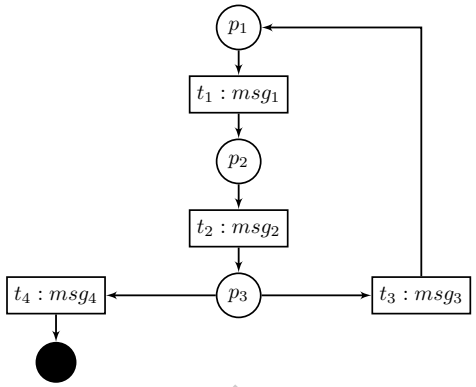


Fig. 2. An example of a LPN representing a system flow.

protocol, component 1 sends two consecutive messages msg_1 to component 2, which responds with either msg_2 or msg_3 non-deterministically. If msg_2 is responded, component 1 repeats the protocol by sending two consecutive msg_1 again. Otherwise, component 1 stops the protocol upon receiving msg_3. This LPN represents two possible communication sequences as shown below.

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

III. Flow-Directed 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. Due to the limited observability and inherent non-determinism in today's SoC designs, the observed signal trace is difficult to understand, thus providing limited values for debugging. In this section, 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.

The trace analysis method consists of two steps: *trace abstraction* and *interpretation*. The trace abstraction takes a signal trace observed during the execution of the SUD, and abstracts it with respect to information captured in the system flows. This requires recognizing signal events and mapping the signal events or sequences of the signal events to flow events appeared in system flow specifications. For example, a data write message may be a single event in a system flow, however, such a flow event maybe implemented in hardware as a sequence of events including a header, a number of data word transfers and a tail. This abstraction requires a mapping relation from flow events to sequences of signal events, and it is reasonable to assume that this relation is available.

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.

An observed trace of flow events $\rho = e_1 e_2 \dots e_n$ is a result of SUD executing the flow instances of some flow execution scenario. The goal of the trace interpretation is to derive such scenario. 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 $s_{i,j}$. This function is used during the trace interpretation for checking whether an event of an observed trace is a result of executing some flow instance. Formally, $accept(F_{i,j}, s_{i,j}, e)$ returns $(F_{i,j}, s'_{i,j})$ if there exists a transition t in F_i such that $L(t) = e$ and $\bullet t \subseteq s_{i,j}$. In this case, $s'_{i,j} = (s_{i,j} - \bullet t) \cup t\bullet$. It returns $\emptyset$ if no such t exists in F_i.

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.

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 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. 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.

$$msg_1\ msg_1\ msg_1\ msg_2\ msg_3\ \dots$$

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.

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

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

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

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.

$$\begin{array}{l} \{(F_{1,1}, \{p_3\}), (F_{1,2}, \{p_2\})\}, \\ \{(F_{1,1}, \{p_2\}), (F_{1,2}, \{p_3\})\}, \\ \{(F_{1,1}, \{p_2\}), (F_{1,2}, \{p_2\}), (F_{1,3}, \{p_2\})\} \end{array} \quad (3)$$

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.

A. Trace Analysis with 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 corresponding a set of traces of flow events due to the 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. 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.

$$\begin{aligned} e_1 &: \ abc \\ e_2 &: \ \bar{a}bc \\ e_3 &: \ a\bar{b}c \end{aligned}$$

Now suppose that only signals b and c are observable, and a signal trace of this partial observability is obtained below.

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

During the trace abstraction step, 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, $\{e_1, e_2\} \times \{e_1, e_2\} \times \{e_3\}$.

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.

$$\begin{aligned} e_4 &: \ abc\ \bar{a}bc \\ e_5 &: \ abc\ abc\ abc\ \bar{a}bc \end{aligned}$$

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 trace of flow events.

From the above discussions, it can be seen that a signal trace of partial observability is generally mapped to a set of flow traces. As the the number of signals available for observation becomes smaller, the number of flow event traces corresponding to a signal trace as a result of the abstraction can be enormous. This phenomenon can increase the complexity of the trace interpretation as it can cause the number of possible flow execution scenarios generated during the analysis to explode. Possible solutions to address this issue include better trace signal selection for observation and assistance from debuggers' insights. Trace signal selection itself is an important and difficult subject, and a detailed discussion of it is out of scope of this paper. Next, we briefly describe how the debuggers' insights of a system's architecture can help to address the complexity issue in the trace analysis.

B. Inputs from Validators

During the trace interpretation, the number of intermediate flow execution scenarios may become too large due to ambiguous interpretation from signal events to flow events or from flow events to flow execution scenarios. The explosion of the intermediate results can significantly slow down the performance of the trace analysis. To address this problem, the debuggers can use their insights and understanding of the SUD to trim the total number of possible system executions derived from the trace analysis. When a SUD is validated, a debugger is assumed to have deep knowledge about the system's architecture and microarchitecture, and how the test environment affects the system executions. For instance, he/she may have knowledge that in a test environment, the maximal number of instances of a flow can be activated by the SUD, or a flow can be activated after certain other flows have terminated, etc. These debugging insights can be encoded as constraints into the trace analysis method, which can then be used to eliminate a large number of flow execution scenarios that violate these constraints during the trace interpretation step.

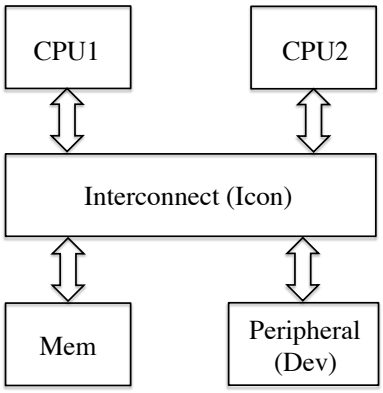


Fig. 3. A simple SoC example.

This approach can be flexible in that it allows a debugger to analyze the observed traces in a trail-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.

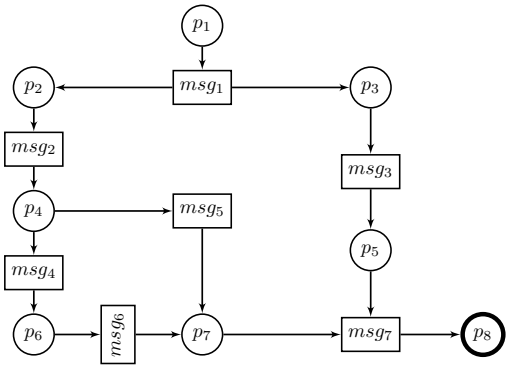
Alternatively, if all derived execution scenarios seem to be plausible, the implication that a debugger may draw from this result is that the failure may be independent of the flows being observed. Therefore, the testing environment can be adjusted in order for a different part or different behavior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

IV. Case Study

The proof concept of the proposed trace analysis method is demonstrated on a transaction level model of a simple SoC design with two CPU cores, memory, and a peripheral device connected by an interconnect as shown in Fig. 3. Since the analysis method presented in this paper is communication centric, the detailed computations of these blocks are not modeled. Instead, the modeling is focused on how they participate in flows for different system level use cases.

A. Flow Specification

In this case study, four system flows are implemented in the simple SoC model. They include cache coherent memory access operations and a memory-mapped peripheral read operation initiated from the CPUs, a message signaled interrupt operation initiated from the peripheral device. These flow specifications capture how messages are exchanged for different use cases. In this model, a message is defined with the following format, $(\text{Src}, \text{Dest}, Cmd, Addr)$, where Src and Dest



Definition of the messages:

$$
\begin{array}{lllll}
msg_1: & (\text{CPU2}, & \text{Icon}, & Wr, & M) \\
msg_2: & (\text{Icon}, & \text{CPU1}, & Wr, & M) \\
msg_3: & (\text{CPU2}, & \text{Icon}, & DVal, & -) \\
msg_4: & (\text{CPU1}, & \text{Icon}, & Hit, & -) \\
msg_5: & (\text{CPU1}, & \text{Icon}, & Miss, & -) \\
msg_6: & (\text{CPU1}, & \text{Icon}, & DVal, & -) \\
msg_7: & (\text{Icon}, & \text{Mem}, & Wr, & M)
\end{array}
$$

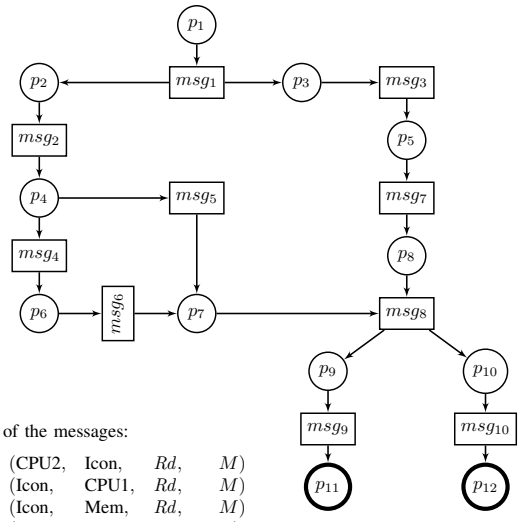
Fig. 4. Flow specification (F_1) of a cache coherent write operation initiated from `CPU2`.

refer to the source and destination components of messages, Cmd refers to the operations that the destination component should perform, and $Addr$ refers to memory addresses where Cmd applies. Cmd can be memory accesses or not. If it is not, then the $Addr$ field of messages is ignored. In this case study, memory mapped IO mechanism is used. Furthermore, detailed memory addresses are not modeled. Instead, the address space is partitioned to main memory addresses and peripheral addresses. In messages, the $Addr$ field is replaced with either M representing a memory address or P representing an address to a peripheral device.

The LPN as shown in Fig. 4 specifies a system flow where CPU2 initiates a memory write operation. In this flow, CPU2 initiates a memory write request followed by a data valid message to the interconnect. The data valid messages are used to model availability or validity of data for transfer. Concurrently, the interconnect inquiries CPU1 if it holds a more updated version of the data by sending a memory write message msg_2. CPU1 generates one of two possible responses. If CPU1 holds the more updated data in its cache in the same memory space that CPU2 intends to write, a cache hit message msg_4 followed by msg_6 are sent to the interconnect. Otherwise, CPU1 sends a cache miss message msg_5. After getting the response from CPU1, Interconnect sends a write request to the memory unit. This flow is symmetric for CPU1.

The LPN specification as shown in Fig. 5 captures the system flow where CPU2 initiates a memory read operation. Basically, CPU2 sends a memory read message to Interconnect, which then generates two concurrent threads, one checks if CPU1 has the more updated data in the memory space for the read operation, and the other thread to get data from the memory. Once Interconnect gets both responses from CPU1

and memory, it synchronizes the responses, and writes the correct data to the CPU2's cache and memory in parallel. Again, this specification is symmetric for CPU1.



Definition of the messages:

msg_1 :	(CPU2,	Icon,	Rd,	M)
msg_2 :	(Icon,	CPU1,	Rd,	M)
msg_3 :	(Icon,	Mem,	Rd,	M)
msg_4 :	(CPU1,	Icon,	Hit,	$-$)
msg_5 :	(CPU1,	Icon,	$Miss$,	$-$)
msg_6 :	(CPU1,	Icon,	$DVal$,	$-$)
msg_7 :	(Mem,	Icon,	$DVal$,	$-$)
msg_8 :	($-$,	$-$,	$-$,	$-$)
msg_9 :	(Icon,	Mem,	Wr,	M)
msg_{10} :	(Icon,	CPU2,	Wr,	M)

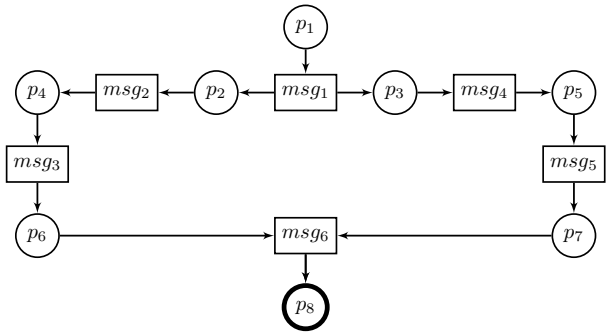
Fig. 5. Flow specification (F_2) of a cache coherent read operation from CPU2.

The LPN specification as shown in Fig. 6 captures a system flow for CPU initiated memory mapped peripheral read operations. When CPU2 tries to read the peripheral device (device hereafter), a read message with address P is sent to Interconnect, which then sends this message to CPU1 and the device simultaneously. When CPU1 sees this message, it responds with a cache miss message as the address P points to the device. At the meantime, the device responds with a data valid message indicating the availability of the requested data. Finally, Interconnect synchronizes the both responses, and sends a data valid message back to CPU2. This specification is also symmetric for CPU1.

The last LPN specification as shown in Fig. 7 captures how interrupts from the device are handled. In this case study, all interrupts are directed to CPU1. When the device triggers an interrupts, it sends a message with $Intr$ in the command field. Then, Interconnect notifies CPU1 by sending a message with MSI and I in the command and address fields, respectively, where I is the symbol referring to the entry points to interrupt service routines. CPU1 responds with a cache miss message as the receipt of the interrupt.

B. Results and Discussions

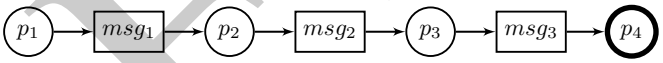
The transaction level model of the simple system implementing the four flow specifications shown in the previous section is described in SystemC. Each component is described as a SystemC module, which may include a number of threads



Definition of the messages:

msg_1 :	(CPU2,	Icon,	Rd,	P)
msg_2 :	(Icon,	CPU1,	Rd,	P)
msg_3 :	(CPU1,	Icon,	$Miss$,	$-$)
msg_4 :	(Icon,	Dev,	Rd,	P)
msg_5 :	(Dev,	Icon,	$Dval$,	$-$)
msg_6 :	(Icon,	CPU2,	$Dval$,	$-$)

Fig. 6. Flow specification (F_3) of a read access to peripheral device from CPU2.



Definition of the messages:

msg_1 :	(Dev,	Icon,	$Intr$,	$-$)
msg_2 :	(Icon,	CPU1,	MSI,	I)
msg_3 :	(CPU1,	Icon,	$Miss$,	$-$)

Fig. 7. Flow specification (F_4) of handling interrupts from the peripheral device.

to model concurrency. The entire model is concurrent and operates in asynchronous mode.

When testing this model, both CPUs and the peripheral device are set up as flow instance generators. They randomly generate the first message in a corresponding flow specification to start a flow instance, and react to incoming messages by generating new messages as defined in the flow specification. In the model, monitors are embedded to observe the messages generated by each component. When a message is observed, it is written to an output trace file for analysis.

Even though this example is conceptually simple, getting the model to correctly implement the flow specifications is not straightforward. On the other hand, results from the trace analysis greatly help the debugging process by providing information to locate problems quickly. For example, in an early version of the model, the trace shown in Table I is observed. The trace analysis finds out that messages $1-8$ in the trace are the results of execution of an instance of flow F_2 as shown, and the flow execution scenario is $\{(F_{2,1}, \{p_{11}, p_{12}\})\}$. Messages $9-11$ are analyzed as the results of executing another instance of flow F_2, message $12-13$ as the results of executing an instance of flow F_3 as shown in Fig. 6. The flow execution scenario after the first thirteen messages in the

TABLE I
AN OBSERVED TRACE OF MESSAGES FOR TRACE ANALYSIS.

1 (CPU2, Icon, Rd, M)	2 (Icon, CPU1, Rd, M)	3 (Icon, Mem, Rd, M)	4 (Mem, Icon, DVal, -)
5 (CPU1, Icon, Hit, -)	6 (CPU1, Icon, DVal, -)	7 (Icon, CPU2, Wr, M)	8 (Icon, Mem, Wr, M)
9 (CPU2, Icon, Rd, M)	10 (Icon, CPU1, Rd, M)	11 (Icon, Mem, Rd, M)	12 (CPU2, Icon, Rd, P)
13 (Icon, CPU1, Rd, P)	14 (CPU1, Icon, Miss, -)	15 (Icon, Dev, Rd, P)	16 (CPU1, Icon, DVal, -)

trace is

$$\{(F_{2,1}, \{p_{11}, p_{12}\}), (F_{2,2}, \{p_4, p_5\}), (F_{3,1}, \{p_4, p_3\})\}.$$

Message 14 can be the result from executing $F_{2,2}$ or $F_{3,1}$, therefore it leads to two following flow execution scenarios:

$$\{(F_{2,1}, \{p_{11}, p_{12}\}), (F_{2,2}, \{p_7, p_5\}), (F_{3,1}, \{p_4, p_3\})\} \quad (1)$$

$$\{(F_{2,1}, \{p_{11}, p_{12}\}), (F_{2,2}, \{p_4, p_5\}), (F_{3,1}, \{p_6, p_3\})\} \quad (2)$$

In either scenario, after mapping message 15 to flow $F_{3,1}$, message 16 cannot be mapped to any existing flow instance or to a new flow instance. From this inconsistent message, we know that it is generated by CPU1. In scenario (1), CPU1 is in the state after generating the message reporting a cache miss. In this case, the DataValid message should not generated. In scenario (2), CPU1 is in the state before generating the message reporting either a cache hit or miss, and again the DataValid message should not generated. This inconsistent message helps to locate a bug in the CPU1 model where a DataValid message is generated after either a cache hit or miss message is generated. After fixing this bug, in a few more iterations of analysis and debugging, the trace analysis can eventually extract all initiated flow instances in the model, all in their terminal states, thus showing that the model implements the four flows correctly.

In the second experiment, partial observability is taken into account during the trace analysis with the assumption that the command Wr and Rd and addresses M and P are indistinguishable due to the lack of observability. This partial observability is simulated with a modification to the monitors such that in each observed message, command Wr or Rd is replaced with (Wr, Rd) and address M or P is replaced with (M, P). The version of the model generating the trace shown in Table I is reused with the modified the monitors, and the generated trace with the simulated partial observability is shown in Table II. Similarly, only the first sixteen messages are shown.

Each message in the trace with partial observability is referred to as an *super* message to distinguish it from the messages of full observability. The traces of super messages are referred to as *super* traces. For example, the first super message in the trace from Table II, (CPU2, Icon, (Wr, Rd), (M, P)), corresponds to four distinct messages: (CPU2, Icon, Wr, M), (CPU2, Icon, Wr, P), (CPU2, Icon, Rd, M), and (CPU2, Icon, Rd, P). Some of these messages do not exist in the flow specification, and are ignored during the trace analysis. In the above example, message (CPU2, Icon, Wr, P) is ignored.

Each super trace represents a set of traces, each of which is interpreted to derive a set of flow execution scenarios. Due to the partial observability, the number of traces represented by a super trace can become very large. For example, the super trace shown in Table II represents about twelve thousand possible traces for that short sequence of messages. The trace analysis algorithm returns the set of flow execution scenarios for each trace for examination, and a very large number of possible flow execution scenarios can be generated. The large number of possible flow execution scenarios not only produces too much information that can overwhelm system validators, but also degrades the performance of the trace analysis algorithm by consuming too much memory. As indicated above, the validators' insights on the SUD can be utilized to trim the possibilities. For example, if the validator knows that no flow F_1 in Fig. 4 is activated in the testing environment, this insight helps to eliminate all flow execution scenarios that include instances of F_1 by interpreting message $\#1$ as either (CPU2, Icon, Rd, M) or (CPU2, Icon, Rd, P). Consider another insight such that a new instance of flow F_2 as in Fig. 5 can be initiated only after the completion of the previous instance of F_2. If an instant of F_2 is assumed to be initiated by the super message $\#9$ (CPU2, Icon, (Wr, Rd), (M, P)) by interpreting it to (CPU2, Icon, Rd, M) during the trace analysis, the super message $\#12$ can only be interpreted to (CPU2, Icon, Wr, M) or (CPU2, Icon, Rd, P) as the instance of F_2 initiated by the super message $\#9$ has not been completed yet at this point. According to the above discussion, the validators' insights help restrict how super messages are interpreted, thus reducing the number of flow execution scenarios that can be generated. At the end of the analysis, all possible flow execution scenarios are returned to system validators for examination.

V. CASE STUDY II

The proof concept of the proposed trace analysis method is demonstrated on a transaction level model of a SoC design from GEM5 with two ARM CPU cores, each with 16KB data cache and 16KB instruction cache. A simple memory controller with 1GB capacity is connected with CPUs and caches by an interconnect bus as shown in Fig. 8. Since the analysis method presented in this paper is communication centric, the detailed computations of these blocks are not modeled. Instead, the modeling focuses on how they participate in flows for different system level use cases. Originally all of the components are connected by ports in GEM5. Ports

TABLE II
A TRACE OF MESSAGES WITH PARTIAL OBSERVABILITY FOR TRACE ANALYSIS.

1 (CPU2, Icon, (Wr, Rd), (M, P))	2 (Icon, CPU1, (Wr, Rd), (M, P))	3 (Icon, Mem, (Wr, Rd), (M, P))	4 (Mem, Icon, DVal, -)
5 (CPU1, Icon, (Hit, Miss), -)	6 (CPU1, Icon, DVal, -)	7 (Icon, CPU2, (Wr, Rd), (M, P))	8 (Icon, Mem, (Wr, Rd), (M, P))
9 (CPU2, Icon, (Wr, Rd), (M, P))	10 (Icon, CPU1, (Wr, Rd), (M, P))	11 (Icon, Mem, (Wr, Rd), (M, P))	12 (CPU2, Icon, (Wr, Rd), (M, P))
13 (Icon, CPU1, (Wr, Rd), (M, P))	14 (CPU1, Icon, (Hit, Miss), -)	15 (Icon, Dev, (Wr, Rd), (M, P))	16 (CPU1, Icon, DVal, -)

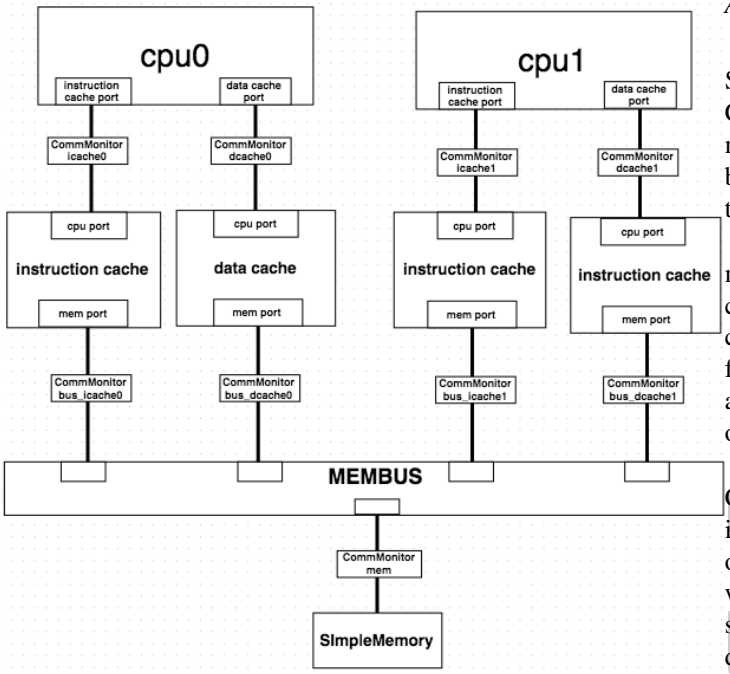


Fig. 8. SoC platform structure.

exist in peers, and by connecting them, one component can send request or response through ports. In order to trace the communication inside the system, we added an communication monitor between each components connection. The communication monitor is a MemObject that monitors the communication between two ports in the system.

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.

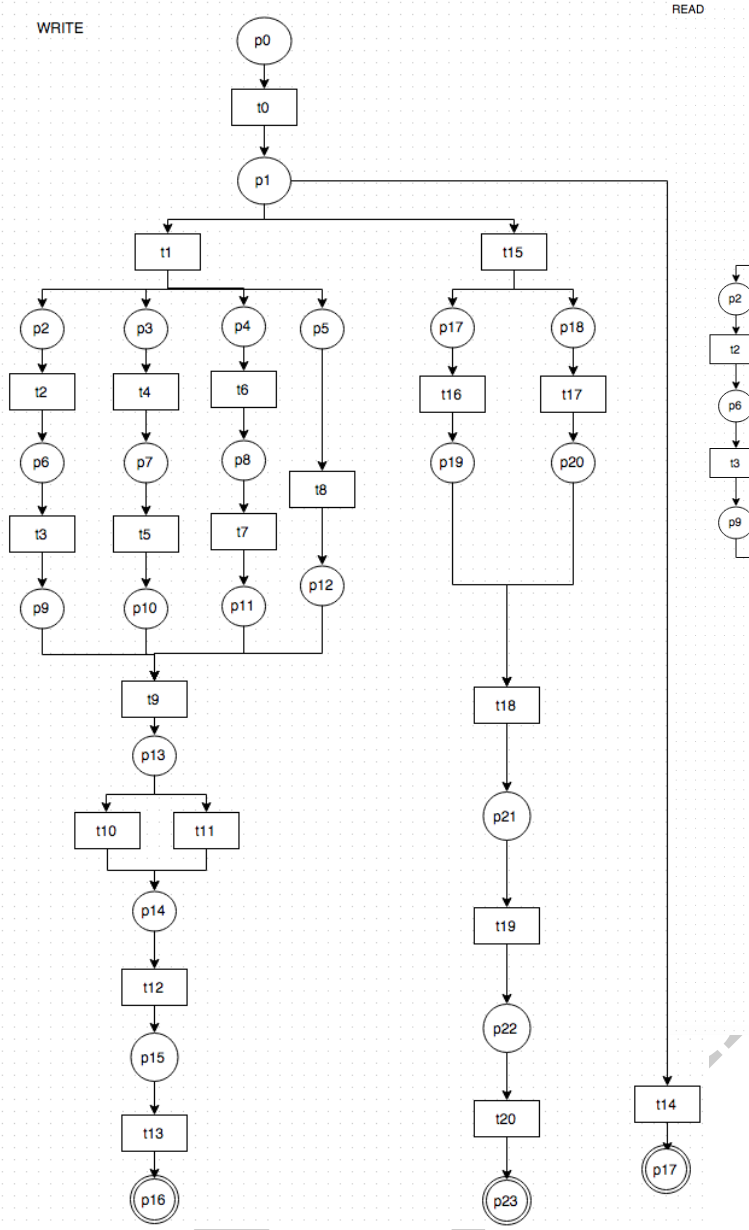
A. Flow Specification

In this case study, two system flows are implemented in the SoC model. They include a cache coherent read request from CPU and a cache coherent write request from CPU. To avoid messages that are not included in these two flows, like write back and cache flushing, we only use data that are related to these two flows.

In the following section, a brief overview for the implemented instances of these flows are given. These flow specifications capture how messages are exchanged for different use cases. In this model, a message is defined with the following format, (Src,Dest,Cmd), where Src and Dest refer to the source and destination components of messages, Cmd refers to the operations that the destination component should perform.

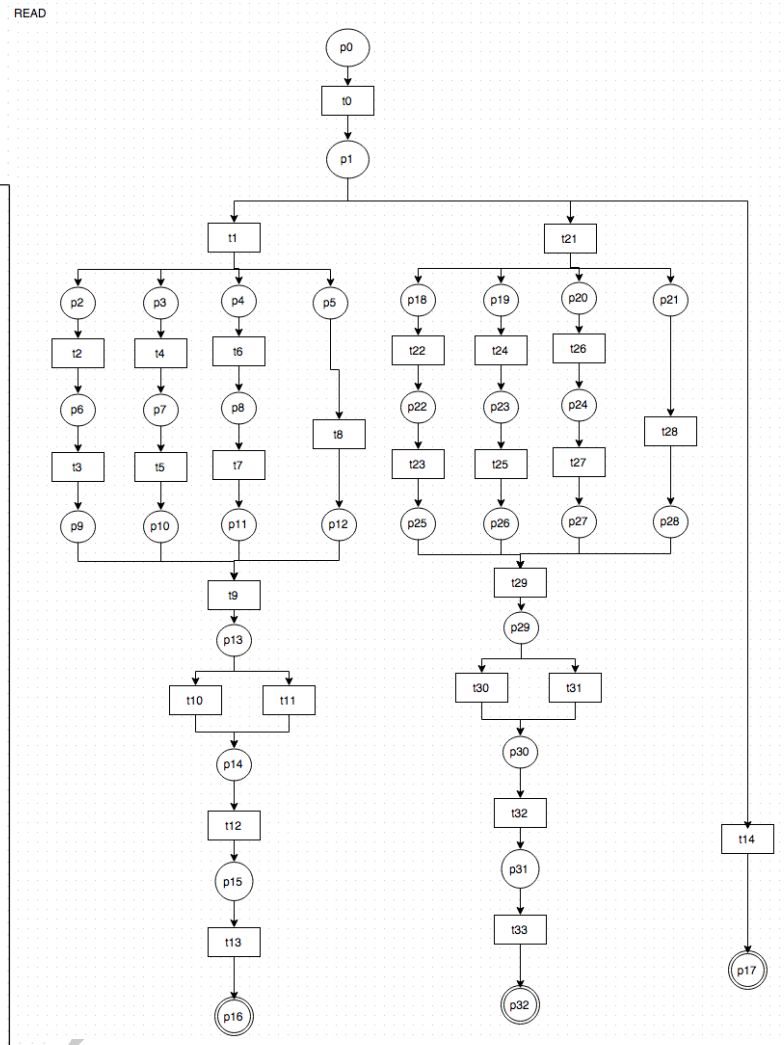
The LPN as shown in Fig. 9 specifies a system flow where CPU1 initiates a memory write operation. In this flow, CPU1 initiates a memory write request to L1 Cache. Next, depends on whether the required data is included in cache1, the cache1 will generate three possible responses. First, the cache1 will send read exclusive request message msg2 to interconnect if data is not present in cache.; Or if the data is shared by CPU0, it will send upgrade message msg16 to interconnect to disable CPU0's ownership of this block of data, else if CPU1 has exclusive right of required data, cache1 will perform the write operation and sent an response message msg15 to CPU1. Afterwards, interconnect will sent request to all connected component (data cache and instruction cache of each CPU and memory). Once the interconnect obtains the response from either CPU0 or memory, it will generate a response message to cache1, then cache1 will generate a write response message msg14 (or msg 21) to CPU1. The flow is symmetric for CPU2.

The LPN specification as shown in Fig. 10 captures the system flow where CPU1 initiate a memory read operation. Similar to memory write, CPU1 will first initiate a memory read request message msg1 to cache1. Cache1 can generate three possible responses. First, if the requested data is not in cache1, cache1 will generate an storeCondReq message msg2 to interconnect asking for data. Second, if the requested data is inside of cache1 but it's shared, cache1 will generate a upgrade request message msg22 to make sure it has the newest data; Last, if cache1 has and data and it's not shared, it will generate a read response message msg15 to CPU1. Afterwards, interconnect will send corresponding request to all of its connected components. When interconnect receives the response, it will generate response message to cache1. Then cache1 will generate the read response message msg14



msg_0 : (CPU1, writeReq, cache1)
msg_1 : (cache1, readExreq , Bus)
msg_2 : (Bus, readExreq, cahce2)
msg_3 : (cache2, readExreq, cpu2)
msg_4 : (Bus, readExreq, cahce2)
msg_5 : (cache2, readExreq, cpu2)
msg_6 : (Bus, readExreq, cahce1)
msg_7 : (cache1, readExreq, cpu1)
msg_8 : (Bus, readExreq, Memory)
msg_9 : (true)
msg_{10} : (Memory, readExres, Bus)
msg_{11} : (cache2, readExres, Bus)
msg_{12} : (Bus, readExres, cache1)
msg_{13} : (cache1, writeRes, CPU1)
msg_{14} : (cache1, writeRes, CPU1)
msg_{15} : (cache1, UpgradeReq)
msg_{16} : (Bus, UpgradeReq, cahce2)
msg_{17} : (Bus, UpgradeReq, Memory)
msg_{18} : (cache2, UpgradeRes, Bus)
msg_{19} : (Bus, UpgradeRes, cache1)
msg_{20} : (cache1, WriteRes, CPU1)

Fig. 9. Flow specification (F_1) of a cache coherent write operation initiated from CPU1



$t0$: (CPU1, ReadReq, cache1)
$t1$: (cache1, StoreCondreq , Bus)
$t2$: (Bus, StoreCondreq, cahce2)
$t3$: (cache2, StoreCondreq, cpu2)
$t4$: (Bus, StoreCondreq, cahce2)
$t5$: (cache2, StoreCondreq, cpu2)
$t6$: (Bus, StoreCondreq, cahce1)
$t7$: (cache1, StoreCondreq, cpu1)
$t8$: (Bus, StoreCondreq, Memory)
$t9$: (true)
$t10$: (Memory, ReadRes, Bus)
$t11$: (cache2, ReadRes, Bus)
$t12$: (Bus, ReadRes, cache1)
$t13$: (cache1, ReadRes, CPU1)
$t14$: (cache1, ReadRes, CPU1)
$t21$: (cache1, LoadLockedreq , Bus)
$t22$: (Bus, LoadLockedreq, cahce2)
$t23$: (cache2, LoadLockedreq, cpu2)
$t24$: (Bus, LoadLockedreq, cahce2)
$t25$: (cache2, LoadLockedreq, cpu2)
$t26$: (Bus, LoadLockedreq, cahce1)
$t27$: (cache1, LoadLockedreq, cpu1)
$t28$: (Bus, LoadLockedreq, Memory)
$t29$: (true)
$t30$: (Memory, ReadRes, Bus)
$t31$: (cache2, ReadRes, Bus)
$t32$: (Bus, ReadRes, cache1)
$t33$: (cache1, ReadRes, CPU1)

Fig. 10. Flow specification (F_2) of a cache coherent read operation initiated from CPU1

or msg34 to CPU1. This specification is also symmetric for CPU2.

B. Results and Discussions

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.

As we mentioned before, we will take away unrelated messages for easier testing purpose. After taking out non-timing request, the number of communication messages reduced from 343581 to 133676, and after removing the message that's not related to tested flow specification, number of messages became 120949.

The total time we used for processing the filtered data is 3 seconds, with 12MB data usage.

Even though this example is conceptually simple, getting the model to correctly implement the flow specifications is not straightforward. On the other hand, results from the trace analysis greatly help the debugging process by providing information to locate problems quickly. For example, the following trace as shown in Table III is observed. The trace analysis finds out that messages 1-18 maps to two CPU0 read as

$$\{(F2^1, < p29 >), (F2^2, < p13 >)\}$$

message 19 can be result of either flow instances, therefore it leads to two floowling flow execution scenarios:

$$\{(F2^1, < p30 >), (F2^2, < p13 >)\}$$
$$\{(F2^1, < p29 >), (F2^2, < p14 >)\}$$

In either scenario, message 20 can't be mapped to any existing flow instance or to a new flow instance. From this example, we can see that the error is originated by CPU0, and the problem is cache0 send the read response to CPU0 before it received any response from memory bus.

VI. Related Work

Our work is closely related to communication-centric and transaction based debug. An early pioneering work is described in [4], 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 [5], [6]. 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 [7]. 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 [8] to address these issues. This approach utilizes a transaction debug pattern specification language [9] 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 [10] elevated to the transaction level.

In [11], 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 [12], 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.

VII. Conclusion

This paper presents a trace analysis based method for post-silicon validation by interpreting observed raw signal traces

TABLE III
EXAMPLE OF TRACE RESULT

1 (cpu0,cache0,readreq)	2 (cache0,membus,LoadLockedreq)	3 (membus,mem,LoadLockedreq)	4 (membus,cache0,LoadLockedreq)
5 (cache0,cpu0,LoadLockedreq)	6 (cpu0,cache0,readreq)	7 (cache0,membus,StoreCondreq)	8 (membus,mem,StoreCondreq)
9 (membus,cache0,StoreCondreq)	10 (cache0,cpu0,StoreCondreq)	11 (membus,cache1,StoreCondreq)	12 (cache1,cpu1,StoreCondreq)
13 (membus,cache1,StoreCondreq)	14 (cache1,cpu1,StoreCondreq)	15 (membus,cache1,LoadLockedreq)	16 (cache1,cpu1,LoadLockedreq)
17 (membus,cache1,LoadLockedreq)	18 (cache1,cpu1,LoadLockedreq)	19 (mem,membus,readres)	20 (cache0,cpu0,readres)
20 (membus,cache0,readres)	22 (mem,membus,readres)	23 (membus,cache0,readres)	24 (cache0,cpu0,readres)

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 paper 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.

REFERENCES

[1] D. Harel and P. S. Thiagarajan, "Message sequence charts," in *In UML for Real: Design of Embedded Real-Time Systems*. Kluwer Academic Publishers, 2003, pp. 77–105.

[2] W. Damm and D. Harel, "Lscs: Breathing life into message sequence charts," in *Formal Methods for Open Object-Based Distributed Systems*, ser. IFIP The International Federation for Information Processing, P. Ciancarini, A. Fantechi, and R. Gorrieri, Eds. Springer US, 1999, vol. 10, pp. 293–311.

[3] S. Krstic, J. Yang, D. Palmer, R. Osborne, and E. Talmor, "Security of soc firmware load protocols," in *Hardware-Oriented Security and Trust (HOST), 2014 IEEE International Symposium on*, May 2014, pp. 70–75.

[4] K. Goossens, B. Vermeulen, R. v. Steeden, and M. Bennebroek, "Transaction-based communication-centric debug," in *Proceedings of the First International Symposium on Networks-on-Chip*, ser. NOCS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 95–106.

[5] B. Vermeulen and K. 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*, ser. VLSI-DAT '09, 2009, pp. 183–186.

[6] K. Goossens, B. Vermeulen, and A. B. Nejad, "A high-level debug environment for communication-centric debug," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, pp. 202–207.

[7] A. M. Gharehbaghi and M. Fujita, "Transaction-based post-silicon debug of many-core system-on-chips," in *ISQED*, 2012, pp. 702–708.

[8] M. Dehbashi and G. 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*, ser. PDP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 400–404.

[9] A. M. Gharehbaghi and M. Fujita, "Transaction-based debugging of system-on-chips with patterns," in *Proceedings of the 2009 IEEE International Conference on Computer Design*, ser. ICCD'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 186–192.

[10] M. Boule, J.-S. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," in *Proceedings of the 8th International Symposium on Quality Electronic Design*, ser. ISQED '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 613–620.

[11] E. Singerman, Y. Abarbanel, and S. Baartmans, "Transaction based pre-to-post silicon validation," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 564–568.

[12] Y. Abarbanel, E. Singerman, and M. 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*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 2:1–2:4.